Evaluating Asynchronous Parallel I/O on HPC Systems

John Ravi
North Carolina State University
jjravi@ncsu.edu

Suren Byna
Lawrence Berkeley National Laboratory
sbyna@lbl.gov

Quincey Koziol
Lawrence Berkeley National Laboratory
koziol@lbl.gov

Houjun Tang
Lawrence Berkeley National Laboratory
htang4@lbl.gov

Michela Becchi
North Carolina State University
mbecchi@ncsu.edu

***Abstract*—Parallel I/O is an effective method to optimize data movement between memory and storage for many scientific applications. Poor performance of traditional disk-based file systems has led to the design of I/O libraries which take advantage of faster memory layers, such as on-node memory, present in high-performance computing (HPC) systems. By allowing caching and prefetching of data for applications alternating computation and I/O phases, a faster memory layer also provides opportunities for hiding the latency of I/O phases by overlapping them with computation phases, a technique called *asynchronous I/O*. Since asynchronous parallel I/O in HPC systems is still in the initial stages of development, there hasn't been a systematic study of the factors affecting its performance.**

In this paper, we perform a systematic study of various factors affecting the performance and efficacy of asynchronous I/O, we develop a performance model to estimate the aggregate I/O bandwidth achievable by iterative applications using synchronous and asynchronous I/O based on past observations, and we evaluate the performance of the recently developed asynchronous I/O feature of a parallel I/O library (HDF5) using benchmarks and real-world science applications. Our study covers parallel file systems on two large-scale HPC systems: Summit and Cori, the former with a GPFS storage and the latter with a Lustre parallel file system.

***Index Terms*—Performance Evaluation, Modeling, Asynchronous I/O, Parallel I/O**

I. Introduction

Many HPC simulations, machine learning, and deep learning applications alternate computation and I/O phases. In order to take advantage of increasingly powerful HPC systems, scientists can scale their simulations in three ways: (1) by targeting more complex problems with high computational demands, (2) by increasing the number of time steps or duration of a computation and I/O phase, and (3) by running the simulation at finer resolution (space or time), which will increase the number of time steps. Several research efforts have focused on the scaling capabilities of computation phases [1], [2]. Scaling the computation performed by an application often leads to scaling its working memory size, thus making the data movements and the resulting I/O overhead more apparent. If the latency of I/O phases in an application is too high, it can become a bottleneck and lead to poor utilization of compute resources.

Recent and upcoming HPC architectures are adding fast solid state storage layers to reduce the I/O latency, but data still have to be moved between long-term storage and memory. For instance, the Summit system at Oak Ridge Leadership Computing Facility (OLCF) contains compute nodes with an SSD for faster access and a shared parallel file system (GPFS). Upcoming exascale computing architectures are expected to contain a fast node-local storage layer, a high performance storage layer, and a high capacity storage layer. With GPUs available on these systems and the need to move data across heterogeneous memory locations, optimizing data movement and parallel I/O becomes even more complicated. Optimizing data transfers within applications can improve performance at the cost of programmability and portability. To improve time-to-solution, I/O libraries, such as HDF5 [3] and ADIOS [4], allow researchers and developers to implement optimizations transparently to the application. Furthermore, asynchronous I/O implementations have been gaining prominence among these I/O libraries, such as HDF5 Async I/O VOL [5] and data management systems, such as PDC [6]. Asynchronous I/O support in these parallel I/O libraries allows overlapping of the I/O phases with the computation phases of the application.

However, there hasn't been a systematic study of benefits and limitations of asynchronous I/O. When computation phases are short, i.e., an I/O phase takes more time than the computation phase, asynchronous I/O might not lead to performance benefits. Moreover, asynchronous I/O requires some buffer space in order to avoid synchronization issues. When the data transfers required to initialize those buffers and set up asynchronous I/O take longer than the following computation phase, the overhead of asynchronous I/O might lead to performance degradation over performing I/O synchronously.

In order to gain a better understanding of asynchronous I/O, we identify and study several factors affecting its performance. These include: the amount of data to be transferred via I/O, the number of MPI ranks and nodes involved in I/O, and various costs (computation time, I/O time, data transfer setup costs, etc.). We use this study to develop a cost model for estimating the performance benefits of asynchronous I/O over synchronous I/O. We validate this cost model using a set of I/O kernels that are representative of plasma physics simulations and a big data clustering application. We then show the benefits of asynchronous I/O on four real science applications with different input parameters.

To the best of our knowledge, an evaluation and model of asynchronous I/O at scale is missing. In summary, this paper makes the following contributions:

1) An experimental evaluation of the aggregate I/O rate achievable by HDF5 without and with asynchronous I/O on two large scale systems: Summit and Cori. We run our experiments up to 2k nodes on Summit and identify when synchronous I/O becomes a bottleneck.
2) An analytical model for estimating performance of synchronous and asynchronous I/O based on computation time, data movement time, and setup costs.
3) A model-based approach to evaluate efficacy of both I/O modes using two I/O kernels and four large-scale science applications.

II. Background and Motivation

A. Hierarchical Data Format version 5 (HDF5)

HDF5 is a popular I/O library and self-describing file format that provides an abstraction layer to manage data and the metadata within a single file [3]. HDF5 is used heavily in various science domains to manage a wide variety of data models and is used for efficient parallel I/O in HPC simulations and machine learning analyses [7]. HDF5 has recently provided a feature, called the Virtual Object Layer (VOL) [8], to enable HDF5 to support dynamic control to the library at runtime. VOL allows intercepting the high-level HDF5 public application programming interface (API) and implementing various optimizations for different types of storage medium, thus enabling better data management transparently to the application. 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.

A team of researchers have used the HDF5 VOL feature to implement an asynchronous I/O VOL connector [5] that enables asynchronous I/O for HDF5 operations using background threads. This implementation can be compiled as a dynamically linked library (DLL) and linked to a user's application directly, remaining separate from the installed version of HDF5 and making it easy to adopt. The background threads are managed by Argobots, a lightweight low-level threading framework [9]). More implementation details of the VOL connector are available in [5]. Since HDF5 is heavily used by science applications, in this study we use it to evaluate four large-scale applications utilizing this HDF5 VOL connector that is available publicly[1]. The VOL feature also allows us to incorporate our modeling strategy, proposed in Section III, which relies on runtime tracking of I/O calls to determine if asynchronous I/O is beneficial over synchronous I/O.

B. Asynchronous I/O Challenges

Asynchronous I/O has the potential for improving application performance by overlapping computation and I/O transfers, and can be particularly beneficial for iterative applications alternating computation and I/O phases. The benefits of asynchronous I/O depend on application and system characteristics (number and duration of computation and I/O phases, data transfer setup costs, etc.). Existing data management libraries and recent programming languages provide the necessary tools to implement asynchronous I/O on these systems; however, profiling and identifying the effectiveness of such methods has become difficult due to application and system complexity. All these challenges motivate the need to provide a transparent and adaptive asynchronous I/O interface to automatically enable asynchronous I/O when needed without placing the burden on application developers. A first step towards this overarching goal is obtaining a thorough understanding of asynchronous data movement. In this paper, we aim to answer some of these challenges by introducing a performance model to decide whether an I/O phase would benefit from asynchronous I/O.

C. Related Work

I/O latency has been a performance bottleneck for several applications due to slow disk performance. Parallel file systems, such as Lustre, PVFS, GPFS, and NFS, aim to provide efficient concurrent access to disk-based parallel storage. These file systems still require users to perform significant tuning to obtain superior performance. With faster storage layers between main memory and disk-based file systems, there are several asynchronous I/O efforts to hide the I/O latency by caching or prefetching data during other phases of application, typically computation. These efforts can be classified into various layers, including operating systems or file systems, I/O middleware, high-level I/O libraries, and other data management layers. We briefly describe these efforts here and motivate this paper's contributions to the existing work.

Among the most prominent efforts of asynchronous I/O, POSIX [10] introduced asynchronous I/O (AIO) for a process to perform I/O operations alongside computation operations [11]. These "`aio_*`" functions are available for reading and writing data asynchronously to the underlying file system. Operating systems, such as Linux provide POSIX AIO support [12]. Elmeleegy et al. [13] proposed Lazy AIO (LAIO) routines for converting any I/O system call into an asynchronous call. All these low-level I/O calls require user involvement in managing dependencies. There have been some asynchronous I/O efforts at the file system level. For instance the Light-weight file system (LWFS) [14] proposes asynchronous I/O support at the file system level. However, to use the asynchronous I/O entire file system has to be replaced with LWFS, which is impractical on production-class supercomputing centers that typically use Lustre, GPFS, etc. that support thousands of users.

In parallel I/O libraries, several I/O overlapping strategies have been explored. Among them Patrick et al. [15] and Zhou et al. [16] explore overlapping I/O with computations in the MPI-IO interface [17] at the MPI-IO level and at the application level, respectively. These studies were either performed at small scale or specific to applications. High-level libraries, such as ADIOS [4] and HDF5 [5] provide asynchronous I/O. In ADIOS, asynchronous I/O is provided using a staging interface, where data is transferred to staging

[1] HDF5 Asynchronous I/O VOL: https://github.com/hpc-io/vol-async

servers that are supported by Data Spaces [18], [19]. HDF5's asynchronous I/O [5] uses background threads for caching data either to a memory buffer on the same node where a process is running or to a node-local SSD. In both these I/O libraries, the staged or cached data is later written to long term storage asynchronously. Similarly, Multilayered Buffer System (MLBS) [20], [21] is a recently proposed specialized I/O library for demonstrating the capability of overlapping I/O with computation phases using background threads. DataElevator [22] uses similar strategy of writing to a burst buffer and move the data to a capacity storage asynchronously. Proactive Data Containers (PDC) [6] is a user-level data management system with servers for object abstractions and performs asynchronous I/O similar to ADIOS and HDF5. Recent work has focused on improving data staging techniques, such as efficiently supporting unstructured mesh transfer [23] and reducing the initialization cost for short-running jobs where it cannot be amortized over the total runtime [24].

None of these studies looked into asynchronous I/O strategies in a generic fashion or provided analytical modeling based on factors such as computation time, data size, I/O time, and overhead of setting asynchronous mechanism. Previous efforts that have focused on modeling I/O performance on HPC systems [25]–[30] target prediction of I/O time using characteristics of application I/O patterns (e.g., contiguous and strided accesses) and performance tuning parameters (e.g., number of MPI-IO aggregators, file system striping). None of them, however, include asynchronous I/O in their models and analysis. Although the libraries described above enable data movement overlapping, they do not motivate the need to switch between synchronous I/O and asynchronous I/O modes automatically. Our work motivates the need to have a runtime analysis of asynchronous I/O and provides a model that can be easily integrated in existing libraries to determine which I/O mode is beneficial. Our empirical model uses a history of previous runs and a statistical approach, similar to that of Behzad et al. [28]. While Behzad et al. targets only synchronous I/O and focuses on tuning file system parameters, such as Lustre stripe count and stripe size, we focus on estimating the aggregate I/O bandwidth achievable by applications with both synchronous and asynchronous I/O. Our analysis also covers file systems where system parameters (such as stripe size and stripe count) are automatically configured, such as GPFS on Summit and rely on best practices for file system parameters based on previous work. Our goal for empirically estimating the aggregate I/O bandwidth is to decide whether asynchronous I/O will be beneficial over the ideal observed synchronous I/O.

III. Methodology

Asynchronous I/O operations at a high-level can be broken into read and write I/O. Read operations can be made asynchronous by incorporating prefetching strategies in an application, while the latency of write operations can be hidden by employing effective caching strategies. However, these asynchronous I/O methods may have negative impact on performance for some applications and datasets. For iterative applications, asynchronous I/O might even benefit some iterations of the application and not others.

In order to understand potential benefits for applications, it is necessary to quantify the effectiveness of asynchronous I/O. In this section we propose a performance model to estimate the performance benefits and overheads of asynchronous I/O methods. With our methodology, we try to target a subset of applications that we believe are important in the next generation of HPC. We note that previous work, such as [31], have recognized common I/O trends in HPC workloads. Our model applies to iterative applications, with each iteration alternating computation and I/O phases. Many HPC workloads follow this pattern, including the applications evaluated in this paper. We consider various costs included in asynchronous I/O operations, i.e., *initialization cost*, *data transfer time*, *computation time*, and *number of iterations*. To reiterate, the proposed performance model has two objectives: estimating the effectiveness of asynchronous I/O on future iterations based on performance observed in previous iterations of the application, and estimating scalability. In particular, we are interested both in strong scaling and weak scaling behaviors of asynchronous I/O (i.e., scalability with the number of nodes and the data size).

A. Iterative-based IO Performance Model

To model asynchronous I/O, we start by breaking down the total execution time of an iterative application into the sum of the execution time of all epochs plus the initialization and termination times (Eq. 1). During initialization, asynchronous I/O methods allocate memory buffers for caching, set up communication channels, initialize background threads for performing I/O, and open file descriptors. The time to complete each iteration is characterized by t_{epoch}. The total number of iterations can vary based on the application and be input-dependent due to convergence requirements. The termination of asynchronous I/O method involves freeing any memory buffers and finalizing background threads, etc. The initialization and termination costs (t_{init} and t_{term}, respectively) are typically small and, based on our experiments, relatively constant as we scale the number of available nodes. The proposed performance model has two objectives: estimating the effectiveness of asynchronous I/O on future iterations based on performance observed in previous iterations of the application, and estimating scalability. We will now focus on estimating each epoch time.

$$t_{app} = t_{init} + \sum_{i=0}^{iters.} t_{epoch} + t_{term} \qquad (1)$$

We assume that each epoch is composed of a computation phase (t_{comp}) and an I/O phase (t_{io}). This is typical for large-scale scientific simulations that converge on a solution by iterating in discrete time steps configured by the user. The computation time in our model includes any communication time and the synchronization time among parallel processes. An I/O phase in our model includes all data transfers that are involved with I/O operations (such as copying from GPU memory to CPU memory before persisting to storage). In Eq. 2a, we show a synchronous epoch time (t_{sync_epoch}), which is composed of separate I/O and computation phases, where computation is stalled during the I/O phase. In Eq. 2b, we show an asynchronous epoch time (t_{async_epoch}), where the I/O phase either partially or fully overlaps with the computation phase. With the *max* operator in Eq. 2b, we either account for only the computation phase time if it can fully overlap with the I/O phase or the remaining I/O phase time if it is partially overlapped. Furthermore, an asynchronous epoch also introduces an additional data transfer, which we denote as $t_{transact\_overhead}$. This additional data transfer is a non-zero-copy, usually between fast volatile memory and slower non-volatile storage, which is often used in asynchronous I/O implementations to eliminate data races between the main application thread and background I/O threads.

$$t_{sync\_epoch} = t_{io} + t_{comp} \tag{2a}$$

$$t_{async\_epoch} = max(t_{comp}, t_{io} - t_{comp}) + t_{transact\_overhead} \tag{2b}$$

In Fig. 1 we show various scenarios comparing a synchronous epoch and an asynchronous epoch. Each scenario has computation phases, I/O phases and transactional overheads to set up the asynchronous data transfers. With asynchronous I/O, the I/O cost changes from fully blocking (i.e., synchronous) to partially blocking because of the transactional overhead. If the computation phase is longer than the I/O phase, the I/O latency can be completely overlapped, as seen in Fig. 1a. If the computation phase is shorter than the I/O phase, it may still achieve some overlap, as seen in Fig. 1b. In both these scenarios, an asynchronous epoch also includes the transactional overhead, which we denote as "Overhead" in the figures. This overhead captures the data transfer to an extra memory buffer to avoid data races or modification by subsequent computation phase for an asynchronous I/O method. Even though the transactional overhead may be small, it makes the transparent asynchronous I/O method not always beneficial. This is shown in Eq. 2b, where the computation is assumed to overlap with the data transfer from the previous iteration. When the $t_{comp} \leq t_{transact\_overhead}$, utilizing asynchronous I/O will result in a slowdown because no amount of overlap will amortize the cost of introduced transactional overhead, as shown in Fig. 1c.

B. Empirical Modeling of Epoch Time

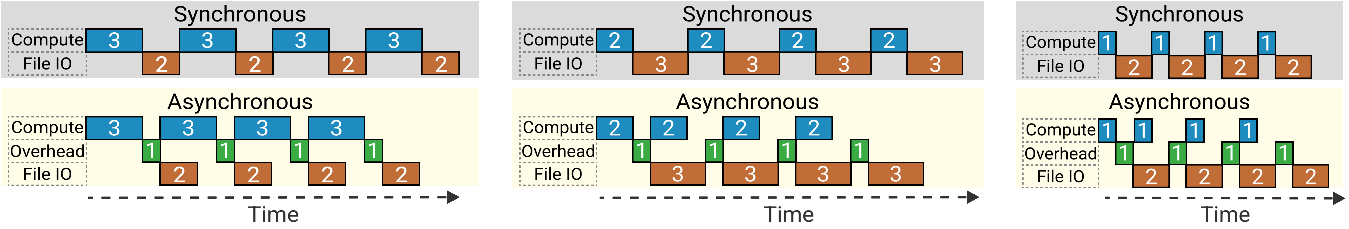
To estimate the different costs mentioned in Eq. 2, we use an empirical modeling approach based on real measurements from two supercomputers: Summit at the Oak Ridge Leadership Computing Facility (OLCF) and Cori at the National Energy Research Scientific Computing Center (NERSC). Refer to Section IV-A for more information about these HPC systems. We use this empirical approach because the I/O costs vary from system to system depending on the hardware configuration and file system characteristics. Recall that our model requires estimating three costs: compute time (t_{comp}), transactional overhead ($t_{transact\_overhead}$), and I/O time (t_{io}).

In Fig. 2, we summarize the process of estimating using a history of time measurements and a statistical method for each cost, and progressively adding new measurements to the history for improving the accuracy of the model. Measurements collected while running applications are also added to the history. We measure the computation time directly in the application and use a weighted average over the measurements taken in previous iterations to estimate the computation time of the next iteration. This simple approach of estimating the computation phase can be replaced with advanced models [1], [2]. In this paper, we focus our estimation to transactional overhead and I/O times.

The transactional overhead and I/O rate are dependent on the hardware used in the compute nodes, communication interconnects, network between compute and storage subsystems, and the parallel file system. Moreover, the number of jobs running on the systems using the shared storage system simultaneously can also affect the observed I/O rate. We define the I/O rate as the ratio of data size and the I/O time. In this subsection, we briefly describe the method we use to estimate the transactional overhead and the I/O rate.

1) Estimating transactional overhead: The transactional overhead poses a challenge in using asynchronous I/O, especially when data or computation scaling is involved. In the strong scaling case, data size is fixed while compute resources are increased. This implies that the overall read and write time for an application will decrease until there is some bottleneck, such as interconnect latency; moreover, the compute time will also decrease. If the on-node DRAM is used for caching, then the cache time is expected to be relatively constant during the execution of the application. This implies that for strong scaling applications at some point the performance of transparent asynchronous I/O will be worse. For weak scaling applications, data size scales with the available resources, which implies that the duration of computation and I/O phases will stay relatively constant given adequate resources. However, the synchronous I/O phase commonly becomes a bottleneck due to interconnect bandwidth. For both weak and strong scaling applications, to determine if asynchronous I/O is beneficial we need to track how much transactional overhead is added and if enough overlapping potential exists to hide the added overhead. We estimate the transactional overhead by measuring data copy costs between different memory buffers, which, depending on the application, can be located either on CPU or on GPU.

For CPU applications that enable transparent asynchronous I/O, the transactional overhead is the cost of a `memcpy` between two CPU memory buffers. We measured the bandwidth of a `memcpy` transfer with varying sizes of data on a single node on both systems using a micro-benchmark. We found the `memcpy` bandwidth to be constant after $32MB$, so we expect a constant bandwidth when we calculate the transactional overhead due to CPU memory-to-memory copy for I/O requests greater than 32 MB. The transactional overhead for smaller I/O requests will depend on the data size. In the considered applications the data sizes are large enough to amortize the data transfer.



(a) **Ideal scenario** of asynchronous I/O, where computation phase is longer than the I/O phase yielding complete overlap of I/O latency.

(b) **Partial overlap scenario**, where computation phases are shorter in time than the I/O phases that may result in a partial overlap of I/O latency.

(c) **Slowdown scenario**, where the introduced transactional overhead is longer than the feasible computation and I/O overlap.

Fig. 1: Each timeline shows a scenario for overlapping a computation phase with I/O phase, synchronous on top and asynchronous on bottom.

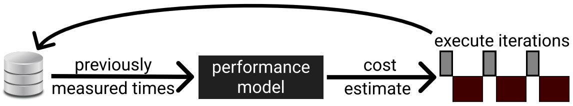


Fig. 2: A model feedback loop added to a high-level I/O library to estimate costs associated with asynchronous I/O.

For GPU-accelerated applications, the I/O transactional overhead includes the blocking memory transfer time between the CPU memory and GPU memory. On some systems the GPUs are connected to the CPUs using PCI-E 3.0 connections which have a theoretical upper limit of $15.75GB/s$ bandwidth. The interconnect on Summit, NVLink 2.0, has a theoretical upper limit of $50GB/s$. However, there is some overhead associated with copying data between the GPU and CPU such as setting up a direct memory access (DMA) controller. Since the DMA controller requires the memory pages to not be swapped out during transfer, the runtime will incur additional overhead for creating a transaction copy when not pinning the host memory pages. In our micro-benchmarks, the memory copy cost is amortized for data sizes greater than $10MB$, and we found that with pinned host memory the peak bandwidth is close to the theoretical maximum.

2) Estimating the I/O time: In the I/O path, the data might have to be copied between multiple storage systems before being ready or resident on the target storage location. In the synchronous I/O case, each of these data movements will result in a blocking I/O call. The full cost of I/O will thus be the sum of all the data transfers. With parallel I/O, since all the nodes have to synchronize after their respective data transfers, the MPI process taking the longest time determines the I/O time for that iteration. Each data transfer time is based on the size of the data being transferred divided by the I/O rate of that data transfer (Eq. 3). In the case of strong and weak scaling scenarios, the size of the data plays a critical role in estimating the I/O cost. We assume that any start up costs involved in setting up the data transfer are constant and our model captures that as an t_{init} in Eq. 1.

$$t_{io} = \frac{data\_size}{f_{io\_rate}} \tag{3}$$

As noted in Section II-C, previous studies exist in estimating I/O performance on HPC systems. While these studies focused on predicting I/O performance based on application I/O patterns and various tuning parameters, our goal is to estimate the overall cost for executing an I/O request (read or write) from a high-level library (i.e., HDF5 in our case) without considering the entire application's or overall system's I/O. We use a statistical model which relies on the number of MPI ranks and data size of a read or a write request, i.e., HDF5 API calls `H5Dread` and `H5Dwrite`.

The data movement time is dependent on how much bandwidth a storage system and interconnect is able to support. We estimate the I/O rate based on a history of I/O requests by an application. For each I/O request, we record the data size, number of MPI ranks, and aggregate I/O rate. Based on the measurements, we apply statistical methods similar to [28]) to fit the I/O performance with varying number of MPI ranks and data size to extract a model that estimates the I/O cost; however, instead of using nonlinear regression methods, we apply linear regression and linear-log regression to estimate model parameters analytically. We found linear regression to be sufficient given the accuracy of our model. We determined non-linear methods were not necessary. With this approach, we are able to model both weak scaling and strong scaling performance. Recall, with weak scaling we increase the problem size while scaling available resources, while with strong scaling we keep the problem size constant while scaling available resources.

Using Eq. 4, we fit a linear regression to estimate how the I/O rate scales as we increase the number of nodes and the dataset size. For synchronous I/O, Y denotes a Nx1 matrix that contains measured aggregate I/O rate to the parallel file system for N past data transfers. For asynchronous I/O, Y denotes a Nx1 matrix that contains the measured aggregate I/O rate of a non-zero-copy to a temporary storage location which is used to calculate the transactional overhead for N past data transfers. For both modes, X denotes a Nx2 matrix that contains the data size and number of participating processes (or MPI ranks) for the previous N data requests. The i in Eq. 4 denotes the i-th row in the Y and X matrices. Each row contains information from a past data transfer.

$$\begin{aligned} f_{est\_io\_rate} \implies y_i &= \beta_0 x_{i,0} + \beta_1 x_{i,1} \\ \beta &= (X^T X)^{-1} X^T Y \end{aligned} \tag{4}$$

We utilize the coefficient of determination, denoted as r^2, to assess how strong of a linear relationship exists between the measured aggregate I/O rate (Y) and the scaling factors used in our model (X). The r^2 values range from 0 to 1 and are commonly expressed as percentages. An r^2 value above 70% indicates a strong linear correlation [32]. We calculate r^2 using Eq. 5. Using the coefficient of determination allows us to quantify how well our linear method is able to model the observed aggregate I/O rate (Y) based on the data size and the number of participating processes for all measured I/O requests (X). In our experiments (Section V), we have observed a strong linear correlation (r^2 values above 80% for synchronous I/O and 90% for asynchronous I/O) between the observed aggregate I/O rate and the considered scaling factors.

$$r^2 = \frac{Cov(X,Y)^2}{Var(X)Var(Y)} \tag{5}$$

IV. Experimental Setup

A. System Configuration

To evaluate the asynchronous I/O cost model introduced in §III-A, we performed our experiments on two systems:

Summit is a pre-exascale supercomputing system with 200 petaflops (PF) performance located at OLCF [33]. It is composed of 4,608 nodes, where each node contains two IBM 3.07 GHz POWER9 CPUs, each with 22 cores, and 6 Nvidia V100 GPUs. The entire compute system is connected to an IBM's SpectrumScale GPFS storage that has a 2.5 TB/s peak bandwidth. Each of Summit's compute nodes has a 1.6 TB NVMe SSD, which enables faster read and write access to data stored in the shared parallel file system.

Cori is a ≈30 PF Cray XC40 system located at NERSC [34]. We use the "Haswell" partition of the system that is comprised of 2,388 Intel Xeon "Haswell" processors. The compute subsystem is connected to a Lustre parallel file system that offers a peak bandwidth of 700 GB/s and an SSD-based burst buffer with a peak bandwidth of 1.7 TB/s. Based on NERSC best practices, we use 72 OSTs (stripe_large) as stripe count for all our experiments.

B. I/O Kernels

We use two I/O kernels, VPIC-IO and BD-CATS-IO, to represent large-scale simulations which commonly use a checkpoint-based approach segmented by multiple epochs, each including a computation and a I/O phase. We increase the data size for the I/O kernels as we scale up the compute nodes. Domain scientists typically configure the frequency of I/O phases, such as number of checkpoints, to increase overall throughput.

VPIC-IO [35] was extracted from a plasma physics code, Vector Particle in Cell (VPIC), that studies interactions of a magnetic re-connection phenomenon [36]. The kernel emulates writing particle data, where each particle has 8 properties and each MPI process writes (8x1024x1024) particles (≈32 MB). The number of particles increases with the number of MPI processes (weak scaling). Each property of the particles is written to a 1-D HDF5 dataset. In [36], the frequency of I/O phase is after ≈2000 time steps, i.e., 2 hours computation time between I/O phases. In our evaluations, we set the periodicity of I/O phases in VPIC-IO using a 30 second sleep in place for the computation.

BD-CATS-IO [35] is an I/O kernel similar to that of the BD-CATS algorithm, where particle data written by plasma physics [36] and astrophysics [37] are read from HDF5 files. Big Data Clustering at Trillion Particle Scale (BD-CATS) is a highly scalable version of `DBSCAN` clustering [7]. In our tests, we read the data written by the VPIC-IO kernel. This I/O kernel reads all the time steps' data, and the clustering computation was replaced with 30 seconds of sleep time.

C. Applications

In addition to I/O kernels, we evaluate three large scale real science simulations. These simulations are similar to the I/O kernels which use a checkpoint-based approach with distinct computation and I/O phase per epoch. We also evaluate a machine learning workload, Cosmoflow, to demonstrate our methodology can extend beyond just simulation and checkpoint-based workloads. For each of the full workloads, we use configurations based on publicly available data sets to demonstrate a strong scaling scenario to evaluate our model.

Nyx [37] is a massively parallel, adaptive mesh, cosmology simulation code that uses the AMReX framework [38] for computation and performing I/O. In each I/O phase, Nyx outputs a single plotfile in the HDF5 format containing information for visualizations. We run two configurations for Nyx: small and large. The former runs at 256x256x256 dimensions and writes a plotfile every 20 time steps; the latter runs at 2048x2048x2048 dimensions and writes a plotfile every 50 time steps. We set the initial I/O frequency based on our discussion with the domain scientists in real world use cases where the number of I/O phases is reduced to increase overall throughput. We do an additional experiment where we vary the duration of the compute phase to understand how partial overlap of data movement can affect overall throughput.

Castro [39] is another cosmology simulation solving compressible radiation / MHD / hydrodynamics equations for astrophysical flows using adaptive mesh resolutions. Castro also uses the AMReX framework for computation and performing I/O, which is implemented with synchronous or asynchronous I/O using HDF5. We run the Castro simulation at 128x128x128 dimensions with 6 components in each multifab and 2 particles per cell.

EQSIM [40] is an earthquake simulation framework using SW4, a 3D seismic modeling code that solves fourth order accurate wave equations. We compare synchronous and asynchronous modes of HDF5 output with SW4. We ran the simulation at grid size 50 with 30000x30000x17000 dimensions and checkpoint every 100 time steps. The simulation size does not increase as we scale up the compute resources, thus also demonstrating a strong scaling scenario.

Cosmoflow [41] is a deep learning tool to process large 3D matter distributions using convolutional neural network (CNN) and predict cosmological parameters. We used the publicly available Cosmoflow 128^3 voxels dataset. We compare synchronous and asynchronous modes of a custom PyTorch DataLoader. We run each scaling scenario for 4 epochs with batch size set to 8.

V. Evaluation

A. Evaluation of I/O Rate

We focus on the scalability of I/O performance across nodes in HPC systems to validate our model. Our results are split into two categories: I/O kernels composed of primarily I/O component of a real-world application (VPIC-IO, BD-CATS-IO) to evaluate the performance model, and four real-world large scale applications to evaluate the efficiency of asynchronous I/O at different scales. We measured the time to perform read or write operations from HDF5. The measured time of read or write operations includes the transactional overhead. We derived the I/O rate (shown as "Aggregate bandwidth" in the plots) that was observed over all the I/O phases of a benchmark or an application. We vary the number of MPI ranks (6 per node on Summit and 32 per node on Cori-Haswell) while measuring the aggregate bandwidth.

1) VPIC-IO (write): With the VPIC-IO kernel, we compare the measured I/O performance in synchronous and asynchronous modes. Using the configuration described in Section IV-B, the VPIC-IO benchmark captures write performance that varies as we increase the data size proportionally to the number of MPI Ranks (i.e., weak scaling). We include the peak measured aggregate bandwidth for all I/O phases in a bar plot in Fig. 3a and 3b. We run each configuration at least 5 times across multiple days to account for interconnect and storage system contention amongst other applications running on the system. In these figures, we compare the synchronous write performance and asynchronous write performance on Summit and Cori-Haswell systems, respectively. The asynchronous aggregate bandwidth demonstrates the ideal scenario described in Section III-A, where the compute phase is sufficiently large enough to completely overlap the I/O phase. We manually tune the length of the simulated computation phase to completely hide the asynchronous I/O time. This means the performance of the asynchronous epoch is based on the performance of the transactional overhead.

We note the aggregate bandwidth scales similarly between both systems for both synchronous and asynchronous epoch in a weak scaling scenario. The synchronous aggregate bandwidth saturates at 768 MPI Ranks (128 nodes) on Summit and 1024 MPI Ranks (32 nodes) on Cori-Haswell, while the asynchronous aggregate bandwidth scales linearly based on the constant bandwidth of the transactional overhead (see Section III-B). Our model fits well with the trend of synchronous write aggregate bandwidth which is based on a linear-log regression and shown as a dotted line in Fig. 3. The asynchronous write aggregate bandwidth scales linearly based on the constant bandwidth of the transactional overhead described in Section III-B1.

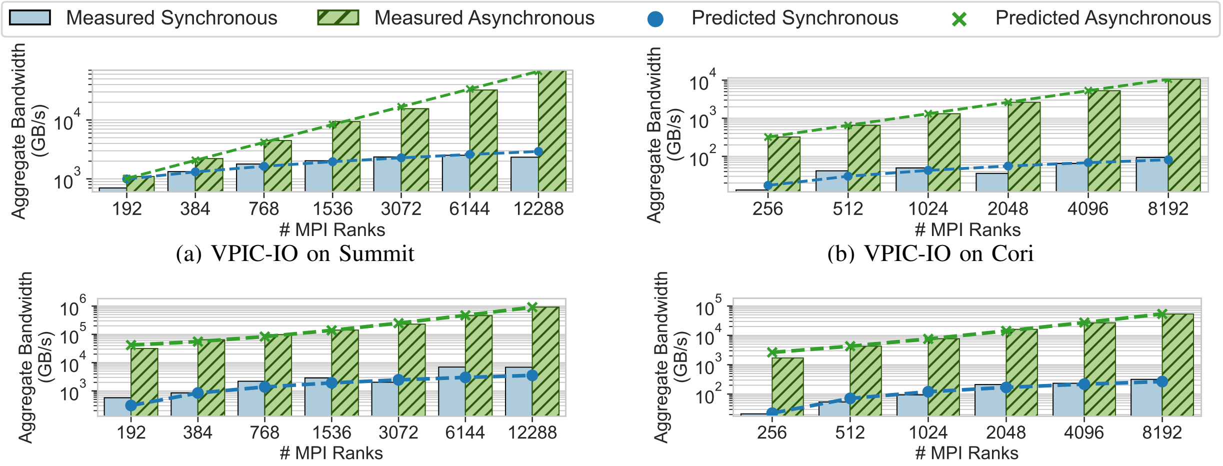
2) BD-CATS-IO (read): We run the BD-CATS-IO benchmark using the configuration described in Section IV-B. We increase the data size proportionally to the number of MPI Ranks (weak scaling). To profile performance of asynchronous read operations, we use the prefetching capabilities of the HDF5 asynchronous I/O VOL connector. In the current implementation of the VOL connector, prefetching is triggered after reading data for the first time step. The first read is a blocking operation (i.e., synchronous) since there is a dependency on the data for the first computational phase. We measured the total read time observed by the BD-CATS-IO kernel. This I/O kernel reads the data written by VPIC-IO in multiple time steps, and we include a simulated compute time in between the I/O phase to demonstrate an asynchronous epoch.

In Fig. 3c and 3d, we compare synchronous and asynchronous I/O on Summit and Cori-Haswell, respectively. The aggregate bandwidth value in GB/s is calculated, which is the ratio of the data read and the I/O phase time. As it can be observed, asynchronous I/O achieves superior performance improvement over synchronous I/O for reading data from subsequent time steps after the first time step. Since the I/O time is overlapped with a simulated computation phase on Summit and on Cori, the calculated bandwidth values for asynchronous I/O are orders of magnitude higher than those observed with synchronous I/O. The estimated values from our regression are also shown in these plots, which are matching with the measured bandwidth values well– similar to the write scenario.

3) Nyx: Since Nyx has an option to use GPUs, we run it with two separate configurations as described in Section IV. Due to space limitation, in Fig. 4a, we show the observed bandwidth of file I/O on Summit with the large configuration sizes, and in Fig. 4b, we show the performance of file I/O on Cori-Haswell with the small configuration. For each configuration, we scale the number of MPI ranks with the same dataset size (strong scaling).

On Summit, the aggregate bandwidth of synchronous I/O decreases slightly as we increase the number of MPI ranks. Similar to the trend we observe with Castro, since each MPI rank has to operate on a smaller amount of data, the overall throughput decreases. This is the opposite for the asynchronous I/O mode, since a smaller size of data will result in faster transaction time, thus reducing the overhead. With complete overlap of the computational and I/O phases, the asynchronous I/O performance scales up linearly with the number of MPI ranks.

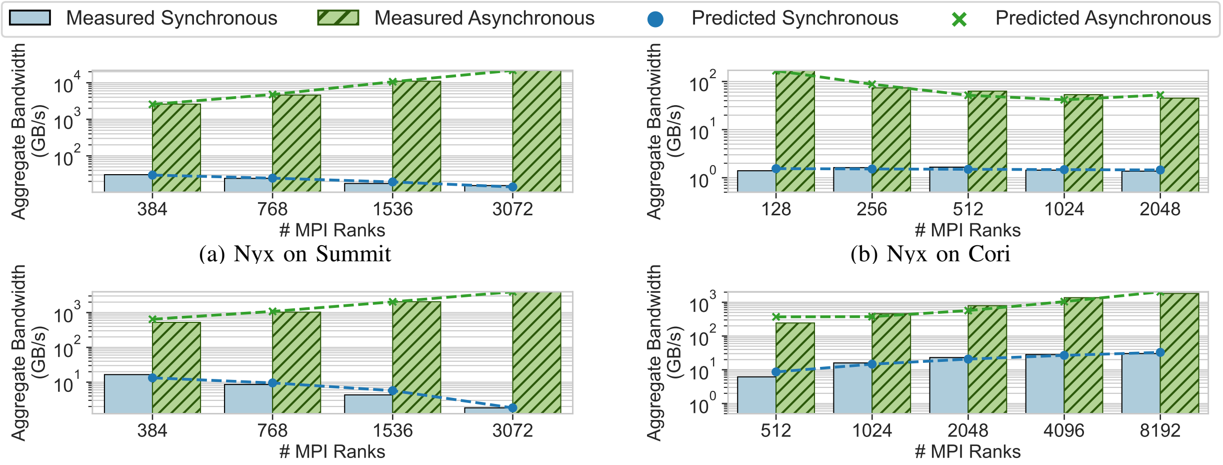
However, on Cori-Haswell, we observe the small data size of each request leads to poor synchronous aggregate write performance at all scales, and the asynchronous aggregate write bandwidth does not scale up linearly. Even though the computation time can overlap the I/O time to the file system fully, the asynchronous write performance is limited by the transactional overhead on Cori-Haswell. The data decreases in size and is not enough to saturate even the bandwidth of the on-node DRAM. When we increase the number of MPI ranks, each rank is operating on a smaller amount of data. This results in the aggregate performance of synchronous I/O to not scale up as the number of MPI ranks increase.



(a) VPIC-IO on Summit (b) VPIC-IO on Cori

(c) BD-CATS-IO on Summit (d) BD-CATS-IO on Cori

Fig. 3: A comparison of the aggregate bandwidth (log scale) of the I/O kernels (weak scaling) between synchronous and asynchronous modes. The estimated performance is shown as a dotted line.



(a) Nyx on Summit (b) Nyx on Cori

(c) Castro on Summit (d) Castro on Cori

Fig. 4: A comparison of aggregate bandwidth (log scale) of two large scale cosmology simulations between synchronous and asynchronous modes with estimated performance shown as a dotted line.

4) Castro: We run Castro using the configuration described in Section IV. Using the same data size, we scale the number of MPI Ranks across multiple nodes. This allowed us to see how the file I/O scales with synchronous I/O and asynchronous I/O modes. In Fig. 4c and 4d, we show the aggregated bandwidth reported on Summit and Cori-Haswell systems, respectively. Since the dataset size remains constant with the number of MPI ranks (strong scaling), the amount of data each rank processes and writes decreases proportionally for writing plot files.

As a result, with the number of ranks increasing, the individual node bandwidth to the parallel file system decreases similar to EQSIM. On Summit, we see for synchronous I/O the aggregate bandwidth decreases as we scale up the number of MPI Ranks. On Cori-Haswell, we see the synchronous I/O performance increases until it saturates at 2048 MPI Ranks. This scaling of the synchronous I/O rate differs between the two systems primarily because of how the Alpine file system on Summit allocates storage resources. The GPFS does not provide a way for the user to declare striping for file or directories; instead, it is tuned to react to the workload. In this case, we can see that a strong scaling scenario impacts the observed aggregate I/O rate to the file system when we decrease the data size for each MPI Rank as we scale up.

With asynchronous I/O enabled, we see the opposite trend. On Summit and Cori-Haswell, the computational phase is sufficiently large to completely hide the I/O cost. This results in a linear speedup on both systems, since the cost to make a transactional copy of the data is constant for each node.

5) Cosmoflow: We evaluate the I/O phase for Cosmoflow training using the configuration described in Section IV. In Fig. 5, we plot the aggregate read bandwidth for reading a batch. For synchronous I/O, the performance does not scale after 128 nodes; whereas, the asynchronous I/O is able to maintain a higher bandwidth. Our model is able to accurately estimate the I/O performance based on the best maximum I/O rates from previous iterations. We only run Cosmoflow on Summit due to the availability of GPUs.

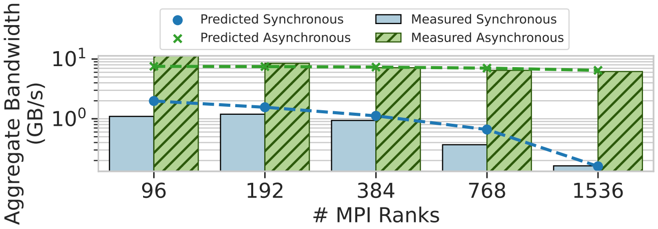


Fig. 5: A comparison of aggregate bandwidth (log scale) of Cosmoflow on Summit between synchronous and asynchronous modes with estimated performance shown as a dotted line.

6) EQSIM: We ran EQSIM by keeping the problem size the same as we increase the number of MPI Ranks across multiple nodes (i.e.,strong scaling). In Fig. 6, we plot the performance difference between synchronous and asynchronous I/O on Summit. Since the problem size remains the same as we scale up, the size of the data on each rank decreases proportionally. This causes the synchronous I/O performance to decrease while the asynchronous I/O performance remains consistent. We are able to model the performance of both I/O modes accurately, as shown by the dotted lines in Fig. 6. We observe similar trend with EQSIM on Cori as Castro, thus we omit those results due to space constraints.

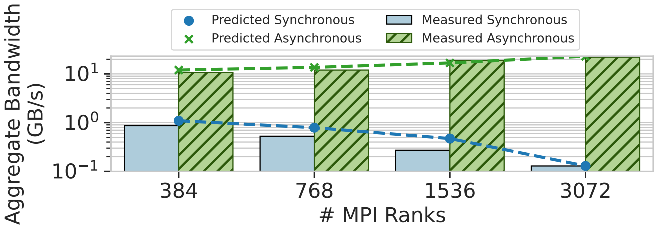


Fig. 6: A comparison of aggregate bandwidth (log scale) of EQSIM on Summit between synchronous and asynchronous modes with estimated performance shown as a dotted line.

B. Impact of Computation and I/O Phase Overlap

1) Partial Overlap of Computation Phase: For iterative simulations, a computation phase is composed of one or more time steps before performing a checkpoint in an I/O phase. The I/O frequency is a parameter that is configured by the application developer. In all the above experiments, we have explored the ideal scenario for asynchronous I/O where computation phases were longer than I/O phases. To study the impact of shorter computation phases, we experiment varying the number of simulation time steps for each computation phase. Partial overlap of computation and I/O phases can decrease the effectiveness of asynchronous I/O.

In Fig. 7, we vary the number of simulation time steps per computation phase on Nyx from 1 to 192. We recall that, in our previous experiments (Fig. 4b), we configured Nyx so as to perform a checkpoint every 20 time steps. In general, increasing the check-pointing frequency (i.e., decreasing the number of time-steps per computation phase) will increase the duration of the application because more I/O is performed. With asynchronous I/O, we see the impact of performing more I/O is less pronounced than with synchronous I/O until the computation phase becomes too short to overlap with the I/O phase (at 1 time-step per computation phase).

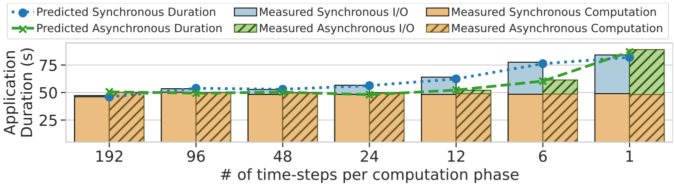


Fig. 7: A comparison of synchronous and asynchronous modes with varying number of time steps per computation phase for Nyx on Cori. The estimated application duration for each mode is shown as a dotted line.

C. Factors Affecting Performance Predictability

The experiments above show that our empirical model can make accurate estimations of the aggregate I/O bandwidth achieved by applications both in strong and weak scaling scenarios. For asynchronous I/O, we observed a very strong linear correlation between the scaling factors (dataset size and number of MPI Ranks) and the observed aggregate I/O bandwidth, with an r^2 above 90% for each estimation. In the synchronous I/O case, while lower, the r^2 consistently above 80% still shows good linear correlation. As noted in Section III-B2, our goal was to model the ideal case performance (i.e., the maximum aggregate I/O bandwidth achieved) for both synchronous and asynchronous I/O. This allowed high accuracy using linear methods. In practice, however, the accuracy of the model depends on the degree of contention, which can vary based on workloads sharing the network and file system. Where *Node-level contention* in HPC systems can be avoided since with batch schedulers typically allocate entire nodes to a single application, *Full System-Level contention* can cause some variability in measurements.

The impact of contention on I/O in synchronous and asynchronous modes at the full system level can observed as variability across runs. For all of our experiments, we run each I/O kernel and application in synchronous and asynchronous modes at least 5 times across multiple days to account for interconnect and storage system contention with other applications running on the system.

Fig. 8 plots all of the aggregate bandwidth measurements for the VPIC-IO benchmark on Summit across multiple days. As can be seen, a benefit of asynchronous I/O is to hide the system-level variability, leading to consistent aggregate I/O bandwidth independent of the full system-level contention.

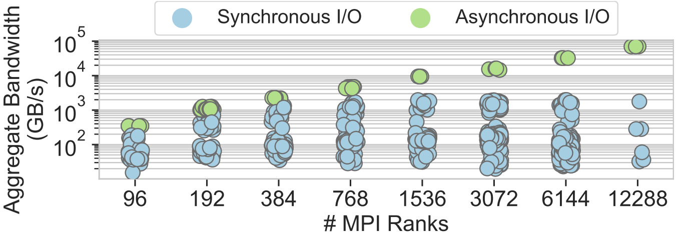


Fig. 8: A comparison of VPIC-IO performance variability on Summit with I/O in synchronous and asynchronous modes.

VI. Summary and Outlook

A. Lessons Learned

In this study we have provided a systematic approach to quantify the benefits of asynchronous I/O. Although enabling asynchronous I/O can alleviate I/O bottlenecks in most cases, we found limitations for scalability in current implementations. By measuring the performance of synchronous and asynchronous I/O with I/O kernels and real-world applications, we also identified key performance differences between two large-scale high-performance computing systems. Although Summit's GPFS storage is capable of a high aggregate bandwidth for synchronous I/O, the performance is heavily dependent on the data size and number of nodes. Both of these parameters affect how Summit's storage system allocates I/O resources to an application. Performance of synchronous I/O on Cori-Haswell also depends on the number of nodes; however, the user is given more control on the allocation of I/O resources for each application (for example, by setting the stripe count). We note that the data size and number of nodes impact how much of the synchronous I/O we can saturate before reaching the limits of the resources selected.

Typically, the performance scalability of an application depends on whether it increases the problem size for more resources, weak scaling, or keeps the problem size constant, strong scaling. From this study, we note that this also applies to the performance scalability of I/O, whether synchronous or asynchronous. For strong scaling applications, I/O latency can become a bottleneck for asynchronous I/O due to smaller data sizes. The cost to setup the data transfers relative to the amount of data being transferred decreases the effective bandwidth for the I/O requests. Nonetheless, we find that asynchronous I/O enables more predictable performance of I/O requests. By using a buffering location that is not being shared across multiple users or workloads, such as in-node memory or SSD, we can hide I/O performance variability with asynchronous I/O. These trade offs on both systems and applications will impact how a user of these checkpoint-based applications configures the I/O frequency. The proposed methodology allows to accurately estimate the aggregate I/O bandwidth achieved by asynchronous I/O (in the absence of node-level oversubscription), and the ideal aggregate bandwidth achieved by synchronous I/O (in the absence of full system-level contention).

B. Conclusions

In this study, we have taken the first steps of understanding the benefits and limitations of asynchronous I/O. We have proposed a simple performance model to estimate the aggregate I/O bandwidth achievable by an application based on past iterations with both synchronous and asynchronous I/O. We have validated our model and performed an extensive experimental evaluation of synchronous and asynchronous I/O on two large-scale HPC systems, Summit and Cory-Haswell, using two I/O kernels and four real-world science applications run at scale. In our experiments, we have used the HDF5 asynchronous I/O feature. Our evaluation and analysis have highlighted benefits and limitations of asynchronous I/O based on application characteristics and data transfer setup costs.

C. Acknowledgements

This manuscript has been authored by an author at Lawrence Berkeley National Laboratory supported under Contract No. DE-AC02-05CH11231 with the U.S. Department of Energy and by authors at North Carolina State University supported under National Science Foundation's award CNS-1812727. This research also 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 and the 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] C.-Y. Chou *et al.*, "A semi-empirical model for maximal linpack performance predictions," in *Sixth IEEE International Symposium on Cluster Computing and the Grid (CCGRID'06)*, vol. 1, 2006, 4 pp.–348. DOI: 10.1109/CCGRID.2006.10.

[2] H. Kredel *et al.*, "A hierarchical model for the analysis of efficiency and speed-up of multi-core cluster-computers," in *2015 10th International Conference on P2P, Parallel, Grid, Cloud and Internet Computing (3PGCIC)*, 2015, pp. 207–215. DOI: 10.1109/3PGCIC.2015.49.

[3] M. Folk *et al.*, "An overview of the HDF5 technology suite and its applications," in *EDBT/ICDT*, 2011, pp. 36–47.

[4] Q. Liu *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, ISSN: 1532-0634.

[5] H. Tang *et al.*, "Transparent Asynchronous Parallel I/O using Background Threads," *IEEE Transactions on Parallel and Distributed Systems*, pp. 1–1, 2021. DOI: 10.1109/TPDS.2021.3090322.

[6] H. Tang *et al.*, "Toward Scalable and Asynchronous Object-Centric Data Management for HPC," in *2018 18th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, 2018, pp. 113–122. DOI: 10.1109/CCGRID.2018.00026.

[7] M. M. A. Patwary *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, USA: IEEE, 2015, pp. 1–12.

[8] S. Byna *et al.*, "Exahdf5: Delivering efficient parallel i/o on exascale computing systems," *Journal of Computer Science and Technology*, vol. 35, pp. 145–160, 2020.

[9] S. Seo *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.

[10] S. R. Walli, "The POSIX Family of Standards," *StandardView*, vol. 3, no. 1, pp. 11–17, 1995, ISSN: 1067-9936.

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

[12] S. Bhattacharya *et al.*, "Asynchronous I/O Support in Linux 2.5," 2003.

[13] K. Elmeleegy *et al.*, "Lazy asynchronous i/o for event-driven servers.," in *USENIX Annual Technical Conference, General Track*, 2004, pp. 241–254.

[14] R. A. Oldfield *et al.*, "Lightweight i/o for scientific applications," in *2006 IEEE International Conference on Cluster Computing*, IEEE, USA: IEEE, 2006, pp. 1–11.

[15] C. M. Patrick *et al.*, "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.

[16] S. Zhou *et al.*, "Application Controlled Parallel Asynchronous IO," in *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, ser. SC '06, 2006, 178–es, ISBN: 0769527000. DOI: 10.1145/1188455.1188639.

[17] R. Thakur *et al.*, "On Implementing MPI-IO Portably and with High Performance," in *ICPADS*, Atlanta, Georgia, USA, 1999, pp. 23–32, ISBN: 1-58113-123-2.

[18] C. Docan *et al.*, "DataSpaces: An interaction and coordination framework for coupled simulation workflows," vol. 15, Jan. 2010, pp. 25–36. DOI: 10.1145/1851476.1851481.

[19] W. F. Godoy *et al.*, "Adios 2: The adaptable input output system. a framework for high-performance data management," *SoftwareX*, vol. 12, p. 100 561, 2020, ISSN: 2352-7110. DOI: https://doi.org/10.1016/j.softx.2020.100561.

[20] T. Alturkestani *et al.*, "Mlbs: Transparent data caching in hierarchical storage for out-of-core hpc applications," in *2019 IEEE 26th International Conference on High Performance Computing, Data, and Analytics (HiPC)*, 2019, pp. 312–322. DOI: 10.1109/HiPC.2019.00046.

[21] T. Alturkestani *et al.*, "Maximizing i/o bandwidth for reverse time migration on heterogeneous large-scale systems," in *Euro-Par*, 2020.

[22] B. Dong *et al.*, "Data Elevator: Low-Contention Data Movement in Hierarchical Storage System," in *HiPC*, 2016, pp. 152–161.

[23] Z. Wang *et al.*, "Facilitating staging-based unstructured mesh processing to support hybrid in-situ workflows," in *2021 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, 2021, pp. 960–964. DOI: 10.1109/IPDPSW52791.2021.00152.

[24] A. Maurya *et al.*, "Towards Efficient Cache Allocation for High-Frequency Checkpointing," in *HiPC'22: 29th IEEE International Conference on High Performance Computing, Data, and Analytics*, Bangalore, India, Dec. 2022.

[25] H. Shan *et al.*, "Characterizing and predicting the I/O performance of HPC applications using a parameterized synthetic benchmark," in *SC' 08*, Austin, TX: ACM/IEEE, 2008.

[26] S. Byna *et al.*, "Parallel I/O Prefetching Using MPI File Caching and I/O Signatures," in *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, ser. SC '08, Austin, Texas: IEEE Press, 2008.

[27] M. Meswani *et al.*, "Modeling and predicting disk I/O time of HPC applications," in *High Performance Computing Modernization Program Users Group Conference*, ser. HPCMP-UGC, 2010, pp. 478–486.

[28] B. Behzad *et al.*, "Optimizing i/o performance of hpc applications with autotuning," *ACM Trans. Parallel Comput.*, vol. 5, no. 4, Mar. 2019, ISSN: 2329-4949. DOI: 10.1145/3309205.

[29] S. Kumar *et al.*, "Characterization and modeling of PIDX parallel I/O for performance optimization," ser. SC '13, Denver, Colorado, 2013, 67:1–67:12.

[30] M. Dorier *et al.*, "Omnisc'io: A grammar-based approach to spatial and temporal i/o patterns prediction," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '14, New Orleans, Louisana, 2014, pp. 623–634.

[31] J. Lofstead *et al.*, "Six degrees of scientific data: Reading patterns for extreme scale science io," Jan. 2011, pp. 49–60. DOI: 10.1145/1996130.1996139.

[32] P. Schober *et al.*, "Correlation coefficients: Appropriate use and interpretation," *Anesthesia and analgesia*, vol. 126, no. 5, pp. 1763–1768, May 2018, ISSN: 0003-2999. DOI: 10.1213/ane.0000000000002864.

[33] S. S. Vazhkudai *et al.*, "The Design, Deployment, and Evaluation of the CORAL Pre-Exascale Systems," Jul. 2018.

[34] T. Declerck *et al.*, "Cori - a system to support data-intensive computing," *Cray User Group (CUG) meeting 2016*, 2016.

[35] S. Byna *et al.*, *Parallel I/O Kernels (PIOK)*, https://code.lbl.gov/projects/piok/.

[36] S. Byna *et al.*, "Parallel I/O, Analysis, and Visualization of a Trillion Particle Simulation," in *Supercomputing*, 2012, 59:1–59:12, ISBN: 978-1-4673-0804-5.

[37] A. S. Almgren *et al.*, ""nyx: A massively parallel amr code for computational cosmology"," *The Astrophysical Journal*, vol. 765, no. 1, p. 39, 2013.

[38] W. Zhang *et al.*, "AMReX: A framework for block-structured adaptive mesh refinement," *Journal of Open Source Software*, vol. 4, no. 37, p. 1370, May 2019. DOI: 10.21105/joss.01370.

[39] A. Almgren *et al.*, "Castro: A massively parallel compressible astrophysics simulation code," *Journal of Open Source Software*, vol. 5, no. 54, p. 2513, 2020. DOI: 10.21105/joss.02513.

[40] D. Mccallen *et al.*, "The earthquake simulation (eqsim) framework for physics-based fault-to-structure simulations," in *17WCEE*, Jun. 2020.

[41] A. Mathuriya *et al.*, "Cosmoflow: Using deep learning to learn the universe at scale," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*, ser. SC '18, Dallas, Texas: IEEE Press, 2018. DOI: 10.1109/SC.2018.00068.