Efficient Asynchronous I/O with Request Merging

Md Kamal Hossain Chowdhury
Department of Computer Science
The University of Alabama
Tuscaloosa, USA
mhchowdhury@crimson.ua.edu

Houjun Tang
Scientific Data Division
Lawrence Berkeley National Laboratory
Berkeley, USA
htang4@lbl.gov

Jean Luca Bez
Scientific Data Division
Lawrence Berkeley National Laboratory
Berkeley, USA
jlbez@lbl.gov

Purushotham V. Bangalore
Department of Computer Science
The University of Alabama
Tuscaloosa, USA
pvbangalore@ua.edu

Suren Byna
Department of Computer Science and Engineering
The Ohio State University
Columbus, USA
byna.1@osu.edu

***Abstract*—With the advancement of exascale computing, the amount of scientific data is increasing day by day. Efficient data access is necessary for scientific discoveries. Unfortunately, the I/O performance is not improved, like the CPU and network speed. So, I/O operations take longer time than data generation or analysis. Asynchronous I/O has been proposed to extenuate the I/O bottleneck by overlapping I/O and computation time. However, multiple small write operations can diminish the benefits of asynchronous I/O, as the I/O time becomes significantly longer than the compute time, with little time to overlap with. To overcome these issues, we present an optimization technique to merge small contiguous write operations. We integrated our solution into the HDF5 asynchronous I/O VOL connector and demonstrated the effectiveness of merging HDF5 write operations automatically and transparently without requiring any code change from the application.**

I. Introduction

As exascale high-performance computing (HPC) systems become available in the coming years, new challenges for scientific data management emerge. With the increasing computation power, an unprecedented amount of data is expected to be generated [1]. Storing and accessing such large amounts of data efficiently remains a difficult task, as the I/O operations' efficiency often limits the overall application performance and scientific productivity. Scientific Applications such as earthquake simulations [2]–[4] and cosmological simulations [5] involve massive amounts of data write operations. The ever-increasing volumes of data need to be stored, shared, analyzed, and visualized, and poor I/O performance can hamper such workflow and reduce scientific productivity.

To speed up the data access efficiency, asynchronous I/O has been proposed and shown to be an effective approach [6]–[9]. The total application runtime can be significantly reduced by overlapping the I/O with sufficient computation time. However, the existing asynchronous I/O approaches execute the I/O operations on background threads [9] or dedicated server processes [6]. If an application is performing a large number of small write requests, the I/O time can still be very long and may exceed the computation time that it can overlap with, resulting in partially hidden I/O time and limited overall performance improvement. Additionally, longer I/O time can lead to increased resource contention among the computation, communication, and asynchronous I/O operations, and slows down the application [10].

To address this issue, we propose to optimize the asynchronous I/O by merging multiple small write operations into fewer large and contiguous writes, which could be much faster when writing to the parallel file system. The asynchronous I/O frameworks enable such optimization as they can accumulate a number of operations in a queue before executing them. With the asynchronous VOL connector [9], a background thread is launched together with an application. I/O operations are intercepted and converted into asynchronous tasks, which are then added to a task queue. The background thread monitors the application's activity and only triggers the I/O operations when the application is performing non-I/O operations. By inspecting the queued I/O tasks, we can extract the offsets and sizes of the write requests, and merge those that can form a larger contiguous chunk. This is especially useful for applications that produce time-series data, with each writer appending a small amount of data to the previously written datasets. To minimize the impact on the application, we have (a) developed a highly efficient algorithm that detects and merges compatible write operations and (b) integrated it into the HDF5 asynchronous I/O VOL connector. Our work is fully automatic and transparent, with no requirement to change the application's code as long as they use the asynchronous I/O VOL connector.

In summary, our method makes the following contributions:

- We propose an I/O optimization strategy that merges multiple small write operations into fewer and larger contiguous writes.
- Our solution supports up to 3-dimensional data and supports merging out-of-order write operations.
- We demonstrate the effectiveness of our solution compared with existing work using benchmarks with various workloads at different scales.

The rest of the paper is structured as follows. We review related literature in Section II. We provide the background of the asynchronous I/O framework in Section III, and explain the implementation of the merge operation in Section IV. In Section V, we describe the experimental setup and present the parallel I/O results of the experiments. Finally, we conclude the paper in Section VI.

II. Related Work

Merging I/O operations has been used to save I/O time in different fields. We briefly cover its use in distinct contexts.

Hard-Disk Drives (HDDs) are commonly used with various optimizations to provide the necessary throughput and low latency for mail, web, file, database, and backup/archive data services, as well as for desktop environments where instant responsiveness is desired. In order to meet the ever-increasing performance requirements of big data analysis software, internet-scale online services, and other diverse application classes, Yu et al. [11] presented reordering or merging requests in a request queue to maximize the amount of data within a single I/O request.

In Deep Learning (DL) and cloud computing platforms, file systems are an essential component. However, in DL datasets, many small files can lead to a performance penalty when using Hadoop Distributed File System (HDFS). To address the issue, Zhu et al. [12] propose Pile-HDFS (PHDFS), which incorporates a new file aggregation method to improve performance where I/O units combine small files based on their correlation.

Performance variability in solid-state drives (SSDs) is a common issue caused by garbage collection (GC). Commercial off-the-shelf SSDs may experience unexpected drops in throughput when workloads have a higher percentage of writes. This is because GC is triggered to clean invalid pages, producing free space, but incoming requests may be pending in the I/O queue, delaying their service until GC is finished. This problem is particularly severe for bursty write-dominant workloads commonly found in server-centric enterprise or HPC workloads. To address this issue, Lee et al. [13] propose merging I/O requests with internal GC I/O requests and demonstrating a significant performance improvement.

The rapid increase in genomics data has significantly increased the computational and data-intensive nature of searching sequence databases. Lin et al. [14] explore the computation and I/O scheduling for irregular, data-intensive scientific applications in the context of genomics sequence search. They report that a lack of coordination between computation scheduling and I/O optimization can negatively impact performance. They propose integrated scheduling to improve sequence-search throughput by coordinating computation load balancing and high-performance non-contiguous I/O.

Mobile devices are becoming increasingly integrated into people's daily lives, and system responsiveness is crucial for maintaining a positive user experience. Despite advancements in technology, mobile devices still experience unpredictable delays. Wu et al. [15] found that improper merging operations in the I/O scheduler layer significantly contribute to these delays. To reduce system latency caused by large merged requests, they propose a new dynamic merging/splitting-based I/O scheduling approach to enhance system responsiveness.

I/O deduplication is widely used for conserving storage space and reducing I/O load. This technique benefits various systems, including storage servers in data centers, personal devices, and field-deployed sensing systems. Moreover, a deduplication system uses metadata, including information about mapping logical blocks to physical blocks, the number of references to physical blocks, and unique identifiers for blocks (fingerprints), to ensure consistency in storage even after system failures occur. I/O deduplication is an effective way to save storage space and decrease the number of costly flash writes. By implementing soft updates-style metadata write ordering, storage data consistency can be maintained without incurring additional I/O. Moreover, the anticipatory I/O delay optimization is particularly effective for increasing the chances of merging metadata when working with a strong I/O persistence model [16].

I/O is a major bottleneck for data-intensive scientific applications in high-performance computing (HPC) systems and leadership-class machines, I/O is a significant bottleneck for data-intensive scientific applications. These systems may struggle to handle the high volume and intensity of I/O requests, leading to bottlenecks. I/O forwarding is used to aggregate and delegate I/O requests to storage systems to address this issue. Ohta et al. [17] present a method for merging I/O requests and scheduling their delegation to parallel file systems, resulting in improved application I/O throughput.

Decentralized file systems have been plunged for the load asymmetry of different nodes and the scalability issues. Due to the absence of a metadata server in decentralized file systems, clients and servers need to send more RPC requests to control the metadata processing, which badly impacts the I/O performance, and also causes traffic imbalance due to increased RPC latency. An et al. [18] proposed an I/O scheme to reduce the number of RPC requests. Instead of a single RPC request at a single time, they queued the RPC requests and merged them into a larger RPC request, thus skipping huge RPC overhead.

Inspired by the above-mentioned research papers, we applied the idea of merging multiple write operations into one to improve the I/O performance. Our work is based on the HDF5 asynchronous I/O VOL connector [8], which accumulates a number of write operations in a queue, and allows us to check for contiguous writes and merge the compatible ones.

III. Background

This section briefly discusses how high-level interfaces such as HDF5 can be used with asynchronous I/O and our merge optimization to improve I/O performance.

A. HDF5

The Hierarchical Data Format (HDF) [19] provides a system for organizing and storing data, including an abstract data model and storage format. The HDF5 library offers a programming interface that implements these abstract models, as well as a method for efficiently transferring data between stored representations. HDF5 is a widely used high-level I/O library in many scientific domains [20]–[22]. HDF5 relies on the user to define where the data should be written, using the data space creation and hyperslab or point selection.

B. HDF5 Virtual Object Layer (VOL)

HDF5 provides the Virtual Object Layer (VOL) [23] interface, an intermediary for all HDF5 API calls that might access objects in a file. These calls can be redirected to an external VOL connector that can change the I/O behavior, which provides the architecture support for performing I/O operations asynchronously. The external VOL connectors are dynamic link libraries, and can be loaded by the HDF5 library through an environment variable during runtime,

C. HDF5 Asynchronous I/O VOL connector

Asynchronous I/O is proposed as a solution to reduce the I/O time. It performs I/O while allowing other tasks to proceed before the I/O operation is completed. The existing asynchronous I/O VOL connector [8], [9] supports queuing several I/O tasks and then executing them in background threads so that the application's main process can proceed without waiting for their completion.

With the HDF5 asynchronous I/O VOL connector, every I/O operation creates a task object. The task object holds all the information needed for the execution, including a copy of I/O parameters, a function pointer to execute the operation, data pointers, and internal states like its dependency and execution status. The background thread's execution engine added the task to a queue, which manages the task dependency and is hidden from the user. After a task is created, the corresponding function returns to the application without blocking it. The execution engine decides which task will be executed and passes the task's information to the background thread for execution.

The asynchronous I/O may take the same or longer time (due to overhead) than synchronous I/O. To further improve the I/O performance and take advantage of the queued tasks in an asynchronous I/O framework, we propose a new optimization that can merge multiple compatible write operations into fewer contiguous writes, as writing the same amount of data in multiple operations are much slower than writing them in one or a few write requests.

IV. Write Request Merge

We have developed an efficient algorithm that detects and merges small write operations to improve the asynchronous I/O performance, as described in Algorithm 1. Currently, it supports up to 3-dimensional data, but it can be extended to support higher-dimensional data with the same logic. Additionally, it can also be applied to merge read requests.

To check whether two write requests can be merged, we need to extract their data selection information first. This

Algorithm 1: Data selection merge

```
Data: $W0(off_0[], cnt_0[]), W1(off_1[], cnt_1[])$
Result: merged write, $W2(off_2[], cnt_2[])$
if dimension==1 then
    if $off_0[0] + cnt_0[0] == off_1[0]$ then
        $off_2[0] = off_0[0]$
        $cnt_2[0] = cnt_0[0] + cnt_1[0]$
    end
end
if dimension==2 then
    if $off_0[0] + cnt_0[0] == off_1[0]$ then
        if $off_0[1] == off_1[1]$ and $cnt_0[1] == cnt_1[1]$
          then
            $off_2[] = off_0[]$
            $cnt_2[1] = cnt_0[1]$
            $cnt_2[0] = cnt_0[0] + cnt_1[0]$
        end
    end
    if $off_0[1] + cnt_0[1] == off_1[1]$ then
        if $off_0[0] == off_1[0]$ and $cnt_0[0] == cnt_1[0]$
          then
            $off_2[] = off_0[]$
            $cnt_2[0] = cnt_0[0]$;
            $cnt_2[1] = cnt_0[1] + cnt_1[1]$
        end
    end
end
if dimension==3 then
    if $off_0[0] + cnt_0[0] == off_1[0]$ then
        if $off_0[1] == off_1[1]$ and $cnt_0[1] == cnt_1[1]$ and
          $cnt_0[2] == cnt_1[2]$ and $off_0[2] == off_1[2]$ then
            $off_2[] = off_0[]$
            $cnt_2[0] = cnt_0[0] + cnt_1[0]$
            $cnt_2[1] = cnt_0[1]$
            $cnt_2[2] = cnt_0[2]$
        end
    end
    if $off_0[1] + cnt_0[1] == off_1[1]$ then
        if $off_0[0] == off_1[0]$ and $cnt_0[0] == cnt_1[0]$ and
          $cnt_0[2] == cnt_1[2]$ and $off_0[2] == off_1[2]$ then
            $off_2[] = off_0[]$
            $cnt_2[0] = cnt_0[0]$;
            $cnt_2[1] = cnt_0[1] + cnt_1[1]$
            $cnt_2[2] = cnt_0[2]$;
        end
    end
    if $off_0[2] + cnt_0[2] == off_1[2]$ then
        if $off_0[1] == off_1[1]$and $cnt_0[0] == cnt_1[0]$ and
          $cnt_0[1] == cnt_1[1]$ and $off_0[0] == off_1[0]$ then
            $off_2[] = off_0[]$
            $cnt_2[2] = cnt_0[2] + cnt_1[2]$
            $cnt_2[0] = cnt_0[0]$
            $cnt_2[1] = cnt_0[1]$
        end
    end
end
```

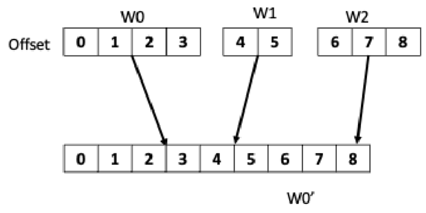
can be done within the HDF5 VOL layer by retrieving the dataspace selection information (offset and count arrays) from an HDF5 dataset write function call. For example, Fig. 1 (a) shows three 1D data writes $W0$, $W1$, and $W2$, with offset values 0, 4, 6 and count values 4, 2, and 3, respectively.

Algorithm 1 shows how we compare the offset ($off[]$), and count ($cnt[]$) values of two writes to detect and merge compatible requests. For 1D data, we check if the end offset of $W0$ is equal to the start offset of $W1$. If it is true, $W0$ and $W1$ are considered to be contiguous and can be merged into a larger one, $W0'$. $W0'$ has the same offset value as $W0$, but a larger count value equals the summation of $cnt_0[0]$ and $cnt_1[0]$. To reduce the memory overhead, we replace the data selection of $W0$ with the newly merged $W0'$ and remove the merged write request $W1$. The data buffer also needs to be merged by concatenating the buffers of the two write requests. We found that performing two *memcpy* operations per merge can take a significant amount of time, especially if many write operations can be merged and the total data size grows. We devised an optimization to extend the larger buffer with the new merge size using memory reallocation ($realloc$) and only perform one *memcpy* from the smaller buffer to the new buffer. For example, Fig.1 (a) shows three writes $W0$ (offset 0 and count 4), $W1$ (offset 4 and count 2), and $W2$ (offset 6 and count 3). Since the end offset of $W0$ and the start offset of $W1$ are the same (which is 4), then $W0$ and $W1$ are contiguous and can be merged. Furthermore, $W1$ and $W2$ also are contiguous because the end offset of $W1$ is the same as the start offset of $W2$ (which is 6). By performing two merges, the three write requests become one contiguous write, $W0'$. The offset of $W0'$ is copied from $W0$ (which is 0), and its count value is the summation of $W0, W1, W2$ (which is $4 + 2 + 3 = 9$). When merging $W0$ and $W1$, we extend $W0$'s buffer size to be $4+2 = 6$, then copy the 2 elements from $W1$ to the end of the extended buffer. Since $W2$ is also contiguous with $W0'$, with 3 elements, we can further extend the merged buffer to a size of $6 + 3 = 9$ and copy 3 data elements to the end of the buffer. Overall the merged write will have an offset of 0 and a count of 9. Data of write $W0$ is copied by the *memcpy* function with offset 0 and count 4.

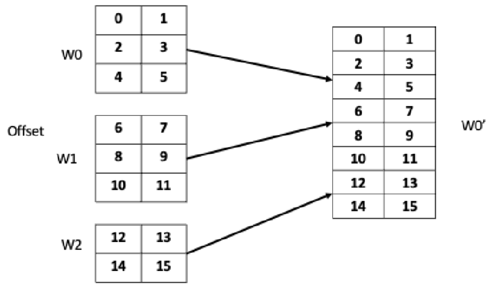
For 2D data, the contiguous data check depends on $off[]$ and $cnt[]$ values, each with 2 elements. In the beginning, we need to check the write operations $W0$ and $W1$ are contiguous in any of the 2 dimensions. If the end offset value of $W0$ is equal to the start offset value of $W1$ in any dimension, then that dimension is the merge dimension. We need to check further whether the offset and count values of another dimension are the same, if so, we can say the two writes are contiguous. For example, in Fig.1 (b) shows three writes $W0$ (offset $0, 0$ and count $3, 2$), $W1$ (offset $3, 0$ and count $3, 2$), and $W2$ (offset $6, 0$ and count $2, 2$). Since the end offset of $W0$ and the start offset of $W1$ are the same in the first dimension (which is 3), and the count and offset values of another dimension are identical (offset is 0 and the count is 2), then $W0$ and $W1$ are contiguous. Furthermore, $W1$ and $W2$ also are contiguous because the end offset of $W1$ is the same as the start offset of $W2$ (which is 6), and the offset and count values of the other dimension are also matched (the offset is 0 and the count is 2). The three write requests can then be merged into one, $W0'$. $W0'$ has offset values copied from $W0$ $(0, 0)$, and count values from $W0$ except in the merged direction, which is the summation of the count values $(3+3+2 = 8)$. Different from the 1D case, we can not always extend one buffer and copy the other buffer to the end, as the two buffers may be interleaved in the merged buffer unless the merge dimension is the last dimension.

For 3D data, the contiguous write requests are checked by comparing the $off[]$ and $cnt[]$ values, each with 3 elements. Similar to the 2D case, we check if $W0$ and $W1$ are contiguous in any of the three dimensions. We check each dimension by comparing the end offset $W0$ with $W1$'s start offset; if they match, we say that it is the merged dimension. After that, we check further two other dimensions. If both dimensions offset and count values are equal, we can consider them as contiguous data, and they can be merged. The newly merged write $W2$ will have the same offset values from $W0$, and the count of the merged dimension will be changed by the summation of the count value of $W0$ and $W1$, with other dimensions equal to $W0$. We perform the $memcpy$ operations to reconstruct the merged buffer by calculating the target locations of the data elements in each buffer. This can be calculated using $W2$ dimensions and offset and counts from $W0$. For example, Fig. 1(c) shows that $W0$ has a count of $3, 3, 3$, and the offset is $0, 0, 0$. The merged $W0'$ will copy the values to $0, 0, 0$. For the $W1$ *memcpy*, we need to calculate the new offset values using the offset of $W0$ $(0, 0, 0)$ and count $(3, 3, 3)$ values of $W0$, and it becomes $3, 0, 0$. If the merge happens in the last dimension, then we can extend the size of one buffer to the merged size by *realloc* function and copy the data from the other buffer to the extended space, which saves time.

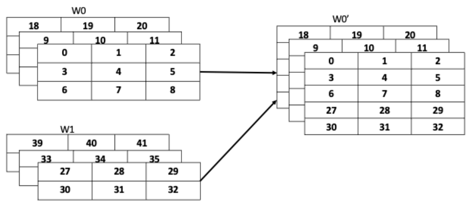
Fig. 2 shows how our proposed method works within the HDF5 asynchronous I/O VOL connector. Three write operations are in the asynchronous task queue named $W0$, $W1$, and $W2$. Our merge optimization checks the contiguousness of two write operations based on their offset and count values. When 2 write operations can be merged (e.g. $W0$ and $W1$), we replace $W0$ (or $W2$, depending on which one has a larger buffer size) with the newly merged data selection and concatenated buffer, and remove $W2$ from the Async Task Queue. We continue to check whether the newly merged $W0'$ can be merged with any other write request in the queue, e.g. $W2$. This process is repeated until no tasks in the queue can be merged. By performing a multi-pass of the data selection merge operations, we can merge multiple write requests even if they are out-of-order (e.g. the starting offsets of $W0$, $W1$, $W2$ are in non-increasing order). The time complexity of our algorithm is $O(N^2)$, where N is the number of write requests in the queue. For append-only applications, the time complexity is reduced to $O(N)$, as the offsets of write requests would be in increasing order, and a new write request can always be merged with the one existing task in the queue. We have tested the experiment up to 3D cases. Our algorithm can be extended to a higher number of dimensions, similar to the extension from 2D to 3D in Algorithm 1. In our proposed merge algorithm, we expect a time complexity of $O(N)$ to be the typical case, as scientific applications rarely perform out-of-order writes. Moreover, we provide the same consistency guarantee as the asynchronous I/O, as we do



(a) 1D data merge



(b) 2D data merge



(c) 3D data merge

Fig. 1: Example of different write request merges, where (a) shows three 1D writes ($W0$, $W1$, $W2$) merged into one ($W0'$), (b) shows three 2D writes ($W0$, $W1$, $W2$) merged into one ($W0'$), and (c) shows two 3D writes ($W0$, $W1$) merged into one ($W0'$) based on the offset values.

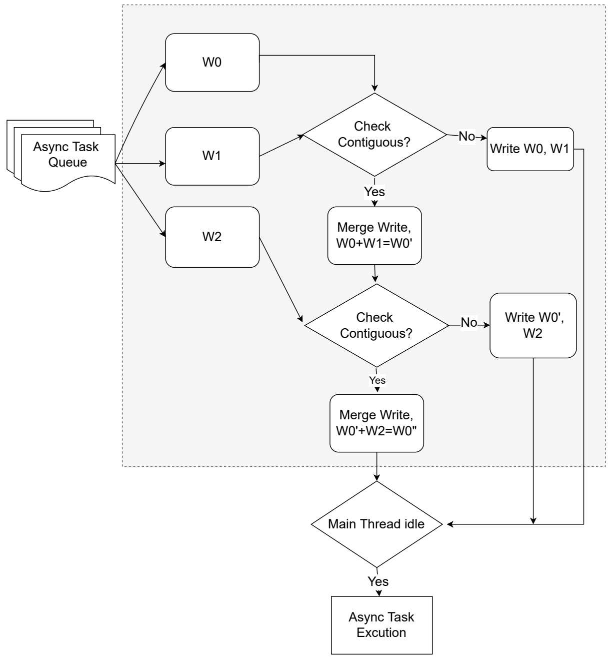


Fig. 2: Write merge optimization for asynchronous I/O. The components in the shaded area are the new additions we implemented on top of the HDF5 ASYNC I/O VOL connector.

not merge overlapping writes from the same process. In our experiments, we found that merging write operations are most effective when the original write sizes are less than 1MB.

V. Evaluation

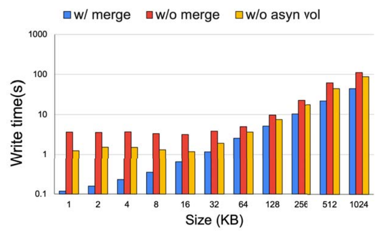
A. Experimental Setup

We have evaluated the performance of our proposed merge optimization for the asynchronous I/O VOL connector using synthetic benchmarks that mimic the I/O patterns from scientific applications that produce time-series data. The experiments are conducted on the Cori supercomputer at the National Energy Research Scientific Computing Center (NERSC). Cori is a Cray XC40 supercomputer with 1630 Intel Xeon "Haswell" nodes, each containing 32 cores and 128GB of memory. It also has a shared Lustre storage system with 248 OSTs. The default Lustre stripe size is 1MB and stripe count is 1.

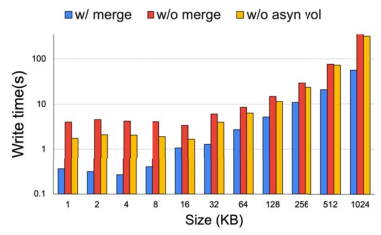
To demonstrate the effectiveness of our proposed optimization, we performed extensive benchmark evaluation with different data dimensions, write sizes, and the number of nodes/processes. We compared the performance of the merge-enabled asynchronous I/O VOL connector, vanilla asynchronous I/O without merge optimization, and synchronous HDF5 write. Since our goal is to measure the I/O performance improvement, we do not include any compute time between the writes, and thus the measured time for the two methods that use asynchronous I/O includes both the I/O time and the asynchronous I/O overhead, while the time for synchronous HDF5 is I/O time only. In our experiments, we set a time limit of 30 minutes per job, and some of the large-scale runs without using the merge optimization exceed this limit and are shown as bars with stripes. In our experiments, we have enough memory to merge all write requests into a single one. And the actual asynchronous write operation is triggered at file close time in our benchmark code.

B. Parallel I/O Performance

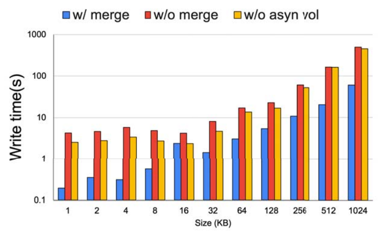
We compare the results using three workloads with 1D, 2D, and 3D data. For every dimension scenario, each process issues 1024 write contiguous requests with sizes ranging from $1KB$ to $1MB$. We performed these operations using 1 to 256 Cori Haswell nodes and 32 MPI ranks per node. The data from all processes are written to one HDF5 dataset. Figures 3, 4, and 5 show the comparison between merge-enabled Asynchronous VOL (w/ merge), vanilla Asynchronous VOL (w/o merge), and synchronous HDF5 write (w/o async), respectively. The



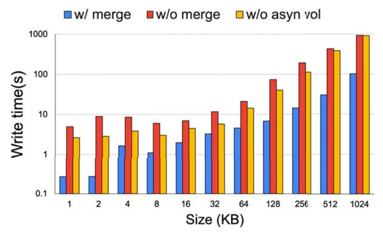
(a) 1 node



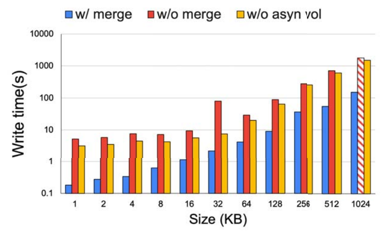
(b) 2 nodes



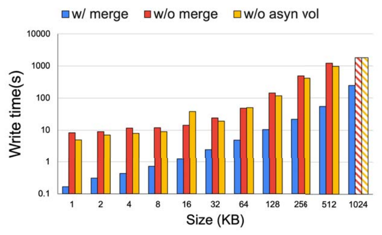
(c) 4 nodes



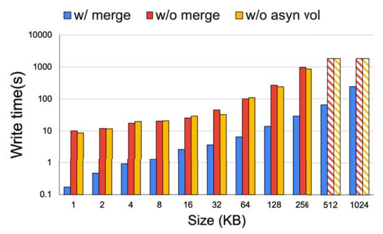
(d) 8 nodes



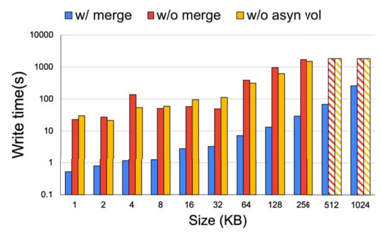
(e) 16 nodes



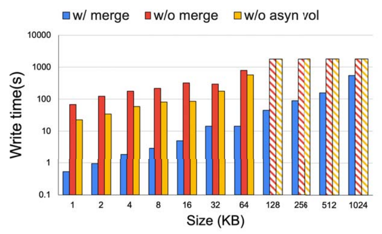
(f) 32 nodes



(g) 64 nodes



(h) 128 nodes



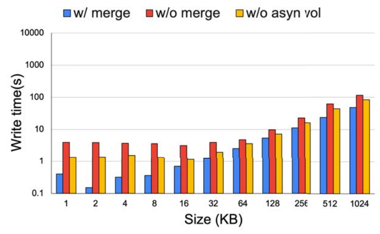
(i) 256 nodes

Fig. 3: Comparing write time in 1D datasets with different numbers of nodes, each with 32 ranks. "w/ merge" means using merge optimization with asynchronous I/O VOL, "w/o merge" means using asynchronous I/O VOL without merge optimization, and "w/o async vol" means synchronous HDF5 write. The bars with stripes are cases that exceed the 30-minute time limit.

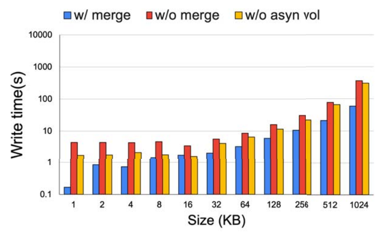
comparison shows that the merge-enabled asynchronous I/O offers significant performance improvement. In every case, the comparison figures show that our implementation provides better performance than the other two.

Figure 3 compares the results for all the 1D cases. When 1KB data is written at a time by each process, merge-enabled asynchronous I/O is $30\times$ faster than vanilla asynchronous I/O, and more than $10\times$ faster than synchronous HDF5 write. The vanilla asynchronous I/O is slower than the synchronous HDF5 because there is no computation to overlap the I/O time, and the asynchronous I/O overhead is comparable to the individual small-size write time. When running with the same number of nodes, the speed-up of the data writes by merge-enabled asynchronous I/O decreases as the data size increases. For 1MB data, merge-enabled asynchronous I/O is $2.5\times$ faster than vanilla asynchronous I/O whereas about $2\times$ faster than synchronous HDF5. As mentioned previously, we fixed the time limit to a maximum of 30 minutes. If any job takes more than 30 minutes, we plot that case as a 30-minute stripe bar. From 32 to 256 nodes, for 1MB data, both vanilla asynchronous I/O and synchronous HDF5 writes exceed the time limit, whereas merge-enabled asynchronous I/O took less than 10 minutes. When the write size per node is fixed, the speed-up increases as we increase the number of nodes. For 256 nodes, merge-enabled asynchronous I/O shows about $130\times$ improvement in write operations than vanilla merge-enabled asynchronous I/O for 1KB and 2KB data. However, for 32KB data, merge-enabled asynchronous I/O improves $20\times$ more than vanilla asynchronous I/O and $12\times$ more than synchronous HDF5 write.

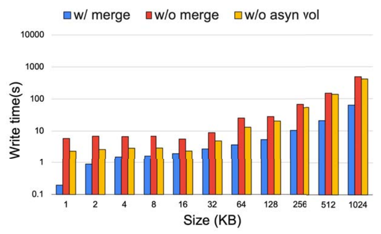
In 2D data writes shown in Figure 4, when each process writes 2KB data at a time, the merge-enabled asynchronous I/O is $25\times$ faster than vanilla asynchronous I/O and more than $9\times$ faster than synchronous HDF5. For 1MB data, merge-enabled asynchronous I/O shows the best performance with 16 nodes, which is $11\times$ faster than vanilla asynchronous I/O and about $9\times$ faster than synchronous HDF5 write. Similarly to 1D data, we have a job time limit of 30 minutes. With 1MB data writes between 32 and 256 nodes, both vanilla asynchronous I/O and synchronous HDF5 exceed the time limit, shown as striped bars in Figure 4. The merge-enabled asynchronous I/O took less than 9 minutes for cases. Besides, for 256 nodes, the merge-enabled asynchronous I/Os VOL shows about $55\times$ improvement in write operations than vanilla merge-enabled



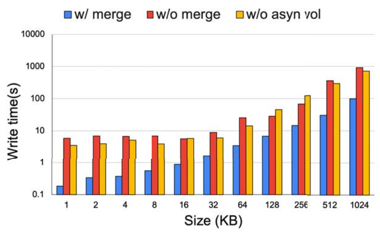
(a) 1 node



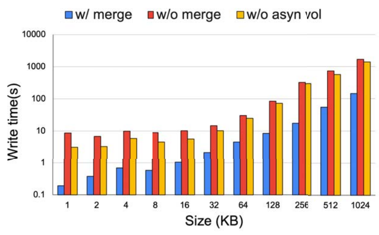
(b) 2 nodes



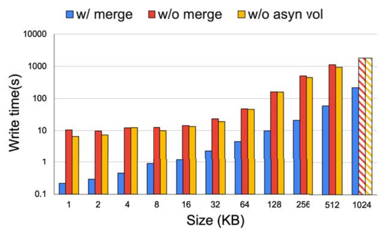
(c) 4 nodes



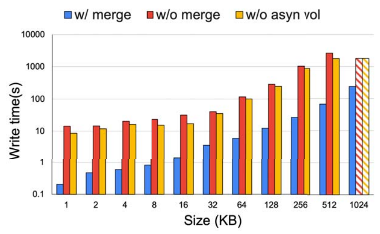
(d) 8 nodes



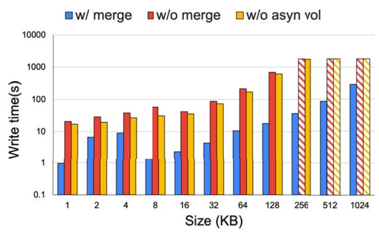
(e) 16 nodes



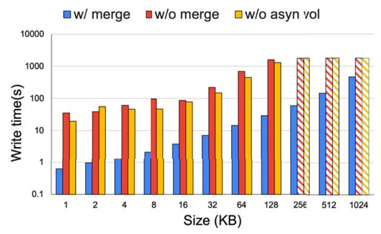
(f) 32 nodes



(g) 64 nodes



(h) 128 nodes



(i) 256 nodes

Fig. 4: Comparing write time in 2D datasets with different numbers of nodes, each with 32 ranks. "w/ merge" means using merge optimization with asynchronous I/O VOL, "w/o merge" means using asynchronous I/O VOL without merge optimization, and "w/o async vol" means synchronous HDF5 write. The bars with stripes are cases that exceed the 30-minute time limit.

asynchronous I/O for 1KB data. Additionally, with 256 nodes and 128KB data, merge-enabled asynchronous I/O improves $54\times$ more than vanilla asynchronous I/O and $44\times$ more than synchronous HDF5 write.

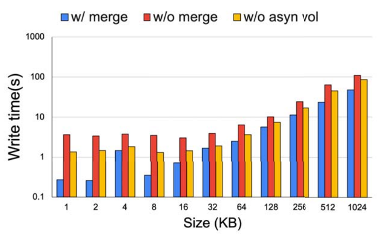
Figure 5 shows the results for 3D data writes. With 128 nodes and 4096 MPI ranks, each writes 1KB data, the merge-enabled asynchronous I/O is about $70\times$ faster than vanilla asynchronous I/O, and more than $33\times$ faster than synchronous HDF5 write. For 2KB data write, with 256 nodes, the best performance with merge-enabled asynchronous I/O is $100\times$ faster than vanilla asynchronous I/O. For 256KB data, the merge-enabled asynchronous I/O shows the best performance with 16 nodes, which is $25\times$ faster than vanilla asynchronous I/O and $18\times$ faster than synchronous HDF5 write. Like 1D and 2D data, in 1MB data writes between 16 and 256 nodes, both vanilla asynchronous I/O and synchronous HDF5 writes exceed the 30 minute time limit, which is shown as striped bars in Figure 5. In contrast, merge-enabled asynchronous I/O takes less than 8 minutes for all cases.

Overall, our experiments show that the merge-enabled asynchronous I/O offers significant performance in all cases compared to the vanilla asynchronous I/O and the synchronous I/O. Higher speedup (up to $130\times$) is observed when the write size is small (a few KB). When there is a large number of writers (more than 32 nodes, 1024 MPI ranks), both the vanilla asynchronous I/O and the synchronous I/O take more than 30 minutes. In contrast, the merge-enabled asynchronous I/O finishes in less than 10 minutes.

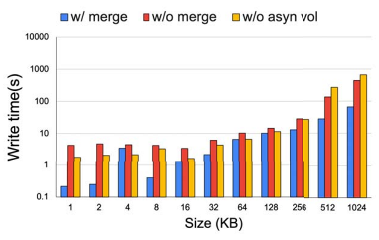
VI. Conclusion and Future Work

Our findings indicate that merging small write operations can effectively reduce the I/O time. We have developed an optimization strategy to merge write requests in the HDF5 asynchronous I/O VOL connector. We demonstrate the effectiveness of our solution by comparing the I/O performance among the merged-enabled asynchronous I/O with vanilla asynchronous I/O and the synchronous HDF5 I/O. Our results can achieve up to $130\times$ speedup compared with vanilla asynchronous I/O.

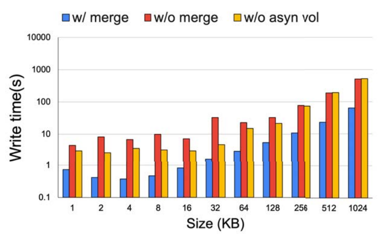
Our future work includes evaluating the merged-enabled asynchronous I/O with more benchmark workloads and real scientific applications, as well as extending it to other asynchronous I/O libraries.We will explore methods to perform merge operations under different consistency models and op-



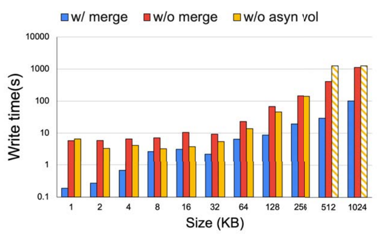
(a) 1 node



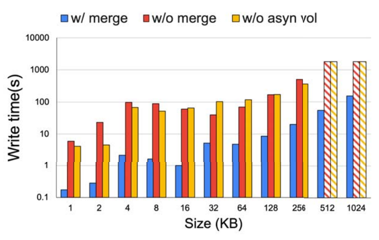
(b) 2 nodes



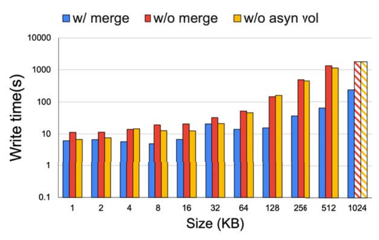
(c) 4 nodes



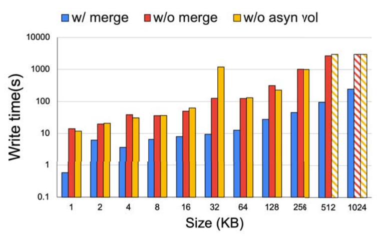
(d) 8 nodes



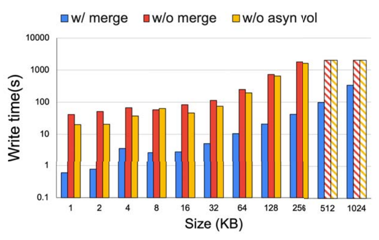
(e) 16 nodes



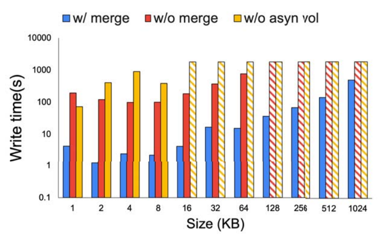
(f) 32 nodes



(g) 64 nodes



(h) 128 nodes



(i) 256 nodes

Fig. 5: Comparing write time in 3D datasets with different numbers of nodes, each with 32 ranks. "w/ merge" means using merge optimization with asynchronous I/O VOL, "w/o merge" means using asynchronous I/O VOL without merge optimization, and "w/o async vol" means synchronous HDF5 write. The bars with stripes are cases that exceed the 30-minute time limit.

timize the algorithm to reduce the time complexity for the worst case.

Moreover, multiple clients writing over the same data in distributed systems can also lead to consistency problems which can be addressed in the future work.

Acknowledgment

This research was supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration. This research was partially supported by The Ohio State University under a subcontract (GR130444), which was supported by the U.S. Department of Energy (DOE), Office of Science, Office of Advanced Scientific Computing Research (ASCR) under contract number DE-AC02-05CH11231 with LBNL. This research was also partially support by the University of Alabama and the National Science Foundation through grant number EF-2150107. This research used resources of the National Energy Research Scientific Computing Center, a DOE Office of Science User Facility supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231.

References

[1] S. Byna, J. Chou, O. Rubel, H. Karimabadi, W. S. Daughter, V. Roytershteyn, E. W. Bethel, M. Howison, K.-J. Hsu, K.-W. Lin *et al.*, "Parallel I/O, analysis, and visualization of a trillion particle simulation," in *SC*, 2012.

[2] D. McCallen, A. Petersson, A. Rodgers, A. Pitarka, M. Miah, F. Petrone, B. Sjogreen, N. Abrahamson, and H. Tang, "Eqsim—a multidisciplinary framework for fault-to-structure earthquake simulations on exascale computers part i: Computational models and workflow," *Earthquake Spectra*, vol. 37, no. 2, pp. 707–735, 2021.

[3] D. McCallen, F. Petrone, M. Miah, A. Pitarka, A. Rodgers, and N. Abrahamson, "Eqsim—a multidisciplinary framework for fault-to-structure earthquake simulations on exascale computers, part ii: Regional simulations of building response," *Earthquake Spectra*, vol. 37, no. 2, pp. 736–761, 2021.

[4] H. Tang, S. Byna, N. A. Petersson, and D. McCallen, "Tuning parallel data compression and i/o for large-scale earthquake simulation," in *2021 ieee international conference on big data (big data)*. IEEE, 2021, pp. 2992–2997.

[5] A. S. Almgren, J. B. Bell, M. J. Lijewski, Z. Lukić, and E. Van Andel, "Nyx: A massively parallel amr code for computational cosmology," *The Astrophysical Journal*, vol. 765, no. 1, p. 39, 2013.

[6] H. Tang, S. Byna, F. Tessier, T. Wang, B. Dong, J. Mu, Q. Koziol, J. Soumagne, V. Vishwanath, J. Liu *et al.*, "Toward scalable and asynchronous object-centric data management for hpc," in *2018 18th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*. IEEE, 2018, pp. 113–122.

[7] G. Lee, S. Shin, W. Song, T. J. Ham, J. W. Lee, and J. Jeong, "Asynchronous i/o stack: A low-latency kernel i/o stack for ultra-low latency ssds." in *USENIX Annual Technical Conference*, 2019, pp. 603–616.

[8] H. Tang, Q. Koziol, S. Byna, J. Mainzer, and T. Li, "Enabling transparent asynchronous i/o using background threads," in *2019 IEEE/ACM Fourth International Parallel Data Systems Workshop (PDSW)*. IEEE, 2019, pp. 11–19.

[9] H. Tang, Q. Koziol, J. Ravi, and S. Byna, "Transparent asynchronous parallel i/o using background threads," *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 4, pp. 891–902, 2021.

[10] R. Jain, H. Tang, A. Dhruv, J. A. Harris, and S. Byna, "Accelerating flash-x simulations with asynchronous i/o," in *2022 IEEE/ACM International Parallel Data Systems Workshop (PDSW)*. IEEE, 2022, pp. 13–19.

[11] Y. J. Yu, D. I. Shin, W. Shin, N. Y. Song, J. W. Choi, H. S. Kim, H. Eom, and H. Y. Yeom, "Optimizing the block i/o subsystem for fast storage devices," *ACM Transactions on Computer Systems (TOCS)*, vol. 32, no. 2, pp. 1–48, 2014.

[12] Z. Zhu, L. Tan, Y. Li, and C. Ji, "Phdfs: Optimizing i/o performance of hdfs in deep learning cloud computing platform," *Journal of Systems Architecture*, vol. 109, p. 101810, 2020.

[13] J. Lee, Y. Kim, G. M. Shipman, S. Oral, and J. Kim, "Preemptible i/o scheduling of garbage collection for solid state drives," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 32, no. 2, pp. 247–260, 2013.

[14] H. Lin, X. Ma, W. Feng, and N. F. Samatova, "Coordinating computation and i/o in massively parallel sequence search," *IEEE Transactions on Parallel and Distributed Systems*, vol. 22, no. 4, pp. 529–543, 2010.

[15] C. Wu, C. Ji, L. Shi, C. J. Xue, B. Huang, and Y. Wang, "Dynamic merging/splitting for better responsiveness in mobile devices," in *2016 5th Non-Volatile Memory Systems and Applications Symposium (NVMSA)*. IEEE, 2016, pp. 1–6.

[16] Z. Chen and K. Shen, "{OrderMergeDedup}: Efficient{Failure-Consistent} deduplication on flash," in *14th USENIX Conference on File and Storage Technologies (FAST 16)*, 2016, pp. 291–299.

[17] K. Ohta, D. Kimpe, J. Cope, K. Iskra, R. Ross, and Y. Ishikawa, "Optimization techniques at the I/O forwarding layer," in *International Conference on Cluster Computing*. IEEE, 2010, pp. 312–321.

[18] B. C. An and H. Sung, "Efficient i/o merging scheme for distributed file systems," *Symmetry*, vol. 15, no. 2, p. 423, 2023.

[19] "The hdf group. (1997-) hierarchical data format, version 5," https://www.hdfgroup.org/solutions/hdf5/.

[20] S. Byna, J. Chou, O. Rubel, Prabhat, H. Karimabadi, W. S. Daughter, V. Roytershteyn, E. W. Bethel, M. Howison, K.-J. Hsu, K.-W. Lin, A. Shoshani, A. Uselton, and K. Wu, "Parallel i/o, analysis, and visualization of a trillion particle simulation," in *SC '12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, 2012, pp. 1–12.

[21] S. Byna, M. S. Breitenfeld, B. Dong, Q. Koziol, E. Pourmal, D. Robinson, J. Soumagne, H. Tang, V. Vishwanath, and R. Warren, "Exahdf5: Delivering efficient parallel i/o on exascale computing systems," *Journal of Computer Science and Technology*, vol. 35, no. 1, 1 2020.

[22] M. M. A. Patwary, S. Byna, N. R. Satish, N. Sundaram, Z. Lukić, V. Roytershteyn, M. J. Anderson, Y. Yao, Prabhat, and P. Dubey, "Bd-cats: big data clustering at trillion particle scale," in *SC '15: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2015, pp. 1–12.

[23] "The hdf group. (2015-) hdf5 virtual object layer (vol) documentation," https://portal.hdfgroup.org/display/HDF5/HDF5+VOL+User's+Guide.