Enabling Transparent Asynchronous I/O using Background Threads

Houjun Tang*, Quincey Koziol*, Suren Byna*, John Mainzer†, Tonglin Li*
*Lawrence Berkeley National Laboratory, Berkeley, CA
†The HDF Group, Champaign, IL
*{htang4, koziol, sbyna, tonglinli}@lbl.gov, †mainzer@hdfgroup.org

***Abstract*—With scientific applications moving toward exascale levels, an increasing amount of data is being produced and analyzed. Providing efficient data access is crucial to the productivity of the scientific discovery process. Compared to improvements in CPU and network speeds, I/O performance lags far behind, such that moving data across the storage hierarchy can take longer than data generation or analysis. To alleviate this I/O bottleneck, asynchronous read and write operations have been provided by the POSIX and MPI-I/O interfaces and can overlap I/O operations with computation, and thus hide I/O latency. However, these standards lack support for non-data operations such as file open, stat, and close, and their read and write operations require users to both manually manage data dependencies and use low-level byte offsets. This requires significant effort and expertise for applications to utilize.**

To overcome these issues, we present an asynchronous I/O framework that provides support for *all* I/O operations and manages data dependencies transparently and automatically. Our prototype asynchronous I/O implementation as an HDF5 VOL connector demonstrates the effectiveness of hiding the I/O cost from the application with low overhead and easy-to-use programming interface.

I. Introduction

Exascale high performance computing (HPC) systems are arriving soon, and will produce an unprecedented amount of data that presents new I/O management and performance challenges. However, the I/O sub-system of supercomputers has not kept up with the trend of CPU and network speed improvements, and applications are likely to suffer from poor I/O performance, significantly impacting scientific productivity.

To help mitigate this problem, some I/O systems offer an asynchronous interface that allows applications to schedule an I/O operation and then proceed with computation, testing/waiting on the I/O request when desired, instead of blocking until its completion. Many applications can take advantage of an asynchronous interface by scheduling I/O as early as possible and coming back to check on the status of the I/O operations when needed. This allows an overlap of communication and computation with I/O operations, hiding some or all of the cost associated with I/O.

The POSIX I/O [1] standard provides "aio_*" routines for reading and writing data asynchronously to the file system. MPI-I/O [2] provides a non-blocking interface to read and write data from files with MPI parallel applications. However, these two solutions only support basic read and write operations, leaving others such as file open and close to remain synchronous. The asynchronous operations provided are also at a lower level, operating on byte offsets and counts, which requires more effort from an application developer. High-level I/O libraries and data management systems, such as ADIOS [3] and PDC [4], offer asynchronous data movement to and from their server nodes through network data transfer. While they are effective in moving the data to different locations, managing separate servers require extra resources and effort.

In addition to these issues, additional challenges must be overcome for the effective implementation of asynchronous I/O, particularly in an HPC environment. Managing data dependencies and retaining the correct execution order is crucial to data consistency. For example, an asynchronous file read can only be executed after a successful asynchronous file open, and an asynchronous read issued after an asynchronous write of the same data must be performed in its original issue order. Asynchronous collective I/O operations must also be correctly supported, so that they execute without causing a deadlock. To reduce application impact, asynchronous tasks should only start execution when the application's main thread is no longer issuing I/O operations and has moved into a compute phase. Additionally, the user should not be burdened with manual management of all these operations and should have a low-effort mechanism to take advantage of the asynchronous I/O operations and monitor their completion.

To tackle these challenges, we propose an asynchronous I/O framework that *supports all types of I/O operations*, including both collective and independent ones, *requires no additional servers, manage data dependencies transparently and automatically from users, and requires minimal code modifications*.

Implementation of asynchronous I/O operations can be achieved in different ways. Since the native asynchronous interface offered by most existing operating systems and low-level I/O frameworks (POSIX AIO and MPI-IO) does not include all file operations, we chose to perform I/O operations in a background thread. With recent increases in the number of available CPU threads per processor, it is now possible to utilize a thread to execute asynchronous operations from the core that the application is running on without significant impact to the application's performance. To manage data

978-1-7281-6005-4/19/$31.00 ©2019 IEEE
DOI 10.1109/PDSW49588.2019.00006

dependencies and support collective operations, we maintain a queue of different types of tasks and tracks their dependencies, such that only when an asynchronous task's dependent parents are completed will it be scheduled for execution in the background thread. We also manage both the collective and independent operations such that the collective operations are executed in the same order and avoid any mismatched deadlock situation. To remain low overhead and avoid the contention of shared resources between the application's main thread and the background thread, we used a status detection mechanism to check when the main thread is performing non-I/O tasks.

In summary, our proposed method makes the following contributions:

- We adopt a background thread approach that accumulates I/O tasks and starts their execution when we detect that the application's main thread is idle or performing non-I/O operations.
- We propose a task dependency management protocol to guarantee the data consistency and support both collective and independent operations.
- We provide asynchronous I/O support to applications transparently, in a way that requires no more than a few lines of code changes.
- We show the evaluation of our implementation that adds support to the HDF5 library [5] with several benchmarks and I/O kernels that demonstrate the effectiveness of the asynchronous I/O with low overhead.

We have evaluated the proposed asynchronous I/O framework on the Cori supercomputer at the National Energy Research Scientific Computing Center (NERSC) with several benchmarks and I/O kernels. Experimental results show that our method can effectively mask the I/O cost when the application is idle or performing non-I/O operations. The remainder of the paper is organized as follows: We introduce our asynchronous I/O framework's design in Section II, and the implementation as an HDF5 VOL connector in Section III. In Section IV, we describe our experimental setup and present the results in Section V. We discuss the relevant literature in Section VI and conclude the paper in Section VII.

II. Asynchronous I/O

To enable asynchronous I/O for applications, the least intrusive implementation is to make all the I/O operations *implicitly* asynchronous, such that those operations return immediately once issued. In this way, the user can still use their existing code, without managing the asynchronous requests or explicitly waiting for operations to complete. To enable such mode of asynchronous operations, one can invoke the asynchronous framework's initialization routine before file create or open within the application or set an environment variable before running the application. The I/O operations will then be executed in a separate background thread that is fully managed by the asynchronous framework without the need of user intervention, and avoids the requirement of creating extra server nodes that move data to their memory or node-local storage.

Executing asynchronously in the correct order is possible by creating a graph representation of the dependencies between operations, internal to the asynchronous I/O framework. As a result, operations that have a dependency will be paused until their dependencies are fulfilled and then will be scheduled to run when resources are available. This approach allows an existing application to execute all of its I/O operations asynchronously in the background without blocking the application's main process/thread's progress, while still being confident that they will be issued in the correct order. Such an approach is particularly useful for checkpoint operations, as the I/O time can be effectively masked by the compute time between checkpoints.

Figure 1 illustrates the workflow of our proposed asynchronous I/O framework. We will describe details on our design in the following sub-sections.

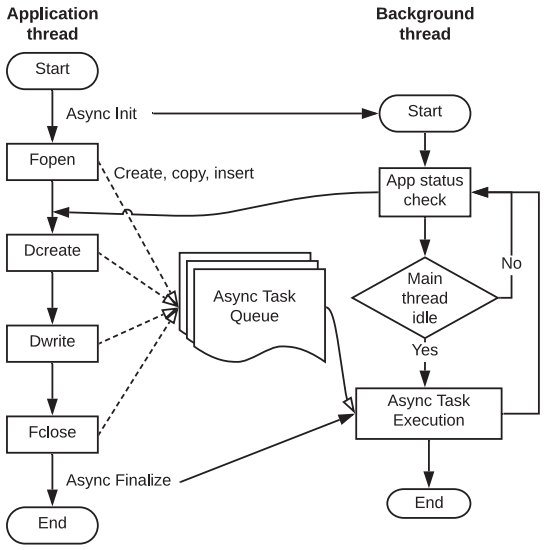

Fig. 1. **A workflow overview of the asynchronous I/O framework.** When an application enables asynchronous I/O, a background thread is started. Each I/O operation is intercepted and an asynchronous task is created, storing all the relevant information before inserting it into the asynchronous task queue. The background thread monitors the running state of the application, and only starts executing the accumulated tasks when it detects the application is idle or performing non-I/O operations. When all I/O operations have completed and the application issues the file close call, the asynchronous I/O related resources, as well as the background thread itself, would be freed.

A. Asynchronous Task

When the asynchronous I/O framework is enabled, an asynchronous task object is created for each I/O operation, storing all the information needed for that operation. It includes a copy of all parameters, a function pointer to the operation to execute, data pointers, and internal states such as its dependency and execution status. After the creation, the task is appended to a queue (more details explained in the next sub-section) that the background thread's execution engine can access. These asynchronous tasks are internal data that are

managed only by the asynchronous I/O framework, and are not exposed to users. Once the task is created, the corresponding function returns to the application without blocking it for the I/O operation to complete.

When the asynchronous execution engine determines that a task can be executed, the task's information is passed to the background thread, which is then executed atomically and run to completion before the next task starts running. A task can be canceled while it is queued, but it is rarely needed.

To allow applications to reuse a buffer immediately after a data write call (i.e., H5Dwrite in HDF5) and prevent data consistency issues while the task is awaiting execution, we make a copy of the data in the user's buffer at task creation time by default. However, we also provide the option to not duplicate the data if the application is memory-sensitive (see Section III).

The asynchronous I/O framework requires less than 1KB of extra memory to store each asynchronous task, plus a copy of the application buffer, for data write calls. The tasks are freed once the corresponding objects are no longer used, e.g. an object is explicitly closed by the application. As a result, the framework typically requires no more than a few MBs of extra memory at run time.

B. Task Dependency Management

Data dependencies are common for scientific application I/O operations. For example, all data read and write operations depend on the successful completion of the corresponding file create or open operation. Reads and writes on the same object should be executed in an order that guarantees the data access consistency. Additionally, collective operations must be handled properly to avoid mismatched operations being executed that lead to a deadlock. With these considerations in mind, we use the following set of rules:

1) All I/O operations can only be executed after a successful file create/open.
2) The file close operation can only be executed after all previous operations in the file have been completed.
3) Any read/write operation must be executed after a prior write operation to the same object.
4) Any write operation must be executed after a prior read operation to the same object.
5) Any collective operation must be executed in the same order with regard to other collective operations.
6) There can only be 1 collective operation in execution at any time (among all the threads on a process).

To manage the tasks with the above rules in mind, we maintain a queue of task lists that are categorized into three types: **Regular Task List (RTL)** - tasks that have no dependencies between them and can be executed independently; **Collective Task List (CTL)** - tasks that are collective operations; **Dependent Task List (DTL)** - tasks that depend on one or more tasks. We implemented the queue as a linked list, as shown in Figure 2. Note that a collective task may or may not have dependent parents, and is treated as a special dependent parent when it does.

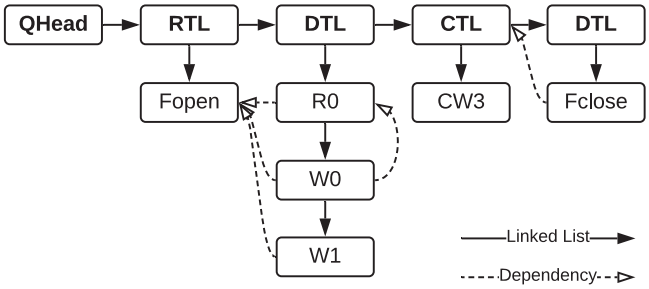

Fig. 2. Example of dependency management, where RTL: regular task list, CTL: collective task list, DTL: dependent task list, R0: read object 0, W0: write object 0, W1: write object 1, CW3: collective write object 3. R0 depends on Fopen, W0 depends on both Fopen and R0, W1 depends on Fopen, Fclose depends on all previous operations.

There can be multiple task lists of each type in the task queue. Any time a non-regular task is created, a new task list will be created. To create an asynchronous task, we first check if it is a collective operation. If it is, either a new CTL is created, or the task is appended to an existing CTL that is the tail in the queue. We then check for object dependency, for example, a group open depends on the corresponding file open/create operation, according to the following rules:

1) Any read/write operation depends on a prior write operation of the same object.
2) Any write operation depends on a prior read operation of the same object.
3) Any object close operation depends on all previous operations of the same object.

If satisfies either of the above rules, create a new DTL and insert to it. Otherwise, create or insert it into the current tail RTL. If the current RTL is also the head of the queue, mark it as ready for execution and send the first operation to the scheduler. Any time an asynchronous task has completed, we dequeue the head of the task list and schedule it for execution next. In this way, the asynchronous tasks are chained and can be executed safely and reliably.

Note that with the above rules, it is possible that some asynchronous tasks are not executed in the same order as in the application's code. Specifically, there are two cases that out-of-order execution could happen: 1) non-collective reads on the same or different objects, and 2) Non-collective writes on different objects. These two exceptions would not cause data consistency issues and thus are allowed in our framework.

C. Background Thread Execution

In the past, it was not always possible to "steal" a thread to do asynchronous I/O from the application resources of available threads. However, with recent increases in the number of available CPU threads per processor, this is no longer the case. Additionally, recent HPC system architectures and early designs for some future exascale systems employ forms of I/O forwarding, where compute nodes do not have local disks that store data, but use a fast network interconnect to forward I/O requests to dedicated servers that handle them.

I/O forwarding could be done asynchronously from the applications running on compute nodes by using a background thread. On these architectures, the file system needs not even expose an asynchronous interface to implement asynchronous I/O since the data is moved through the network to a separate I/O server. Furthermore, a background thread on the compute node to transfer the data might not even be needed if the network hardware provides a native asynchronous RDMA interface.

We use Argobots [6] as the background thread execution engine due to its low overhead. Argobots is a lightweight, low-level threading and tasking framework. It directly leverages the lowest-level constructs in the hardware and OS: lightweight notification mechanisms, data movement engines, memory mapping, and data placement strategies. However, our implementation is not tied to Argobots and it could be replaced by other thread management libraries, such as future versions of OpenMP [7]. By default, we use one background thread for executing I/O tasks separate from the applications main thread, which provides the asynchronous I/O benefits and limits the performance impact to the multi-threaded applications.

D. Main Thread Status Detection

Without special intervention, task execution could happen concurrently in the background thread and the application's main thread, and it is possible that the two threads would compete for access to shared resources, such as the modifying the asynchronous task queue. It is also common for a scientific application to perform a number of I/O operations before or after a period of computation. Thus it is desirable for the background thread to know the status of the application's main thread, such that it does not start asynchronous task execution immediately after a new task is created by the main thread, but rather waits until the main thread finishes queuing all its I/O operations and moves on to other operations. In this way, the two threads avoid competition for shared resources that could lead to an effectively synchronous execution.

When the application issues a number of asynchronous I/O operations, the time gap between consecutive ones are very small, in the microseconds level. Thus by maintaining a counter to track the number of asynchronous I/O tasks created by the application, and monitoring the value of the counter, we can determine if the application is currently busy issuing I/O requests or not. The asynchronous execution engine checks the counter value twice with a sleep time in between (by default it is set 100 microseconds). If the counter value does not increase between the two checks, then the main thread is considered to have finished queuing I/O operations, and the background thread can start the execution of I/O tasks. Otherwise, the background thread waits and repeats the procedure. This mandatory sleep time can become a significant overhead when executing a large number of I/O operations that take comparable time to the wait time, such as metadata access that may only take a few microseconds. To reduce such overhead, we do not perform the counter status check every time, but check every few operations. We also vary the frequency between fast I/O (i.e., metadata operations, frequency = 8) and the slow I/O (i.e., raw data operations, frequency = 2) to achieve the best performance.

III. Implementation as an HDF5 VOL connector

HDF5 [5] is a popular high level I/O library that has been used in a wide variety of scientific domains. Currently, HDF5 does not support asynchronous I/O, adding asynchronous I/O support to the HDF5 library would benefit a large number of existing applications. HDF5 provides the Virtual Object Layer (VOL) [8], which intercepts all HDF5 API calls that could potentially access objects in the file and forwards those calls to a VOL connector that accesses the objects. The user still gets the same data model where access is done to a single HDF5 "container"; however the VOL connector translates from what the user sees to how the data is actually stored. The HDF5 VOL is an ideal place for us to implement our asynchronous I/O framework and enables existing applications that already use HDF5 to take advantage of this feature with minimal code changes. Our implementation can be compiled as a dynamic-link library, and can be linked to the user's application directly, remaining separate from the installed version of HDF5.

However, adding asynchronous operations to the HDF5 library is not an easy task, because of the large number and variety of HDF5 operation types. HDF5 operations can be divided into three categories:

1) Metadata operations: Operations like creating files and objects (groups, datasets, etc), managing the HDF5 group hierarchy, creating and operating on attributes, etc. This further breaks down into several categories:
   a) Initiation operations – Create and open objects, etc.
   b) Modification operations – Extend dataset dimensions, write an attribute, flush the file, etc.
   c) Query operations – Get the number of links in a group, get the datatype or dataspace for a dataset, read an attribute, etc.
   d) Close operations – Close an object or file.
2) Raw Data operations: Reading and writing HDF5 datasets (i.e., `H5Dread` and `H5Dwrite`).
3) HDF5 local operations: Those are operations that do not access the HDF5 file but just aid users in accessing the files (creating and managing property lists, IDs, dataspaces, etc.). These types of operations are memory-only operations and can not perform any I/O, so they do not need to be performed asynchronously.

Implicitly executing operations asynchronously requires that we make `H5Fclose` and metadata query operations blocking. `H5Fclose` waits for all the previous asynchronous tasks to complete before returning, so that the application won't exit until all tasks have been completed. Metadata query operations return information that the application may use immediately after the call, and thus must be executed successfully before proceeding to the next operation. All other HDF5 I/O functions are non-blocking by default.

1) Additional Functions: To allow users to explicitly enable asynchronous I/O operations and check their status, we provide a few APIs in the HDF5 asynchronous VOL connector: `H5Pset_vol_async`, `H5Pset_dxpl_async_cp_limit`, `H5Dtest`, `H5Dwait`, `H5Ftest`, and `H5Fwait`. `H5Pset_async_vol` sets the HDF5 file access property list to enable asynchronous I/O when it is used at file open or creation time, this call can also be replaced by setting an environment variable (`HDF5_VOL_CONNECTOR`) before launching the application. `H5Pset_dxpl_async_cp_limit` controls the size limit when making a copy of the user's data buffer, setting it to 0 would disable the duplication and it would become the application's responsibility to ensure the validity of the data before it is written by the background thread. `H5Dtest` and `H5Ftest` return the status of the asynchronous task for a dataset or all operations in a file. `H5Dwait` and `H5Fwait` block the application's main thread until the corresponding asynchronous dataset or file operations have completed in the background thread. All of these functions are optional and the user could simply set an environment variable before running their program and take advantage of the asynchronous I/O.

2) Error Reporting: One downside of our proposed approach is its lack of immediate feedback for errors that occur when implicitly executed asynchronous operations fail, which could cause the remaining operations to fail subsequently. Thus effective error reporting becomes important to help users locate the root cause of the errors when they occur. In our implementation, we log and output the asynchronous tasks' errors, and prevent the remaining tasks that depend on the successful execution of the failed task from getting executed. Using this approach can greatly reduce the number of error messages and makes it easier for the user to trace the root cause of failures.

IV. Experimental Setup

We have evaluated the performance of our proposed asynchronous I/O framework with benchmarks and I/O kernels from scientific applications. We ran experiments 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, where each node consists of 32 cores and 128GB memory. Its Lustre storage system has 248 OSTs and is shared by all users. We configured the asynchronous I/O framework to use 1 background thread.

To measure the performance, we have created micro-benchmarks that simulate the I/O behavior of a serial or parallel application, which can write or read time-series data. The data of each timestep is stored in one group of an HDF5 file. Each group has a number of datasets (raw data) as well as attributes (metadata). The micro-benchmark also includes a configurable "computation" time duration between writes or reads, which mimics the behavior of computation between timesteps in scientific applications.

In addition to the micro-benchmark, we used two I/O kernels, VPIC-IO and BD-CATS-IO, to evaluate the parallel performance of our proposed asynchronous I/O framework.

1) VPIC-IO: The VPIC-IO kernel[1] is extracted from a plasma physics code called VPIC [9], which simulates kinetic plasma in a multi-dimensional space. In this kernel, each MPI process writes 8M (8×2^{20}) particles with eight properties, including x, y, z, Ux, Uy, Uz, id1, and id2, with a total size of $256MB$ data for each process. With a fixed amount of data written by each process, VPIC-IO is a weak scaling test. Only a few lines of code are modified to enable the asynchronous I/O. When there is sufficient computation time, only the last timestep's write time plus the asynchronous I/O framework's overhead are observed by the application.

2) BD-CATS-IO: The BD-CATS-IO kernel[2] is extracted from a parallel clustering algorithm code [10], used for analyzing the data produced by particle simulations, such as VPIC. In this kernel, the I/O patterns exactly match that of the simulation and analysis, such that data related to the particles are read among all the MPI processes with an even distribution. Similar to the VPIC-IO, BD-CATS-IO is also a weak scaling test. We have modified the BD-CATS-IO code to prefetch the next timestep's data before processing the current timestep, allowing the background thread to overlap I/O with the application's computation. In this way, only the first timestep's read time and the overhead are observed by the application.

For all the results presented in the following section, we have measured the elapsed I/O time observed by the application, which is the time from the first I/O operation until the last I/O operation finishes, and excludes the computation time between the I/O operations if there are any. We ran each experiment at least **10** times and report the time of the best results. The variance of different runs for the same configuration is less than 10% for the majority of cases. For serial tests, we set the Lustre stripe count to 4 and stripe size to 4MB, and for parallel experiments, the stripe count is set to 128 and stripe size is 32MB. To avoid any cache effect, we configured each run to operate on a separate file.

V. Evaluation

A. Serial I/O Performance

To verify the correctness and evaluate the overhead of our asynchronous I/O framework, we first show the results that compare the write and read time between the original HDF5 and HDF5 with asynchronous I/O support. We configured the micro-benchmarks to write or read 10 timesteps of both data and metadata, stored as datasets and attributes in 10 different groups in one HDF5 file. Each group has 0 or 5 datasets with varying sizes (1MB to 128MB), and a varying number of attributes (0 to 128). By varying the number and size of datasets and the number of attributes, we simulate serial

[1]https://sdm.lbl.gov/exahdf5/ascr/software.html
[2]https://github.com/glennklockwood/bdcats-io

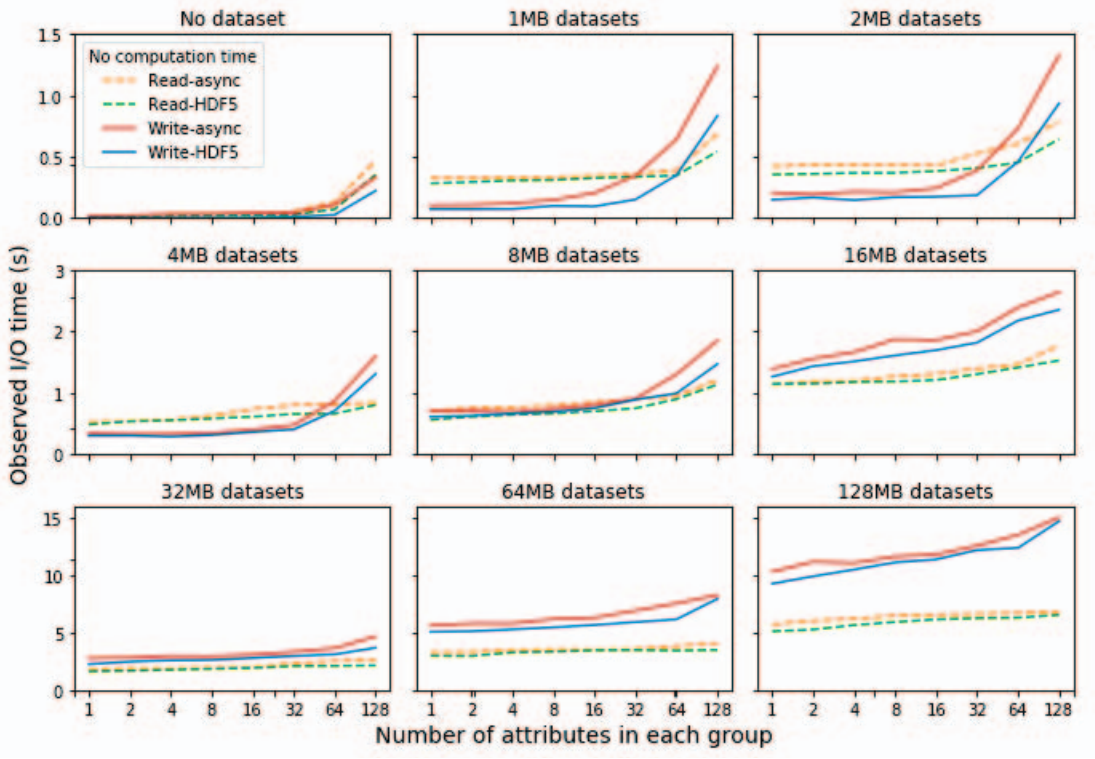

Fig. 3. Performance comparison between asynchronous-enabled HDF5 and the default HDF5, with no computation time between timesteps. The write benchmark creates 10 HDF5 groups, each corresponding to one timestep. Each group contains 0 (No Dataset) or 5 datasets with varying sizes (1MB to 128MB), and a varying number of attributes (from 0 to 128). The read benchmark reads back the previously created data.

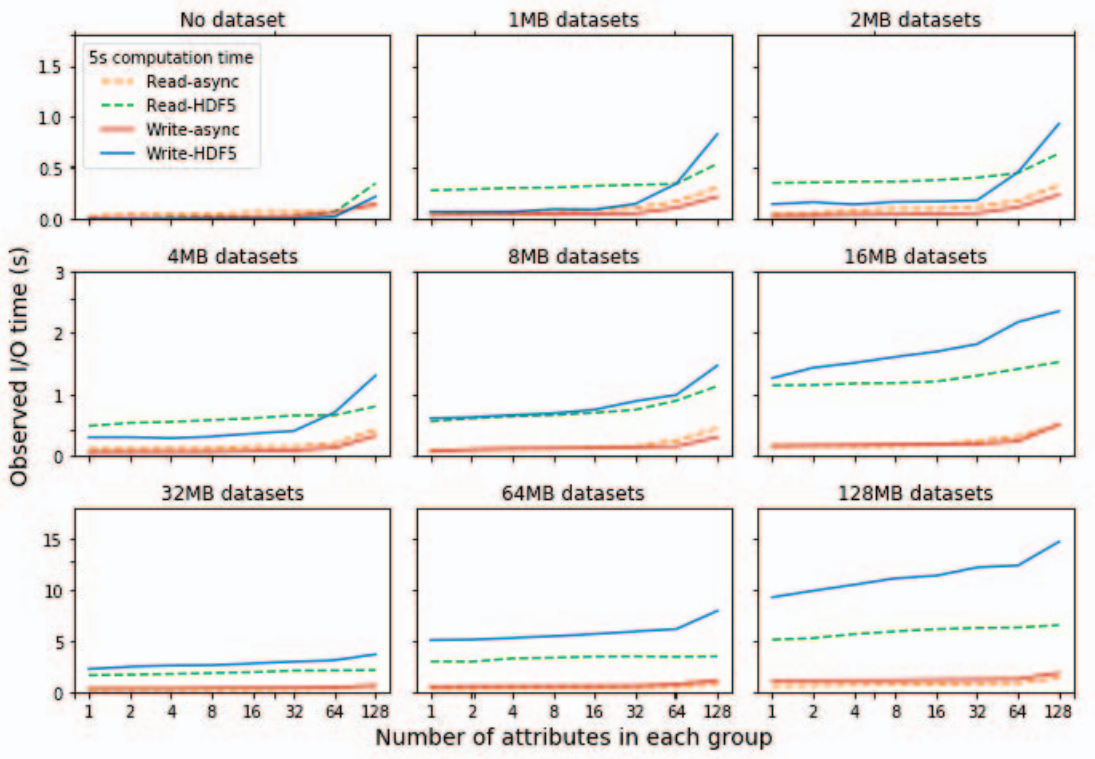

Fig. 4. Performance comparison between asynchronous and the default HDF5 synchronous I/O, with 5 seconds computation time between timesteps. The write benchmark creates 10 HDF5 groups, each corresponding to one timestep. Each group contains 0 (No Dataset) or 5 datasets with varying sizes (1MB to 128MB), and a varying number of attributes (from 0 to 128). The read benchmark reads back the previously created data.

applications that have data-intensive, metadata-intensive, or a mixture of data and metadata workload.

In Figure 3, we compare the asynchronous I/O-enabled HDF5 with the original HDF5 for reading and writing data with *no* computation time between the I/O operations. Essentially the performance difference between the two is the overhead of creating, managing, and executing the asynchronous I/O operations. As the HDF5 library caches metadata in memory and does not flush the metadata to the storage system every time, the metadata operations are almost all memory operations, with an execution time comparable to those of asynchronous task management. Thus the overhead is more significant with the metadata-intensive workload. However, when there are more raw data I/O operations, which have more disk I/O accesses, the overhead becomes lower, as shown in the bottom three cases.

Unlike the previous figure, in Figure 4, we show the performance comparison when there is computation time between the I/O operations, which is more common in time-series data accesses. We set the computation time between each timestep to be 5 seconds, which is more than that of the I/O time and can fully mask the background threads' I/O operations. We can see from the figure the asynchronous I/O-enabled HDF5 has a multi-fold I/O time speedup than the original HDF5, demonstrating the effectiveness of our implementation.

B. Parallel I/O Performance

1) Parallel Metadata I/O: In the previous section, we have observed that with the metadata-intensive workload, a serial application may not get the full benefits from using asynchronous I/O. This is mainly due to the HDF5's metadata cache mechanism, where the majority of the metadata I/O are executed as in-memory operations, and can finish in milliseconds. However, when it comes to parallel metadata I/O, the communication cost among all processes (necessary for HDF5 metadata operations) becomes significant and is much more time-consuming. This added communication cost provides an opportunity to take advantage of asynchronous I/O and hide such cost.

Figures 5 and 6 compare the I/O time using the parallel version of the micro-benchmark with the metadata-only workload. The micro-benchmark writes or reads 10 timesteps, each with 64 attributes. Due to the communication cost within HDF5 for those operations, the total I/O time increases with the number of application processes. "Async-0%" has *no* computation time for the I/O operations to overlap with, and thus has similar performance with the regular HDF5, with the time difference being the asynchronous framework overhead. On the other hand, when there is enough computation time, as shown in the "Async-100%" cases, we can see a significant I/O time reduction, demonstrating the advantage of using asynchronous I/O.

2) Parallel Raw Data I/O: We use the results from running two I/O kernels that perform primarily raw data operations to demonstrate the speedup of our proposed asynchronous I/O framework at large scale. Using the original HDF5 as a baseline comparison, we show three different I/O kernel configurations with asynchronous I/O enabled, by varying the amount of computation time between each timestep.

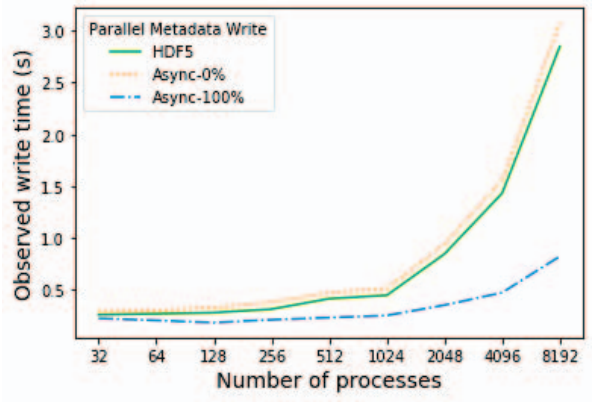

Fig. 5. Parallel metadata write performance comparison between HDF5 and HDF5 with asynchronous I/O support. At each timestep, 64 attributes are created and write to an HDF5 file. "Async-0%" writes all timesteps' data without computation time between the writes, and "Async-100%" has computation time that is more than the metadata write time.

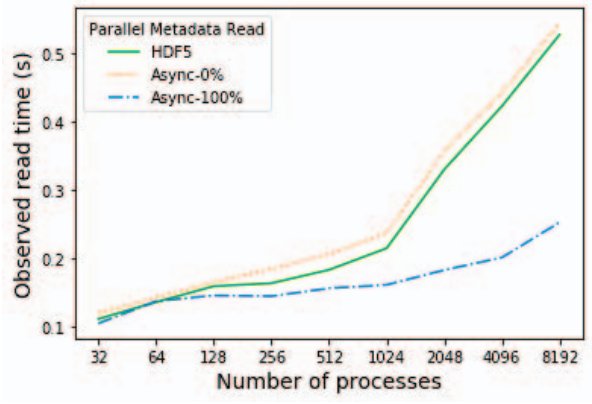

Fig. 6. Parallel metadata read performance comparison between HDF5 and HDF5 with asynchronous I/O support. At each timestep, 64 attributes are read from the HDF5 file. "Async-0%" reads all timesteps' data without computation time in between, and "Async-100%" has computation time that is more than the metadata I/O time.

Figure 7 shows the VPIC-IO performance comparison: with no computation time between the timesteps, our asynchronous I/O-enabled HDF5 shows a small amount of overhead compared to the original version of HDF5. However, when there is computation time, significant performance improvement can be observed. The "Async-50%" case has a computation time after each timestep's write and is approximately half the I/O time of each timestep, with the I/O partially hidden by the computation time. The "Async-100%" case has a computation time that is more than the I/O time of each timestep, such that the I/O time can be fully overlapped with the computation. Note that after the last timestep's write, there is no computation time for the I/O time to overlap with, so the observed write time of "Async-100%" is composed of the last timestep's write time plus the asynchronous I/O framework's overhead of

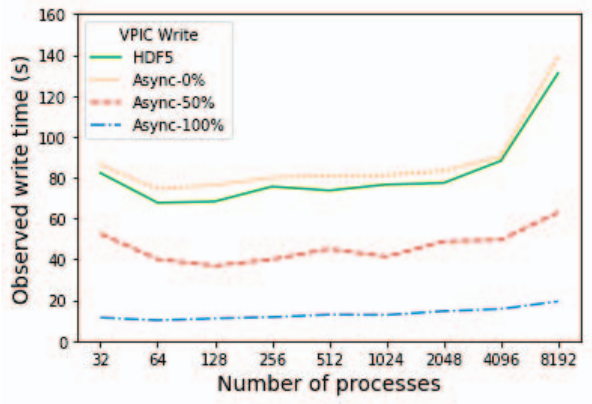

Fig. 7. Performance comparison between HDF5 and HDF5 with asynchronous I/O support. 10 timesteps of VPIC data is written with a varying amount of computation time between the writes. "Async-0%" writes all timesteps' data without computation time between consecutive writes, "Async-50%" has a simulated computation time that is approximately 50% of the individual write time between the writes, and "Async-100%" has computation time that is more than the individual write time.

all 10 timesteps.

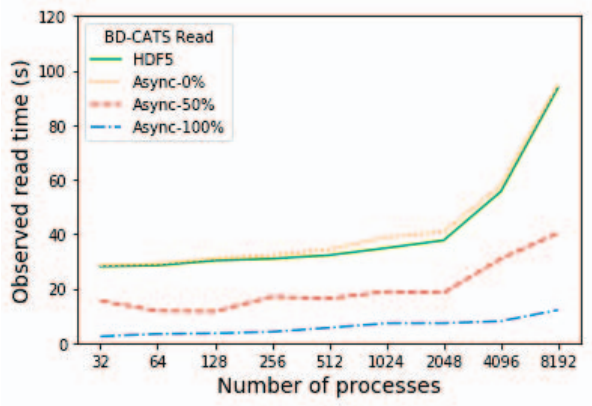

Fig. 8. Performance comparison between HDF5 and HDF5 with asynchronous I/O support. 10 timesteps of VPIC data is read and prefetched with a varying amount of computation time between them. "Async-0%" reads all timesteps' data without computation time between consecutive reads, "Async-50%" has a simulated computation time that is approximately 50% of the individual read time, and "Async-100%" has computation time that is more than the individual read time, such that the prefetching is fully overlap with the computation.

Figure 8 shows the performance comparison using the BD-CATS-IO, which reads the VPIC data. The asynchronous framework's overhead for read operations is almost negligible in most of the cases. Similar to the previous VPIC-IO results, when the computation time is half or more than the I/O time of each timestep, the performance of asynchronous I/O is already several times faster than the original HDF5. Since there is no computation time before the first timestep's read, the observed I/O time for "Async-100%" includes the first timestep's read time and the asynchronous overhead.

C. Overhead

We measured the overhead of our asynchronous I/O framework by running the same workload using HDF5 with and without asynchronous I/O support (shown in previous figures). When there is no computation time between the I/O operations, the overhead is fully exposed to the application. For the serial write and read cases in Figure 3, the overhead ranges from 0.6% to 20% of the total I/O time, with an average of 5%. For the parallel results in Figures 7 and 8, the overhead ranges from 0.8% to 9%, with an average of 4%. On the other hand, if there is some computation time between the I/O operations, the asynchronous I/O overhead can be partially masked by the computation. In this case, the overhead of asynchronous I/O becomes less than 2% of the total I/O time in all cases.

VI. Related Work

As exascale supercomputers are deployed in the next few years, an increasing amount of data will be generated and analyzed. Dealing with such huge volumes of data is not an easy task, and various I/O performance optimizations have been proposed to alleviate the I/O bottleneck and accelerate the process of scientific discovery. Existing parallel file systems such as Lustre [11], PVFS [12], GPFS [13], and NFS [14] aim to provide efficient parallel data access, but still requires a significant amount of expertise and effort to reduce the I/O latency.

There is a growing trend toward enabling applications to use asynchronous I/O. Lazy AIO has been proposed as a general OS mechanism for automatically converting any system call that blocks into an asynchronous call [15], however, it still requires the user to manage data dependencies and is operating with low-level system calls. The light weight file system (LWFS) [16] has been proposed, however, to utilize its asynchronous I/O support, the entire file system must be replaced with LWFS, which is not practical in leadership computing facilities. The impact of various overlapping strategies of MPI-IO has been studied [17], but only at a rather small scale.

Other approaches focus on data staging and caching approaches to offload the I/O to dedicated servers. [18] improves the I/O performance by caching data on additional compute nodes. I/O middleware such as ADIOS [3] provides asynchronous I/O support through their staging interface, where the data can be transferred to the staging nodes through remote direct memory access. Proactive Data Containers (PDC) [4] framework uses extra cores to run I/O servers that move data across the storage hierarchy without blocking the application. Compared with our proposed background thread approach, these methods require extra computing resources to move and store the data, as well as configuration and set up by users.

VII. Conclusions and Future Work

Our results show that enabling asynchronous I/O operations can effectively alleviate the I/O bottleneck in scientific applications. We have presented our design and implementation of an asynchronous I/O framework that supports all types of I/O operations (including collective parallel I/O), manages

the I/O tasks' dependencies transparently and automatically, and has low overhead. Our implementation of an HDF5 VOL connector allows the goal of minimal code changes. The experimental results using benchmarks and I/O kernels demonstrate a multi-fold I/O time reduction.

Our future work includes the support of providing asynchronous task "tokens" to users such that it is easier to track and wait for a group of asynchronous tasks instead of individual ones. We will also apply this work to more applications and to other I/O libraries and frameworks and further optimize the performance.

ACKNOWLEDGMENT

This work was supported by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research, under contract number DE-AC02-05CH11231 (Project: Exascale Computing Project [ECP] - ExaHDF5 project). 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. R. Walli, "The POSIX Family of Standards," *StandardView*, vol. 3, no. 1, pp. 11–17, 1995.

[2] R. Thakur, W. Gropp, and E. Lusk, "On Implementing MPI-IO Portably and with High Performance," in *ICPADS*, 1999, pp. 23–32.

[3] Q. Liu, J. Logan, Y. Tian *et al.*, "Hello ADIOS: the challenges and lessons of developing leadership class I/O frameworks," *Concurrency and Computation: Practice and Experience*, vol. 26, no. 7, pp. 1453–1473, 2014.

[4] 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 *Proceedings of the 18th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*. IEEE Press, 2018, pp. 113–122.

[5] The HDF Group. (1997-) Hierarchical Data Format, version 5. Http://www.hdfgroup.org/HDF5.

[6] S. Seo, A. Amer, P. Balaji, C. Bordage, G. Bosilca, A. Brooks, P. Carns, A. Castelló, D. Genet, T. Herault *et al.*, "Argobots: A lightweight low-level threading and tasking framework," *IEEE Transactions on Parallel and Distributed Systems*, vol. 29, no. 3, pp. 512–526, 2017.

[7] L. Dagum and R. Menon, "Openmp: An industry-standard api for shared-memory programming," *Computing in Science & Engineering*, no. 1, pp. 46–55, 1998.

[8] The HDF Group. (2015-) HDF5 Virtual Object Layer (VOL) documentation. Https://bit.ly/2lIJXem.

[9] K. J. Bowers, B. J. Albright, L. Yin, W. Daughton, V. Roytershteyn, B. Bergen, and T. Kwan, "Advances in petascale kinetic plasma simulation with vpic and roadrunner," in *Journal of Physics: Conference Series*, vol. 180, no. 1. IOP Publishing, 2009, p. 012055.

[10] M. M. A. Patwary, S. Byna, N. R. Satish, N. Sundaram, Z. Lukić, V. Roytershteyn, M. J. Anderson, Y. Yao, P. Dubey *et al.*, "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*. IEEE, 2015, pp. 1–12.

[11] P. Braam, "The lustre storage architecture," *arXiv preprint arXiv:1903.01955*, 2019.

[12] P. Carns, W. Ligon III, R. Ross, and R. Thakur, "PVFS: A Parallel Virtual File System for Linux Clusters," *Linux J.*, 2000.

[13] F. B. Schmuck and R. L. Haskin, "GPFS: A Shared-Disk File System for Large Computing Clusters." in *FAST*, vol. 2, 2002, pp. 231–244.

[14] B. Pawlowski, D. Noveck, D. Robinson, and R. Thurlow, "The NFS version 4 protocol," Network Appliance, Tech. Rep., 2000.

[15] K. Elmeleegy, A. Chanda, A. L. Cox, and W. Zwaenepoel, "Lazy asynchronous i/o for event-driven servers." in *USENIX Annual Technical Conference, General Track*, 2004, pp. 241–254.

[16] R. A. Oldfield, L. Ward, R. Riesen, A. B. Maccabe, P. Widener, and T. Kordenbrock, "Lightweight i/o for scientific applications," in *2006 IEEE International Conference on Cluster Computing*. IEEE, 2006, pp. 1–11.

[17] C. M. Patrick, S. Son, and M. Kandemir, "Comparative evaluation of overlap strategies with study of i/o overlap in mpi-io," *ACM SIGOPS Operating Systems Review*, vol. 42, no. 6, pp. 43–49, 2008.

[18] A. Nisar, W.-k. Liao, and A. Choudhary, "Scaling parallel i/o performance through i/o delegate and caching system," in *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*. IEEE Press, 2008, p. 9.