Transparent Asynchronous Parallel I/O using Background Threads

Houjun Tang, Quincey Koziol, John Ravi, and Suren Byna

Abstract—Moving toward exascale computing, the size of data stored and accessed by applications is ever increasing. However, traditional disk-based storage has not seen improvements that keep up with the explosion of data volume or the speed of processors. Multiple levels of non-volatile storage devices are being added to handle bursty I/O, however, moving data across the storage hierarchy can take longer than the data generation or analysis. Asynchronous I/O can reduce the impact of I/O latency as it allows applications to schedule I/O early and to check their status later. I/O is thus overlapped with application communication or computation or both, effectively hiding some or all of the I/O latency. POSIX and MPI-I/O provide asynchronous read and write operations, but lack the support for non-data operations such as file open and close. Users also have to manually manage data dependencies and use low-level byte offsets, which requires significant effort and expertise to adopt. In this paper, we present an asynchronous I/O framework that supports all types of I/O operations, manages data dependencies transparently and automatically, provides implicit and explicit modes for application flexibility, and error information retrieval. We implemented these techniques in HDF5. Our evaluation of several benchmarks and application workloads demonstrates it effectiveness on hiding the I/O cost from the application.

Index Terms—Asynchronous I/O, parallel I/O, background threads.

1 Introduction

With the dawn of exascale high performance computing (HPC) systems in the coming years, new challenges arise for scientific data management. It is expected that an unprecedented amount of data will be generated with the increasing computation power. However, storing the produced data and retrieving the data efficiently are challenging tasks, as the I/O sub-system of the supercomputers has not kept up with the pace of CPU and network speed improvements, making it likely to become a bottleneck and limit overall application performance.

Large-scale applications running on supercomputers typically perform computation and I/O in phases. For instance, in simulations, after computation of a pre-set number of timesteps, a snapshot of the data is written to storage. Several simulations also perform checkpointing of the state of the simulation for tolerating any application failures. Similarly, analysis applications iterate between reading data and analyzing it. However, HPC applications using I/O libraries such as HDF5 have to wait until a given I/O phase is complete before continuing with computations. This synchronous I/O causes significant overhead due to slow performance of the disk-based file systems.

To reduce the I/O performance bottleneck further and to improve the efficiency of data-intensive applications by utilizing fast storage layers, *asynchronous I/O* has become a popular and effective option. Applications can take advantage of asynchronous operations by scheduling I/O tasks as early as possible, overlapping them with communication and computation, and check the I/O operations' completion status later when needed. This overlapping can allow an application to hide some or all of the costs associated with the I/O.

Despite several existing interfaces and systems to support asynchronous I/O, most applications have yet to take advantage of this approach. The POSIX I/O [1] and the MPI I/O [2] interfaces support asynchronous data read/write operations through the "aio_*" and "MPI_I*" interfaces, respectively. However, they are rarely used by applications that use a high-level I/O middleware library such as HDF5, because most middleware does not provide an asynchronous interface. POSIX and MPI I/O also lack support for asynchronous non-data I/O operations, such as file open and close and they operate on byte offsets and sizes, requiring extra effort and expertise to convert existing application code to use asynchronous I/O features. Some high-level I/O libraries and data management systems, such as Data Elevator [3] and Proactive Data Containers (PDC) [4], provide a level of asynchrony, but these systems require extra server processes in addition to the application to move data without blocking the client processes, at the cost of extra computing resources and user effort.

There are several challenges to providing transparent asynchronous parallel I/O support in a way that minimizes changes to application code. It is critical to manage data dependencies and retain the correct order of operations when executing I/O tasks asynchronously, as some operations may depend on previous operations' successful completion. For example, any data read or write operations must only be executed asynchronously after a successful asynchronous file create/open, and operations that need collective communication must be executed in the same order across all processes to prevent deadlock. Additionally, to minimize the impact on the application, asynchronous tasks should start execution when the application is no longer issuing I/O operations. Finally, application developers should not be burdened with manual management of all these operations and should have a low-effort mechanism to take advantage of the asynchronous operations, monitor their status, and check for errors.

- *Houjun Tang, Quincey Koziol, and Suren Byna are with Lawrence Berkeley National Laboratory, Berkeley, CA 94720. E-mail: {htang4, koziol, sbyna}@lbl.gov*
- *John Ravi is with NC State University, Raleigh, NC 27606. E-mail: jjravi@ncsu.edu*

Manuscript received ; revised .

To tackle these challenges, we propose an asynchronous I/O framework that *supports all types of I/O operations* – including both independent and collective parallel I/O, *requires no additional servers, manages data dependencies transparently and automatically from users, provides aggregated operation status and error checking, and requires minimal code modifications*.

Our implementation of asynchronous I/O uses background threads, as the asynchronous interface offered by existing operating systems and low-level I/O frameworks (POSIX AIO and MPI-IO) does not support all file operations. We have implemented this approach for the HDF5 [5] I/O library. HDF5's Virtual Object Layer (VOL) allows interception of all operations on a file and VOL connectors can perform those operations using new infrastructure, such as background threads.

Our VOL connector maintains a queue of asynchronous tasks and tracks their dependencies as a directed acyclic graph, where a task can only be executed when all its parent tasks have been completed successfully. Collective operations are executed in the same order as in the application, in an ordered but asynchronous manner. To reduce overhead and avoid contention for shared resources between an application's main thread and the background thread that performs the asynchronous I/O operations, we use a status detection mechanism to check when the main thread is performing non-I/O tasks. We also provide an EventSet interface in HDF5 to monitor asynchronous operation status and to check errors for a *set* of operations instead of individual ones.

In summary, our method makes the following contributions:

- We adopt a background thread approach that accumulates I/O operations and starts their execution when the background thread detects that the application's main thread is idle or performing non-I/O operations.
- We develop a task dependency management protocol to guarantee data consistency and support both collective and independent operations.
- We provide transparent asynchronous I/O support to HDF5 applications, in a way that requires no more than a few lines of code changes when using this "implicit" mode.
- We provide the HDF5 EventSet interface to application developers, so they can fully utilize our asynchronous I/O framework with low-effort control and error checking.
- We show the evaluation of our implementation with several benchmarks and applications to demonstrate the effectiveness of the asynchronous I/O.

We have evaluated this asynchronous I/O framework on Summit, the second fastest supercomputer in the top500 list [6], located at the Oak Ridge Leadership Computing Facility (OLCF) and on the Cori supercomputer at the National Energy Research Scientific Computing Center (NERSC) with several I/O kernels and scientific application workloads. Experimental results show that our method can effectively hide the cost of I/O when the application is performing non-I/O operations such as computation or communication.

The remainder of the paper is organized as follows: We introduce our asynchronous I/O framework's design in Section 2, and the implementation as an HDF5 VOL connector in Section 2.4. In Section 3, we describe our experimental setup and present the results in Section 4. We discuss the relevant literature in Section 5 and conclude the paper in Section 6.

2 Asynchronous I/O Framework

Asynchronous I/O can significantly reduce the I/O time for data-intensive applications, as the I/O operations can fully or partially overlap with the computation and communication. It is especially effective for applications that write or read data periodically, e.g., time-series data simulations and analyses. In simulations, only the last step's write or in analysis applications first step's read time cannot be overlapped and all other write or read operations can be overlapped. The observed I/O cost of overlapped I/O will be near-zero if the cost of asynchronous I/O management is negligible.

In Figure 1, we illustrate the architecture of our framework. When asynchronous I/O is enabled, a background thread is automatically started for each of the application's processes. We intercept all I/O operations and create corresponding asynchronous tasks. The asynchronous tasks are stored in a queue for dependency evaluation and later execution. Our background thread monitors the running state of the application's main thread, and only starts executing the previously accumulated tasks when it detects the application is no longer issuing I/O operations. When an application is shutting down, the asynchronous I/O framework executes any remaining I/O operations, frees resources, and terminates the background thread.

We have implemented our asynchronous I/O framework as an HDF5 VOL connector (more details in Section 2.4), as HDF5 is a popular I/O middleware library that is used by a wide range of scientific applications [7], [8], [9].

In the following sections, we describe implicit and explicit asynchronous operation modes. We also provide details of asynchronous task management, including dependency tracking, application thread status detection, and background thread execution.

2.1 Implicit and Explicit Mode

We provide two ways for applications to use the I/O in HDF5 – an *implicit* mode, which requires minimal code changes but has performance limitations and *explicit* asynchronous operations, which requires some code changes but can take full advantage of asynchronous execution. In the implicit mode, the user only needs to initiate the use of asynchronous I/O by running the application with an environment variable set. In the explicit mode, the application must be modified to bundle asynchronous I/O operations into EventSets. The explicit mode gives more control to applications over when to execute asynchronous operations and a better mechanism for detecting errors.

2.1.1 Implicit Mode with Environment Variable

Implicit asynchrony is the least intrusive method for application developers to benefit from asynchronous I/O. It allows developers to use their existing code, without managing asynchronous requests or explicitly waiting for operations to complete, while transparently performing operations asynchronously when safely possible. The HDF5 VOL connector framework (more details in Section 2.4) supports implicit mode by setting environment variables and dynamically loading a connector: the user simply sets two environment variables: `HDF5_PLUGIN_PATH`="/path/to/async_lib" to specify the asynchronous I/O dynamic library's location and `HDF5_VOL_CONNECTOR`="async under_vol=0;under_info={}" to specify using the asynchronous I/O framework for I/O operations when running the application. I/O operations will then be transparently executed in a background thread and fully managed by the asynchronous I/O framework. This approach enables an

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2021.3090322, IEEE Transactions on Parallel and Distributed Systems

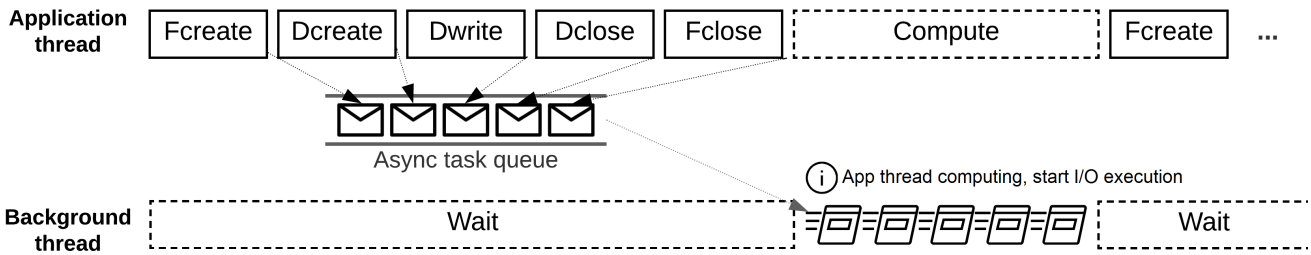


Fig. 1: **An overview of our proposed asynchronous I/O framework**. *Fcreate* and *Dcreate* refer to file and dataset creation, respectively. *Dwrite* refers to writes to a dataset. *Dclose* and *Fclose* are to close the dataset and the file, respectively. When an application's main thread is performing compute operations, our background thread executes operations from the "async task queue" and waits for more I/O operations. If the compute phase is shorter, the main thread may start adding more tasks into the queue.

application to execute I/O tasks asynchronously in a *completely* compatible manner to its current synchronous behavior.

However, there are several limitations of the implicit mode, as we need to guarantee data consistency for the unmodified application. First, all read operations (including metadata 'get' operations) are executed synchronously and wait for previous implicit asynchronous operations to complete before executing, to prevent the application from using a buffer that has not been filled by a background asynchronous read operation. Second, the file close call is also synchronous and blocking, as it needs to wait for all previous asynchronous tasks to complete, ensuring that all I/O operations have been completed before an application exits. Third, the implicit mode has an *optimistic* view regarding asynchronous task execution, as the execution of the actual I/O operation often happens well after its function call, error checking and debugging becomes very difficult if not impossible. Additionally, to allow applications to reuse or free a buffer after a write call, we make a copy of the data from the user's buffer at task creation time by default, which requires more temporary memory space before the write task completes and the copied buffer is freed.

2.1.2 Explicit Mode with EventSet API

In the explicit mode, we provide a direct specification of asynchronous I/O operations. As opposed to the implicit mode, the explicit mode requires modifying an application's source code to replace existing I/O functions with their corresponding asynchronous version, but this can be done with simple find-and-replace with minimal effort. The explicit mode allows an application to fully use all asynchronous I/O features, lifting the limitations in the implicit mode.

To make it easier for existing HDF5 applications to transition to this approach, and avoid managing individual operations, we offer *EventSet* APIs that can track and inspect multiple I/O operations. An event set is an in-memory object that is created by an application, and functions similar to a "bag" – holding request tokens from one or more asynchronous I/O operations.

To support asynchronous operations, the HDF5 API routines have been extended with asynchronous versions of each API routine that operates on a file. These new API routines add `_async` as the suffix of existing routines such as `H5Fcreate` (HDF5 file create) or `H5Dcreate` (HDF5 dataset create), and an extra parameter to pass in an EventSet ID.

EventSet management routines are also added: `H5EScreate` creates an event set which can be associated with multiple asynchronous operations, `H5ESget_count` returns the number of operations in an event set, `H5EStest` checks whether there is any incomplete operations, `H5ESwait` waits for operations to complete within a user-specified time threshold, `H5EScancel` attempts to cancel all operations in an event set (in-progress operations might not be canceled), and `H5ESclose` closes the event set. In Figure 2, we show an example of converting existing HDF5 I/O calls to corresponding asynchronous I/O calls and enable asynchronous I/O with the event set APIs.

We recommend application developers to try their application with the implicit mode of our asynchronous I/O VOL connector first, which would execute most of the write operations asynchronously. Once they are familiar with using the asynchronous I/O VOL connector, depending on the requirement to control asynchronous I/O operations, we suggest to convert their existing application code to use the explicit mode using the EventSet APIs. The EventSet API provides full asynchronous execution capabilities and error handling. We currently do not fully support mixing implicit and explicit modes, since whenever a non-async API is invoked, it can lead to synchronous execution that blocks and waits for all previous I/O operations to finish. The event set functions such as `H5ESwait` will also not include operations that are called with the implicit mode, which may cause unexpected behaviors.

2.1.3 Memory Management

With asynchronous execution of I/O tasks, memory usage becomes an aspect that must be carefully managed. For the implicit mode write operations, a double-buffering approach is used that duplicates the data in a temporary buffer. The additional memory may be significant to memory-intensive applications, so we recommend applications developers make the necessary calculations to prevent the copied buffers from exceeding the memory limit at runtime. For the explicit mode, we rely on the application to manage the buffers and make sure they are not modified or freed before the operation completes. We plan to utilize node local storage, such as SSDs, as a temporary cache location to replace the memory double-buffering, which would write to the SSD first and then flush the data from there to the parallel file system asynchronously.

2.1.4 Error Reporting

Using asynchronous I/O comes with delayed feedback for the status of operations. The default, synchronous, HDF5 API routines return an error status immediately, but asynchronous versions of API routines only return the status of creating the asynchronous task, as the actual I/O operation is executed by a background thread at a future time. However, effective error reporting is critical to users when locating root causes of failures and must be provided by an asynchronous I/O framework before users will rely on it

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2021.3090322, IEEE Transactions on Parallel and Distributed Systems


```
// Synchronous file create
fid = H5Fcreate(...);
// Synchronous group create
gid = H5Gcreate(fid, ...);
// Synchronous dataset create
did = H5Dcreate(gid, ..);
// Synchronous dataset write
status = H5Dwrite(did, ..);
// Synchronous dataset read
status = H5Dread(did, ..);
...
// Synchronous file close
H5Fclose(fid);
// Continue to computation

...




...
// Finalize
```

```
// Create an event set to track async operations
es_id = H5EScreate();
// Asynchronous file create
fid = H5Fcreate_async(.., es_id);
// Asynchronous group create
gid = H5Gcreate_async(fid, .., es_id);
// Asynchronous dataset create
did = H5Dcreate_async(gid, .., es_id);
// Asynchronous dataset write
status = H5Dwrite_async(did, .., es_id);
// Asynchronous dataset read
status = H5Dread_async(did, .., es_id);
...
// Asynchronous file close
status = H5Fclose_async(fid, .., es_id);
// Continue to computation, overlapping with asynchronous
    operations
...
// Finished computation, Wait for all previous operations in the
    event set to complete
H5ESwait(es_id, H5ES_WAIT_FOREVER, &n_running, &op_failed);
// Close the event set
H5ESclose(es_id);
...
// Finalize
```

Fig. 2: Example code showing converting existing HDF5 code (left) to utilize the explicit EventSet asynchronous I/O APIs (right).

for production use. To achieve this, we record an error stack for asynchronous tasks when a failure occurs, and prevent the remaining tasks that depend on the successful execution of the failed task from being executed, as well as also preventing the addition of more tasks to an event set with unhandled failures. We have provided an API to query the error status of operations in an event set, as we show in Figure 3. Using these API functions makes it possible for users to trace root causes of failures from operations in an event set.

2.2 Asynchronous Task Management

2.2.1 Asynchronous Tasks

We enable asynchronous execution by creating asynchronous tasks for each operation and defer its execution to a future time. An asynchronous task object is a transient in-memory object holding all the information needed for executing a specific operation. It may include a copy of all its parameters, a callback function pointer, data pointers, and internal states such as its dependency and execution status. This converts a blocking I/O operation into a non-blocking operation – once the task is created and put into the asynchronous task queue, the function can return without waiting for its completion. These asynchronous tasks are internal data structures and are not exposed to users.

When the background thread begins executing I/O operations, it chooses the oldest task in the asynchronous task queue that has either no dependent operations or all of its dependent operations have successfully completed. This operation then runs normally, as in the synchronous approach, but in the background thread. After it completes, the next task is dequeued in a similar way. More details on deciding when to start background thread execution are given in Section 2.3. The additional memory requirements for managing asynchronous tasks are minimal, less than 1KB for each task and they are freed once the corresponding operation completes.

2.2.2 Task Dependency Management

To ensure data consistency for asynchronous execution, our framework automatically tracks and maintains the dependencies of all asynchronous tasks. Tasks that have dependent operations are paused until their dependencies are fulfilled and then scheduled to run when a background thread becomes available. This approach allows an existing application to execute all of its I/O operations asynchronously and still have confidence that they will be executed in the correct order. This approach is useful for applications writing checkpoint files regularly, as the I/O time can be effectively masked by the compute time between checkpoints. To determine the dependency among tasks, we adopt a rule-based approach, which includes the following rules [10]:

- All I/O operations can only be executed after a successful file create/open.
- A file close operation can only be executed after all previous operations in the file have been completed.
- All read or write operations must be executed after a prior write operation to the same object.
- All write operations must be executed after a prior read operation to the same object.
- All collective operations must be executed in the same order with regard to other collective operations.
- Only one collective operation may be in execution at any time (among all the threads on a process).

Figure 4 illustrates the task dependencies for 7 tasks on different objects in one file. There are three types of tasks with different colored boxes, the white/transparent color box (i.e., box labeled with "1") is a task that has no dependent parent and can be executed at any time. The light grey color boxes (i.e., 2,3,4 and 7) are tasks that have dependent parents, based on the above rules, and must wait for its parent operations to complete before being executed. The dark gray color boxes (i.e., 5 and 6) are collective tasks that must be executed in their original order. With these rules, it is possible that some asynchronous tasks may not be executed in the same order as in the application's code. Two out-of-order execution scenarios may occur: 1) independent (non-collective) read operations on the same or different objects, and 2) independent write operations on different objects. These two

```
// Check if event set has failed operations (es_err_status is set to true)
status = H5ESget_err_status(es_id, &es_err_status);
// Retrieve the number of failed operations in this event set
status = H5ESget_err_count(es_id, &es_err_count);
// Retrieve information about failed operations
status = H5ESget_err_info(es_id, 1, &err_info, &es_err_cleared);
// Retrieve the failed operations' API name, arguments list, file name, function name, and line number
printf(``API name: %s, args: %s, file name: %s, func name: %s, line number: %u'', err_info.api_name,
    err_info.api_args, err_info.api_file_name, err_info.api_func_name, err_info.api_line_num);
// Retrieve operation counter and operation timestamp
printf(``Op counter: %llu, Op timestamp: %llu'', err_info.op_ins_count, err_info.op_ins_ts);
```

Fig. 3: Error checking and reporting for asynchronous operations in an event set.

exceptions would not cause data consistency issues and thus are allowed in our framework.

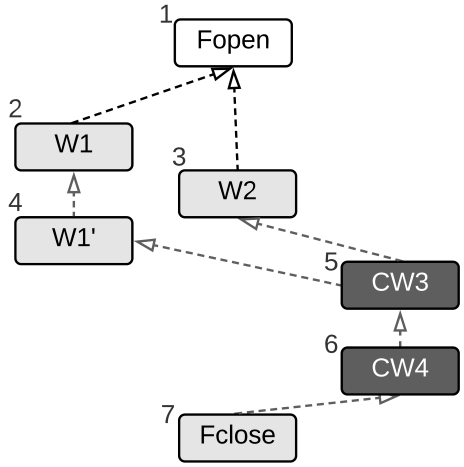


Fig. 4: An example of asynchronous task dependency management, with the labeled number as their order of issue in the application. 1 (Fopen) is a regular task with no dependency; 2 and 3 (W1, W2) are writes to objects 1 and 2, and are both dependent on the previous file open operation; 4 (W1') also writes to object 1 and is dependent on W1; 5 and 6 (CW3, CW4) are collective writes on objects 3 and 4, and must be executed in the same order as in the application; 7 (Fclose) depends on all previous operations.

2.3 Application Thread Status Detection

As mentioned previously, there must be a mechanism to decide when a background thread may start executing tasks in the asynchronous task queue. Simply start executing the tasks as soon as they enter the queue will not lead to effective asynchronous execution, as task execution could happen concurrently in the background thread and the application's main thread, and it is possible that the two threads compete for access to shared resources, thus blocking each other's progress. This may lead to the worst-case scenario where the application and the background thread execute tasks interleaved, making all the asynchronous tasks effectively synchronous.

Our solution to this problem is to actively check the application thread's status. We maintain a counter that tracks the number of accesses the application's main thread makes to the shared resources, and the background thread delays its execution when it sees the counter value increasing in a short time window (typically, the time gap between consecutive accesses is at the microsecond level when the application is actively issuing I/O operations). This value check is done by repetitively retrieving the counter value twice, with a sleep time in between (default 200 microseconds), and only stops when the counter value does not increase. We then infer that the application has finished its I/O phase and we can begin executing queued I/O operations with a background thread.

This approach works well with all our experimental runs in Section 4, as their I/O calls are grouped together, with zero to a few non-HDF5 API calls in between. It does add 200 us overhead for each operation, but is negligible for most I/O operations as they have much higher overhead. We also have a dynamic sleep time adjusting mechanism, which increases the time when it detects that the application had issued I/O calls during the background thread execution by checking the counter value before and after the execution.

2.4 HDF5 VOL Connector Implementation

HDF5 [5] is a popular I/O middleware library with a self-describing file format. It provides an abstraction layer to manage data and metadata within a single file. HDF5 relieves the user from manual file management such as file space allocation and seeking specific offsets to access data. It is used widely in many science domains to manage various data models and is used for efficient parallel I/O in HPC simulations and machine learning analyses [7], [8], [9].

Currently, HDF5 does not support asynchronous I/O, and adding asynchronous I/O support to it would benefit a large number of existing applications, especially on exascale computing architectures. Recent HDF5 versions provide a new framework, called the Virtual Object Layer (VOL) [11], [7], which enables dynamic interception of I/O operations at runtime. The VOL framework allows third-party VOL *connectors* to dynamically intercept HDF5 API calls and implement extensions to those operations or access to new types of storage, enabling transparent changes to application behavior without modification of the application. The user continues to use the same HDF5 data model where access is done to a single HDF5 "container", while the VOL connector translates what operations the application performs into how data is actually stored. We have implemented our asynchronous I/O framework as an HDF5 VOL connector, which can be compiled as a dynamic library and linked to the user's application at runtime, while remaining separate from the installed version of HDF5.

We've chosen to use a background threading model to implement asynchronous operations. As the number of available CPU threads per processor has increased in the modern CPU architecture, and applications offload some computation tasks to accelerators such as GPUs and FPGAs, there is frequently a surplus of CPU threads that can be utilized to perform I/O operations. Additionally, new HPC architectures and early designs for

future exascale systems support I/O forwarding, where compute nodes use a fast network interconnect to forward I/O requests to dedicated servers that handle them. These resources can be used to offload I/O operations, without additional hardware or system software modification. The background threads in our asynchronous I/O framework are managed by Argobots [12], a lightweight low-level thread scheduling and execution package. The threading interface in our asynchronous framework is abstracted to replace the Argobots library with any other libraries.

Currently we use one background thread per process for executing asynchronous tasks. We use only a single background thread because HDF5 has a global mutex that must be acquired at the start of all HDF5 API calls and is released right before the function returns. Using multiple background threads would effectively have the same performance as using one thread, as there can only be one thread executing an I/O operation at any given time, with others waiting for it to release the global HDF5 mutex. The background thread can affect the application's performance if all the cores are used by the application, so we recommend users spare some resources for the background thread to minimize such impact. However, as applications are starting to offload the computation to accelerators such as GPUs and FPGAs, the CPU cores are often idle during the computation phase, which the asynchronous I/O background thread can take full advantage of. The asynchronous tasks that are ready to be executed are pushed to the Argobots execution stream for immediate execution.

Adding asynchronous operations to the HDF5 library requires correctly supporting a large number and variety of HDF5 operations, which fall into the following three categories:

1) **Metadata operations**: These operations create or open HDF5 objects such as files, groups, datasets, and attributes; operations that modify existing objects such as extending dataset dimensions, write an attribute, etc.; operations that query groups such as get the number of links it contains, get the datatype for a dataset, read an attribute, etc.; and operations that close objects such as close a group or a file.
2) **Raw Data operations**: An HDF5 dataset contains a collection of data elements, or raw data. The main I/O raw data operations are read or write HDF5 datasets (i.e., `H5Dwrite` and `H5Dread`, respectively).
3) **In-memory (local) operations** The local operations help users manage objects but do not directly result in actual file I/O operations such as creating or setting property lists, IDs, dataspaces, etc. These operations do not need to be performed asynchronously.

In addition, we utilize the new support for "future" IDs in HDF5. A future ID is an HDF5 ID object (i.e., an `hid_t`) that is returned from an asynchronous operation and has not yet completed execution. It has an internal flag to indicate that the object it refers to is not actually available and is still being created or opened. A future ID is indistinguishable from a "normal" HDF5 ID from an application's perspective and can be used in any HDF5 API call where a normal ID is appropriate. If a future ID is used in an HDF5 API call that returns information to the application, as opposed to a call that returns another ID (e.g., `H5Dget_space`), that call will block until the future ID resolves to an actual object and the queried information can be returned. When a future ID's operation completes execution, the ID's state is transparently updated, without changing its value and without the application's involvement.

3 Experimental Setup

We have evaluated the performance of the HDF5 asynchronous I/O framework using I/O kernels that are representative of simulation and analysis applications as well as real scientific workloads. We conducted our experiments on two supercomputing platforms: Cori at the National Energy Research Scientific Computing Center (NERSC), and Summit at Oak Ridge Leadership Computing Facility (OLCF). Cori is a Cray XC40 supercomputer with 2,338 Intel Xeon "Haswell" nodes, where each node consists of 32 cores and 128 GB memory. We use the Lustre parallel file system on Cori, which has 27 PB storage capacity, with 248 OSTs and 700 GB/s peak performance. Summit is an IBM system consisting of 4,608 compute nodes, with each node containing 2 IBM POWER9 processors (2×22 CPUs) and 6 NVIDIA V100 accelerators (6 GPUs). Summit is connected to Alpine, the center-wide IBM Spectrum Scale file system (GPFS), which provides nearly 250 PB of storage capacity and a 2.5 TB/s peak I/O bandwidth.

To measure the performance and demonstrate the effectiveness of our asynchronous I/O framework, we have used two I/O kernels: VPIC-IO and BD-CATS-IO, as well as two AMReX application workloads: Nyx and Castro. For all the experiments, we have configured the application to write 5 timesteps of data with 'sleep' time in between to represent computation phases, such that the asynchronous I/O operations can fully overlap with it. As a result, only the last timestep's write time or the first timestep's read time plus the asynchronous I/O framework's overhead are observed by the application.

To compare the performance among different approaches, we measure and report 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/sleep time. We ran each experiment at least 10 times and report the **median** value. The observed variance of different runs for the same configuration is less than 10%. for a majority of cases. For Lustre on Cori, we set the stripe count to 128 and the stripe size is 16 MB. Each run writes to or reads from a different file to avoid any caching effects.

3.1 VPIC-IO

The VPIC-IO kernel[1] is a parallel I/O kernel extracted from VPIC [13], a plasma physics code that simulates kinetic plasma particles in a multi-dimensional space [8]. It has a highly regular write pattern, with each MPI process writing eight properties for each particle. There are a total of 8M (8×2^{20}) particles, each with 8 32-bit values, for a total of $256MB$ data from each process, all written to a single HDF5 file. As a fixed amount of data is written by each process, VPIC-IO is a weak scaling test. Besides the original version of VPIC-IO, we have also created new versions that use asynchronous I/O in the explicit mode. For the implicit mode, we simply set environment variables for the HDF5 and asynchronous VOL paths and run the original, synchronous version of VPIC-IO.

3.2 BD-CATS-IO

The BD-CATS-IO kernel[2] is extracted from a parallel clustering algorithm code [9], which represents the I/O read patterns used to analyze the particle data produced by applications such as VPIC.

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

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2021.3090322, IEEE Transactions on Parallel and Distributed Systems

Its read pattern matches that of VPIC-IO, such that data related to the particles are read among all the MPI processes with an even distribution, and is also a weak scaling test. We have also created new versions of the BD-CATS-IO code to use the asynchronous I/O capability. With the explicit mode, it asynchronously reads the next timestep's data, i.e., prefetches the data, before processing the current timestep, allowing the background thread to overlap I/O with the application's computation.

3.3 Nyx

Nyx [14] is a massively-parallel adaptive mesh cosmological simulation code that solves equations of compressible hydrodynamics flow. Nyx uses the AMReX framework [15], which allows for a variety of algorithms, discretizations, and numerical approaches, and supports different programming models such as MPI, OpenMP, and GPU. AMReX has an HDF5 output option that writes out one file with the adaptive mesh refinement (AMR) and application-specific metadata together with the simulation data for each checkpoint step. We used a simulation configuration extracted from a Nyx run and replaced all the computation with a 'sleep' time between data writes, allowing us to write the exact same amount of data with the same data structures without having to perform the actual computation, which is very time-consuming. Each output file has **1** refinement level with 262144 AMR boxes, and approximately $385GB$ data per HDF5 output, which includes the names of the components, dimension and coordinate system information, number of grids, the location and sizes of AMR boxes, the offsets of the data corresponding to different boxes in a flattened array, and the data, etc.

3.4 Castro

Castro [16] is an adaptive-mesh compressible radiation / MHD / hydrodynamics code for astrophysical flows. It is also an AMReX [15] application. Similar to Nyx, we extracted the workload from a Castro run, which writes 6 components in **3** adaptive mesh refinement (AMR) levels, with 4096, 8192, and 49152 AMR boxes, and approximately $559GB$ data to an HDF5 plot file per checkpoint. Nyx and Castro are different from VPIC-IO and have write patterns that include both small (metadata) and large (raw data) writes, which allow us to demonstrate the broad applicability of our asynchronous I/O framework.

4 Results

We use the results from running two I/O kernels (VPIC-IO and BD-CATS-IO) that perform primarily raw data operations and two real scientific application workloads (Nyx and Castro) that have both metadata and raw data operations with complex datatypes to demonstrate the effectiveness and generic applicability of our proposed asynchronous I/O framework at scale. In all the plots below, we are showing the observed I/O time (without any emulated computation time). Using the original HDF5 in the synchronous mode as a baseline, we compare the performance of both the implicit and explicit asynchronous I/O modes. The baseline is labeled as "HDF5", asynchronous I/O in the implicit mode as "Async-implicit", and asynchronous I/O in the explicit mode as "Async-explicit".

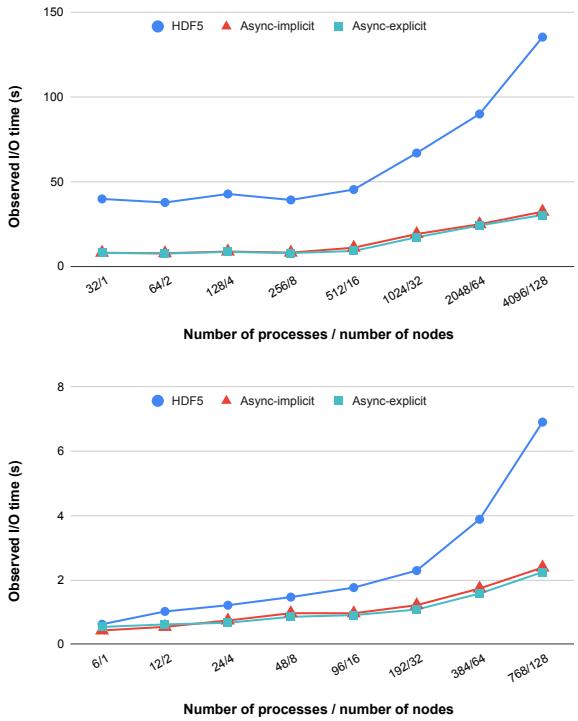


Fig. 5: VPIC-IO write performance comparison on Cori (top) and Summit (bottom) between synchronous HDF5, HDF5 with implicit asynchronous mode, and HDF5 with explicit asynchronous mode. 5 timesteps of VPIC data are written with sufficient compute/sleep time between timesteps. Observed I/O time includes the last timestep's write time and the overhead of the asynchronous I/O framework for all timesteps.

4.1 VPIC-IO

In Figure 5, we compare the observed I/O time (without the emulated computation time with sleep operations) of VPIC-IO on Cori and Summit. As mentioned previously, VPIC-IO writes a fixed amount of data (256 MB) per MPI rank for 8 variables, and increasing the number of processes / nodes also increases the total amount of the data written. We configured VPIC-IO to write 5 timesteps of data, all to a single HDF5 file with each timestep in a different HDF5 group. For all three cases, we added sleep time, which represents the computation time in real application runs, that are sufficient for the I/O time to fully overlap with, which is typical with the VPIC application. In these experiments, we have set the sleep time to be up to 60 seconds on Cori and up to 20 seconds on Summit. In an actual VPIC simulation, computation time is typically more than 1000 seconds [8], hence, the sleep time we used to overlap the entire write time is reasonable. For the results on Cori, we used 32 processes per node and increase the number of nodes from 1 to 128; while on Summit, applications typically run with 6 MPI ranks per node to match and utilize the 6 GPUs, which is why we chose to run VPIC-IO and all other applications using 6 processes per node.

Comparing the observed I/O time on both systems, the HDF5 case with synchronous I/O performs the slowest, as expected. Asynchronous I/O with both the implicit and explicit modes is up to 4.8X faster than writing data synchronously. The performance difference between implicit and explicit modes in this use case is minimal, as they both execute the I/O operations asynchronously in a similar way. For the implicit mode, the data is copied to a

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2021.3090322, IEEE Transactions on Parallel and Distributed Systems

temporary buffer at asynchronous task creation time, while the explicit mode can skip copying the data since the buffer is not reused or freed by the application code. However, the data copy is a very fast memory copy operation and thus the implicit mode performs similarly to the explicit mode.

4.2 BD-CATS-IO

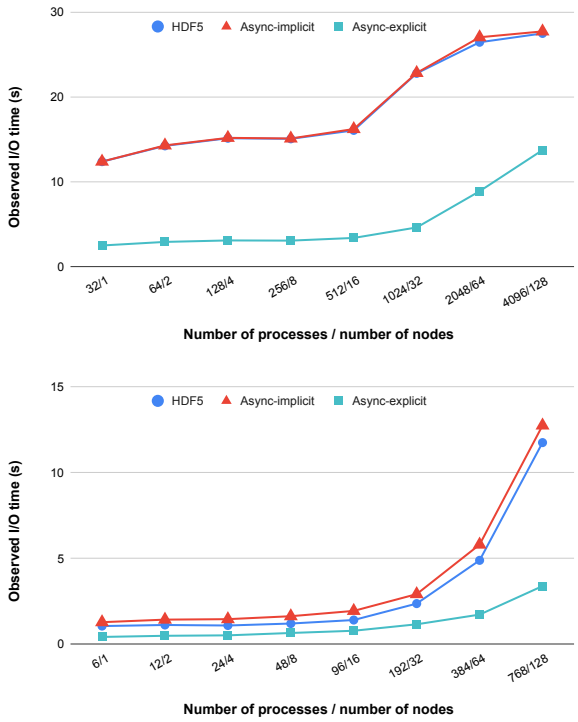


Fig. 6: BD-CATS-IO read performance comparison on Cori (top) and Summit (bottom) between synchronous HDF5, HDF5 with implicit asynchronous mode, and HDF5 with explicit asynchronous mode. 5 timesteps of VPIC data is read with sufficient compute/sleep time between timesteps. Observed I/O time includes the first timestep's read time and the overhead of asynchronous I/O framework for all timesteps.

The BD-CATS-IO kernel reads the data produced by the VPIC-IO kernel. In Figure 6, we show a performance comparison of BD-CATS-IO on both Cori and Summit with a varying number of MPI processes and nodes. We use sleep time to emulate the clustering algorithm's processing time. Again, we used 60 seconds of sleep time on Cori and 20 seconds of that on Summit between the subsequent reading of data. This time is less than the real BD-CATS DBScan processing time [9].

In contrast to the VPIC-IO results, asynchronous I/O in the implicit mode performs the worst, even worse than the synchronous mode. Asynchronous I/O in the explicit mode offers the best performance, which is 4.9X faster than reading the data in the synchronous mode. The slowness of the implicit mode is because the read operations in this mode default to the synchronous mode to maintain data consistency. Without switching to the synchronous mode, the non-blocking read function calls would return immediately without filling the user's buffer with data, while the application assumes the data is ready and starts its procedures to operate on the data. That will result in errors and reading incorrect data. Therefore the implicit mode is slightly slower than the synchronous HDF5 mode because of the added overhead in the asynchronous task creation and management. On the other hand, with the explicit mode, we are able to enable asynchronous read operations to their full potential, effectively reducing the observed I/O time.

4.3 Nyx

We configured the Nyx workload to write a fixed amount of data with a different number of MPI processes and nodes. It writes 5 timesteps with a sufficient amount of sleep time in between to fully overlap the I/O and is much less than the computation phases in actual simulation runs [14]. As opposed to VPIC-IO, each time a checkpoint is written, Nyx creates a new file.

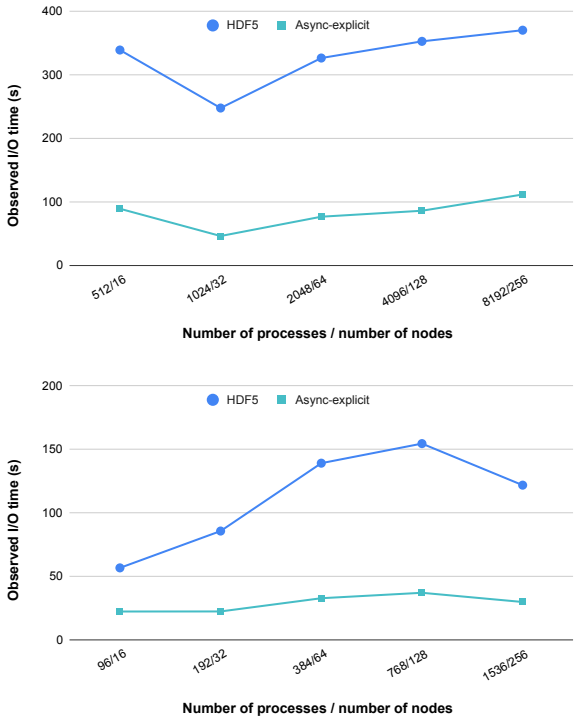


Fig. 7: Nyx workload write performance comparison on Cori (top) and Summit (bottom) between synchronous HDF5 and HDF5 with explicit asynchronous I/O mode. 5 timesteps each with a single AMR level data are written with sufficient compute/sleep time in between. Observed I/O time includes the last timestep's write time and the overhead of the asynchronous I/O framework for all timesteps.

As mentioned previously, the implicit mode uses file close as a synchronization point, which would result in effectively synchronous I/O and thus we only compare the explicit mode against the baseline synchronous HDF5 in Figure 7. The workload requires much more memory than VPIC-IO as additional data structures are maintained by the AMReX framework, so we use 16 nodes on both Cori and Summit as the smallest scale and increase the number to 256 for the largest scale.

The results from both Cori and Summit show up to 4.5X I/O time speedup when using our asynchronous I/O framework, and while the performance of synchronous I/O fluctuates with different numbers of processes, it is much more stable with the asynchronous approach as most of the I/O time is completely hidden.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2021.3090322, IEEE Transactions on Parallel and Distributed Systems

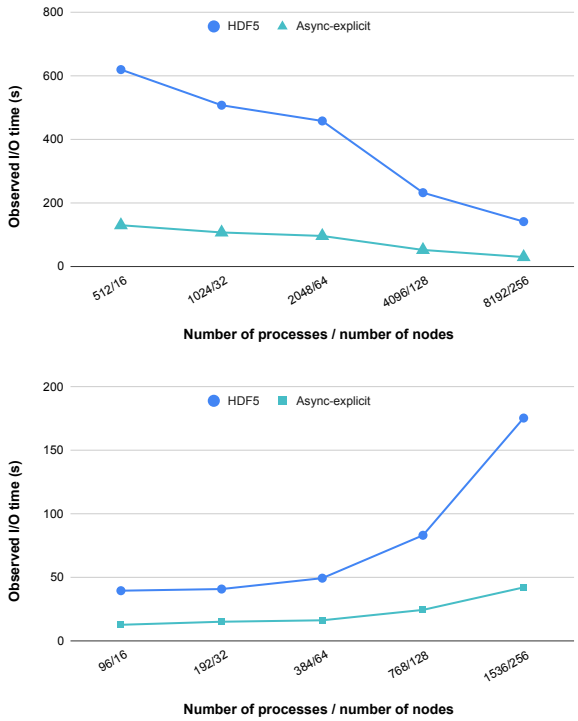


Fig. 8: Castro workload write performance comparison on Cori (top) and Summit (bottom) between synchronous HDF5 and HDF5 with explicit asynchronous mode. 5 timesteps each with 3 AMR levels data are written with sufficient compute/sleep time in between. Observed I/O time includes the last timestep's write time and the overhead of the asynchronous I/O framework for all timesteps.

4.4 Castro

The Castro workload has more complex data structures in the HDF5 output file than the Nyx workload. Castro uses three mesh refinement levels, each with rich metadata such as the refinement ratio, problem domain of the level, number of ghost cells, etc. We also configured Castro to write 5 timesteps and use enough sleep time between them to fully overlap with the asynchronous I/O.

Due to an issue with Summit's Spectrum MPI, we were only able to run the Castro workload with independent MPI I/O (without collective buffering) on Summit, while using the default collective buffering on Cori. This leads to different performance trends when increasing the number of processes on the two supercomputers, as shown in Figure 8. On Summit, as the number of processes increase, the observed I/O time increased due to more MPI processes accessing the file system independently causing higher overhead on the server. On the positive side, we still observe up to 4.7X performance improvement with the asynchronous I/O both on Cori and on Summit.

4.5 Discussion

We have designed and developed an asynchronous I/O framework using background threads and tested it with two I/O kernels and two real cosmology simulation configurations. The comparison among the baseline HDF5 with synchronous I/O, implicit asynchronous I/O, and explicit asynchronous I/O demonstrates the effectiveness of our work: the implicit asynchronous I/O mode requires the least amount of code changes, and is ideal for a write-only application that operates on a single file. In this write pattern, the metadata operations such as file close operations that default to synchronous I/O operations do not impact the performance of implicit asynchronous I/O.

The explicit asynchronous I/O mode requires replacing the I/O function calls with corresponding asynchronous versions and an additional EventSet ID parameter, but it allows all I/O operations to be asynchronous (including file close) and provides convenient methods to check multiple asynchronous operations' status and retrieve the detailed error information at runtime. Our results show that for all the experiments, the explicit mode can hide the I/O time efficiently when there is sufficient computation time to overlap with, leaving only the overhead of the first timestep of read and the last timestep of write observed by the application. This demonstrates that the explicit asynchronous I/O mode in HDF5 greatly improves the overall application performance.

5 Related Work

The size of data stored and accessed by scientific applications is ever increasing as we are moving toward exascale computing. However, I/O hardware has not seen improvements that keep up with the explosion of data to access. Parallel file systems such as Lustre [17], PVFS [18], GPFS [19], and NFS [20] are designed to handle common I/O access patterns in HPC applications, but still suffer significant performance drops when the data is accessed poorly or with a large number of small I/O operations.

Asynchronous I/O is an increasingly popular option for improving I/O performance when handling the large volume of data operations required by today's applications. Applications that perform I/O and computation or communication periodically can take advantage of asynchronous I/O operations by scheduling I/O as early as possible and checking the operations' status later. This allows overlapping I/O operations with the application's communication and computation, hiding some or all of the cost associated with the I/O.

POSIX [1] introduced asynchronous I/O (AIO) to enable performing I/O operations alongside computation operations [21]. Operating systems, such as Linux, provide such support [22] with "`aio_*`" functions, and allow writing and reading data asynchronously to and from the underlying file system. Lazy AIO (LAIO) [23] is proposed for converting any I/O system call into an asynchronous call. However, these low-level I/O calls require user involvement in managing data dependencies. There have also been asynchronous I/O efforts at the file system level. For instance the Light-Weight File System (LWFS) [24] proposes asynchronous I/O support at the file system level. However, to use asynchronous I/O in LWFS, the entire file system has to be replaced, which is impractical on production-class supercomputing facilities that typically use Lustre, GPFS, etc., and support thousands of users.

I/O overlapping strategies have also been proposed for parallel I/O libraries. The impact of various overlapping strategies of MPI-IO have been studied in [25], and [26]. Unfortunately, these studies were either performed at a small scale or are application-specific. High-level I/O libraries, such as ADIOS [27], [28] provide asynchronous I/O support using a staging interface, where data is transferred to staging servers' memory with the DataSpaces [29] transport method before writing to the storage system. DataElevator [3] uses a similar strategy of writing to a burst buffer file system and moving the data to long-term capacity storage asynchronously. [30] proposed a buffering scheme that can output data asynchronously for collective MPI-IO operations, but

is limited to write-only applications. Damaris [31] proposed to use additional cores or dedicated nodes to perform asynchronous data processing, I/O, and in situ visualization. Proactive Data Containers (PDC) [4] is a user-level data management system with servers for object abstractions and performs asynchronous I/O.

I/O libraries and data management systems, such as ADIOS [27], Data Elevator [3], Damaris [31], and PDC [4], provide asynchronous I/O by using extra (server) processes. These processes stage/cache data transferred from the client processes through network (when client and server are on different compute nodes) or shared memory (when on the same node). Comparing with our background thread approach, which does not require launching and maintaining extra server processes, and the data is only copied in the process's memory and never transferred to other compute nodes, these approaches involve more manual control and often have higher overhead. While these systems provide more data management services, such as metadata management [32], in situ analysis, etc., performing asynchronous I/O in HDF5 does not require addition CPU resources. Damaris and PDC also require the user to replace the existing application's I/O related functions with their provided APIs and use a new data storage format, while our proposed approach only requires a minor API change (adding `_async` to the function name and the EventSet ID to the function parameters) and preserving the already in-use HDF5 file format.

The asynchronous I/O framework in this paper expands our previous work [10], adding the explicit mode with EventSet APIs, error checking and information retrieval for failed tasks, as well as the experiments on the Summit supercomputers with additional scientific application workloads. We have demonstrated that the new developments with an explicit asynchronous I/O mode prove to be highly efficient at hiding I/O latency.

6 Conclusions and Future Work

We propose an asynchronous I/O framework implemented with background threads that supports all I/O operations and can effectively reduce an application's observed I/O time. Our framework manages the asynchronous I/O tasks automatically and transparently, with rule-based dependency tracking and dynamic operation maintenance. Our implementation as an HDF5 VOL connector provides two ways for applications to use our framework: an implicit mode for minimal code change requirements but with limited control of asynchronous tasks, and an explicit mode that uses an EventSet API to manage asynchronous tasks with more capability but with modest modification to application code. We demonstrate the performance improvements by comparing standard synchronous HDF5 with the asynchronous approaches using I/O kernels and real scientific workloads, and show that the majority of the I/O cost can be hidden from the application.

Our future work includes exploring new optimization techniques for asynchronous I/O, such as reordering or merging tasks in the asynchronous task queue and dynamically setting the ideal parallel file system tuning parameters (e.g., Lustre stripe size and count) based on the queued operations before starting to execute asynchronous I/O tasks. We will also explore additional optimizations by engaging with more scientific applications – especially from the Exascale Computing Project (ECP), as well as other I/O libraries and frameworks. We are also planning to make the asynchronous I/O feature available to Python-based libraries that use HDF5, such as h5py, which will broaden the impact of this 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 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. This research used resources of the Oak Ridge Leadership Computing Facility at the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725.

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] B. Dong, S. Byna, K. Wu, Prabhat, H. Johansen, J. N. Johnson, and N. Keen, "Data Elevator: Low-Contention Data Movement in Hierarchical Storage System," in *HiPC*, 2016, pp. 152–161.

[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] H. Meuer, E. Strohmaier, J. Dongarra, and H. Simon, "Top500 supercomputer sites." [Online]. Available: https://www.top500.org

[7] S. Byna, M. 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, pp. 145–160, 2020.

[8] S. Byna, J. Chou, O. Rübel, Prabhat, H. Karimabadi *et al.*, "Parallel I/O, Analysis, and Visualization of a Trillion Particle Simulation," in *Supercomputing*, 2012, pp. 59:1–59:12.

[9] 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.

[10] 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.

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

[12] 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.

[13] 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.

[14] 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, p. 39, Mar. 2013.

[15] W. Zhang, A. Almgren, V. Beckner, J. Bell, J. Blaschke, C. Chan, M. Day, B. Friesen, K. Gott, D. Graves, M. Katz, A. Myers, T. Nguyen, A. Nonaka, M. Rosso, S. Williams, and M. Zingale, "AMReX: a Framework for Block-structured Adaptive Mesh Refinement," *Journal of Open Source Software*, vol. 4, no. 37, p. 1370, May 2019. [Online]. Available: https://doi.org/10.21105/joss.01370

[16] A. Almgren, M. B. Sazo, J. Bell, A. Harpole, M. Katz, J. Sexton, D. Willcox, W. Zhang, and M. Zingale, "CASTRO: A Massively Parallel Compressible Astrophysics Simulation Code," *Journal of Open Source Software*, vol. 5, no. 54, p. 2513, 2020. [Online]. Available: https://doi.org/10.21105/joss.02513

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

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

[19] 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.

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

[21] D. McCall, "Asynchronous I/O and event notification on Linux," http://davmac.org/davpage/linux/async-io.html.

[22] S. Bhattacharya, S. Pratt, B. Pulavarty, and J. Morgan, "Asynchronous I/O Support in Linux 2.5," 2003.

[23] 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.

[24] 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.

[25] 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.

[26] S. Zhou, A. Oloso, M. Damon, and T. Clune, "Application Controlled Parallel Asynchronous IO," in *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, ser. SC '06, 2006, p. 178–es. [Online]. Available: https://doi.org/10.1145/1188455.1188639

[27] 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.

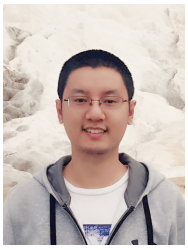
[28] J. F. Lofstead, S. Klasky, K. Schwan, N. Podhorszki, and C. Jin, "Flexible io and integration for scientific codes through the adaptable io system (adios)," in *CLADE*. New York, NY, USA: ACM, 2008, pp. 15–24.

[29] C. Docan, M. Parashar, and S. Klasky, "DataSpaces: An Interaction and Coordination Framework for Coupled Simulation Workflows," in *Cluster Computing*, vol. 15, 01 2010, pp. 25–36.

[30] X. Ma, J. Lee, and M. Winslett, "High-level buffering for hiding periodic output cost in scientific simulations," *IEEE Transactions on Parallel and Distributed Systems*, vol. 17, no. 3, pp. 193–204, 2006.

[31] M. Dorier, G. Antoniu, F. Cappello, M. Snir, R. Sisneros, O. Yildiz, S. Ibrahim, T. Peterka, and L. Orf, "Damaris: Addressing performance variability in data management for post-petascale simulations," *ACM Transactions on Parallel Computing (TOPC)*, vol. 3, no. 3, pp. 1–43, 2016.

[32] H. Tang, S. Byna, B. Dong, J. Liu, and Q. Koziol, "SoMeta: Scalable Object-centric Metadata Management for High Performance Computing," in *CLUSTER*, 2017, pp. 359–369.



Houjun Tang received his Ph.D. in Computer Science from NC State University in 2016, and B.Eng. in Computer Science and Technology from Shenzhen University, China in 2012. He is a Computer Research Scientist at Lawrence Berkeley National Laboratory. His research interests include data management, parallel I/O, and storage systems in HPC.

Quincey Koziol received a B.S in Electrical Engineering from the University of Illinois. He is a principal data architect at Lawrence Berkeley National Laboratory where he drives scientific data architecture discussions and participates in NERSC system design activities. He is the principal architect for the HDF5 project and a founding member of the HDF Group.

John Ravi received his B.S. degree in 2018 and M.S. degree in 2019 in Computer Engineering from NC State University. He is currently pursuing a Ph.D. in Computer Engineering from his alma mater. His research focus is improving GPU utilization in HPC environments.

Suren Byna received his Ph.D. degree in 2006 in Computer Science from Illinois Institute of Technology, Chicago. He is a Staff Scientist in the Scientific Data Management (SDM) Group in CRD at Lawrence Berkeley National Laboratory (LBNL). He works on optimizing parallel I/O and on developing systems for managing scientific data. He leads the ECP funded ExaIO project that is developing features in HDF5 and UnifyFS, and various projects on managing scientific data.