NIOBE: An Intelligent I/O Bridging Engine for Complex and Distributed Workflows

Kun Feng, Hariharan Devarajan, Anthony Kougkas, and Xian-He Sun
Department of Compute Science, Illinois Institute of Technology
{kfeng1, hdevarajan}@hawk.iit.edu, {akougkas, sun}@iit.edu

***Abstract*—In the age of data-driven computing, integrating *High Performance Computing* (HPC) and *Big Data* (BD) environments may be the key to increasing productivity and to driving scientific discovery forward. Scientific workflows consist of diverse applications (i.e., HPC simulations and BD analysis) each with distinct representations of data that introduce a semantic barrier between the two environments. To solve scientific problems at scale, accessing semantically different data from different storage resources is the biggest unsolved challenge. In this work, we aim to address a critical question: "How can we exploit the existing resources and efficiently provide transparent access to data from/to both environments". We propose *iNtelligent I/O Bridging Engine* (NIOBE), a new data integration framework that enables integrated data access for scientific workflows with asynchronous I/O and data aggregation. NIOBE performs the data integration using available I/O resources, in contrast to existing optimizations that ignore the I/O nodes present on the data path. In NIOBE, data access is optimized to consider both the ongoing production and the consumption of the data in the future. Experimental results show that with NIOBE, an integrated scientific workflow can be accelerated by up to 10x when compared to a no-integration baseline and by up to 133% compared to other state-of-the-art integration solutions.**

I. Introduction

As the exascale era approaches, the total data volume produced and consumed has exploded [1]. Large scale scientific simulations, typically running in HPC environments, act as data producers generating large amounts of data stored on a *Parallel File System* (PFS). Data is currently stored in a self-descriptive format, such as HDF5 [2] and netCDF [3], and consumed by large-scale data analysis frameworks custom to the HPC ecosystem. Recently, BD frameworks, such as Hadoop [4] and Spark [5], are emerging with increasingly large popularity and are known for their usability, capability, and versatility in processing large amounts of data. Using BD analysis applications can significantly enhance the data processing power, facilitate the exploration of huge data sets, and boost scientific discovery [6]. BD frameworks are usually deployed in distributed computing environments, significantly different from HPC machines. However, the tools and cultures of HPC and BD have diverged, to the detriment of both [7], and unification is essential to address a spectrum of major research domains. The very first step towards this unification is to solve the semantically different data representations by bridging storage solutions from both ecosystems, and thus, to offer an *integrated data access*. As one of the key system components, storage systems play an imperative role in the feasibility of such mixed environments. When applications from both environments are used in a scientific workflow, resolving data dependencies between different phases of the workflow with diverse semantics while maintaining high performance is challenging and can create obstacles for scientists.

To address this issue, different solutions have been proposed. First, many supercomputing sites, such as *National Aeronautics and Space Administration* (NASA)'s Goddard Space Center [8], utilize two separate clusters to achieve their mission (i.e., one for simulations using HPC software stack called *Discover* and one for analysis and visualization called *Data Analytics Storage Service* (DASS) [9] running *Apache Big Data Stack* (ABDS)). This implies that scientists must copy data between two clusters between each phase of their scientific workflows. Even though the storage solutions can perform well for native workloads inside each cluster, data movement and transformations introduce high I/O cost and significantly reduce the benefit of the native performance [10].

Second, to avoid the expensive data movement, many *connectors* have been introduced to expose data to applications from non-native environments. For instance, IBM's Spectrum Scale [11] and Lustre offer a *Hadoop Distributed File System* (HDFS) connector [12], [13] that allows a variety of BD applications, spanning from MapReduce to SQL, to run on top of a PFS. Similarly, Google's Cloud Storage FUSE, Amazon's AWS Storage Gateway, and Microsoft's Azure Files and Disks all have built file semantics on top of their existing object store solutions. However, using a single storage system as the unified solution cannot serve all workloads since it's vendor-specific, proprietary, and not optimized for all storage systems, which may decelerate the performance of the entire workflow [14].

Lastly, several open-source specialized middleware libraries, such as IRIS [15], SciDP [16], and CephFS [17], that are responsible to map semantically different data representations to one another have been developed to accelerate cross-platform data access for integrated scientific workflows. Resources on the compute nodes are used to facilitate the integrated data access. Specifically, dedicated client processes, CPU cycles, and memory space are required to convert data before issuing I/O requests to storage systems. These operations are carried out along application processes on the same nodes, which consequently slow down the applications due to hardware

978-1-7281-0858-2/19/$31.00 ©2019 IEEE

contention and software overheads.

In this research, we first identify the existence of several available resources in the I/O path between compute nodes and storage nodes (e.g., I/O forwarding layer [18], data staging [19], [20], burst buffers [21], and data nodes [22]) We then propose a new I/O service called iNtelligent I/O Bridging Engine (NIOBE) that can leverage these resources to enable an integrated path to data, avoiding expensive overheads, and accelerate integrated data access for scientific workflows. In other words, NIOBE moves the conversion of integrated I/O away from the compute nodes and returns the compute resources back to application processes. NIOBE utilizes the data aggregation hardware and software to enable asynchronous I/O integration. The repetitive nature of scientific workflows (i.e., periodic I/O behavior [23]) is utilized by NIOBE to make better decisions for the destination of integrated I/O. With NIOBE, the performance of the integrated scientific workflow can be improved significantly by minimizing mapping overheads, performing data aggregations before integration, and offering asynchronous I/O to free precious resources back to the applications. The contributions of NIOBE are as follows:

- Demonstrate how to use existing data aggregation resources to enable transparent data access for scientific workflows.
- Highlight how important a modular and non-intrusive design is to guarantee portability and interoperability with existing I/O middleware.
- Illustrate the benefit of overlapping I/O with data integration to improve the throughput by avoiding blocking I/O.
- Show the power of having a global view of data accesses compute nodes perform to enable several I/O optimizations such as deduplication, compression, and caching.
- Boost end-to-end performance of workflows by up to 10.5x than a baseline of no-integration and by 133% compared to state-of-the-art solutions.

II. Background

A scientific workflow system is built by coupling a sequence of applications to serve different goals of scientific research. In a typical integrated scientific workflow, HPC applications generate large volume of data, which is accessed and analyzed by BD applications. The analysis results are then fed back to the HPC applications to drive discovery. One common issue in such an integrated workflow is the storage which serves as the shared data pool between two distinct environments. In a native environment, HPC applications (e.g., *Message Passing Interface* (MPI)-based simulations) typically utilize PFS to store well-structured data and access it via POSIX I/O or MPI-IO interface. At the same time, BD applications (e.g., *High Performance Data Analytics* (HPDA) applications) are designed to process unstructured or semi-structured data and the preferred storage system is most likely an Object Store. The differences in workloads and data access APIs create an obstacle to access data from/to another environment. For instance, NASA scientists convert output of the climate modeling application *NASA-Unified Weather Research and Forecasting* (NU-WRF) to text format and load it into Hive [24] running on another cluster to enable SQL-like queries upon the data. Such a process is found to be extremely slow [16] and significantly prolongs the turnaround time to generate useful insights from HPC applications.

To overcome this, several approaches have been proposed.

Convert and Copy: Scientific workflows are executed in two separate and isolated clusters. Two sets of compute and storage environments work as independent systems with the data dependency solved by simple data conversion and copy to and from different storage subsystems. Such a process is known to be exceedingly expensive [16]. The data conversion and copying dominates the total execution time, and thus, becomes the bottleneck. This becomes especially prohibitive once the data size that is moved from cluster to cluster grows.

Connector-based integration: Hadoop, as one of the popular BD frameworks, is a target platform to enable integrated data access. Several solutions have been proposed to enable Hadoop to access data from PFS in addition to its native HDFS. HDFS connectors attempt to convert PFS into HDFS-compatible file systems to build unified storage system for both environments. However, HDFS connectors are not designed to serve integrated workflows since using PFS to be the underlying file system for HDFS cannot provide the optimal performance. PFS can only be configured to favor either HPC or BD applications and, consequently, slow down the other group of applications. Moreover, these connectors are either designed for a specific PFS, such as HDFS Transparency from IBM [12] and Intel Enterprise Edition for Lustre Software on top of Lustre [25], or found to be slow [13], [16].

Application-level integration: Several runtime libraries have been proposed to enable cross-environment data access. SciDP [16] is an extended Hadoop framework to utilize the MPI library to access data from PFS. IRIS [15], on the other hand, is a library that dynamically converts PFS data access to *Key-value Store* (KVS). Both frameworks utilize computing resources to carry out the integration of data access to another storage system. Application's CPU cores are responsible in mapping semantically different data between the source and the destination storage subsystems. The associated costs of data translation lies onto the application which decreases the flexibility of integrated systems, increases the programming errors, and forces the data integration to be synchronous.

Our approach for data integration: In modern HPC systems, specialized resources, such as the I/O forwarding layer [26], Burst Buffers [21], and data staging [20], [27], have been introduced to ease the pressure to external storage. These resources are deployed in between compute and storage nodes with the responsibility to aggregate I/O requests, as well as to stage in/out data. With the help of these middleware resources, *overlapped/asynchronous I/O* is enabled to perform data operations in a decoupled fashion as applications continue their computations after data is sent to these data aggregation services which flush the data to the PFS in the background. Existing supercomputers already have such technologies in production. For example, the Cori system at *National Energy Research Scientific Computing Center* (NERSC) has a 1.8 PB

Burst Buffer managed by Cray Data Warp with a peak I/O bandwidth of 1.7 TB/s [28].

The additional dedicated hardware provides a perfect place to also carry out data integration. This middleware layer holds application data closer to compute nodes with high I/O bandwidth and can serve applications from different environments at the same time. This is the observation that our proposed system NIOBE leverages to offload the conversion to those intermediate hardware resources freeing computation resources. NIOBE utilizes the capabilities of existing data aggregation services to aggregate I/O requests from applications and intelligently selects the best storage solution based on the knowledge of the access pattern of all the involved components. Integrated data access is managed transparently as NIOBE intercepts existing incoming data without user's intervention. NIOBE can act as an add-on to existing data aggregation services such as I/O forwarders and Burst Buffers. Custom APIs are also created to maximize the capability of NIOBE with extra information provided by the users. NIOBE is designed to be modular to enable easy extension to support several storage systems in the integrated scientific workflow (e.g., from PFS to KVS to HDFS etc.).

III. NIOBE

A. Design Requirements

NIOBE is expected to be an add-on to the existing data aggregation services for integrated scientific workflows. As a result, we set three major requirements as our design priorities.

Transparency: The existing applications have been developed and used for a long time. The interfaces they are using have also been standardized to maximize the portability across different platforms. Transparency to applications allows NIOBE to benefit more users. Thus, NIOBE needs to work with minimal to no modification and intervention to the user's application source code.

Compatibility: As we mentioned in Section II, there are some existing data aggregation services. NIOBE is required to work alongside these services to accelerate integrated data accesses while not affecting the existing objectives. The design is expected to be compatible with as many existing data aggregation services as possible.

Lightweight: As we move towards exascale, the scale of the system is growing dramatically. Current systems are running using data aggregation services to accelerate their I/O operations. They are designed to work well with hundreds of thousands of processes across the system. The introduction of a new service plugin, such as NIOBE, has to keep minimal overhead to ensure it would not limit the performance of the system and become its bottleneck.

Workflow-specific optimization: Since NIOBE targets integrated scientific workflows, specific optimizations have to be proposed to take advantage of the characteristics of these systems. Each phase of a scientific workflow can be drastically different. These optimizations need to be general enough to be applied to as many integrated scientific workflows as possible.

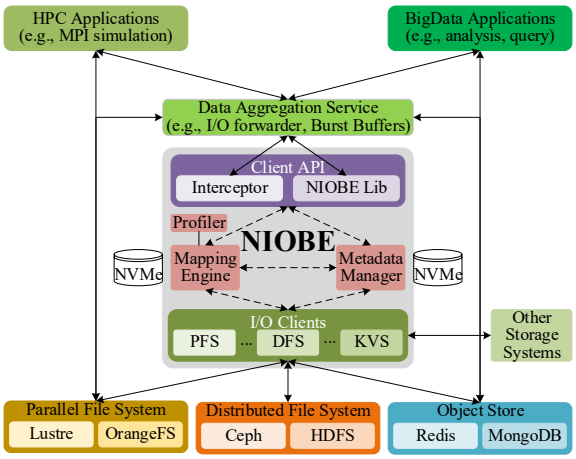


Fig. 1: Architecture design of NIOBE

B. High-level Architecture

Figure 1 shows the architecture of NIOBE. NIOBE sits in-between two environments and works as a bridge for I/O services in both environments. As discussed in Section II, applications in a traditional integrated scientific workflow prefer to access data vertically to/from their native storage systems. Data dependency in-between two environments is currently solved with slow data copy and conversion. NIOBE is designed to bring two native vertical I/O paths closer to each other by integrating the I/O from both paths and make better decisions based on knowledge from executed phases in the workflow. Additionally, with modern hardware architecture, shared data can be stored in the data aggregation nodes (e.g., Burst Buffers) temporarily without reaching the remote global storage system right away. Existing data aggregation services, such as I/O forwarders and Burst Buffers, can be used as storage for the shared data pool for both environments. NIOBE intercepts I/O calls to existing data aggregation services, and, thus, gains access to the aggregated data. The I/O requests are then analyzed and possibly converted into one or more requests targeting a different storage system accordingly. NIOBE is designed to be transparent to applications and aggregation services. Enabling NIOBE is as simple as turning a switch on by preloading the NIOBE library (e.g., via LD_PRELOAD mechanism) with potential user configuration afterwards. Similar interception is also used to connect analysis applications to NIOBE. Analysis applications from another environment accesses the data as from native storage system. NIOBE maps the original metadata into the namespace within NIOBE and carries out the I/O operation as required by the applications.

C. Component Analysis

Several components are designed, each one in charge of different functionalities in NIOBE.

*1) **Client API**:* The client API is designed to provide users an interface to communicate with NIOBE. There are two different ways of using the client API of NIOBE.

Compatibility-prioritized Mode. User applications can run as before and utilize existing data aggregation services, such

as I/O forwarder and Burst Buffers, to enable overlapped I/O and fast data sharing. NIOBE transparently intercepts data aggregation services to enable access to the same memory buffer, which is used to receive data from application processes and flush to the underlying PFS. NIOBE consequently converts the data for integrated access to the optimal storage system for best overall I/O performance in an integrated scientific workflow. Information about the requests is collected from the standard I/O routines and job descriptions to facilitate the decision making. This mode can maximize the application compatibility of NIOBE. Theoretically, all the existing applications which use data aggregation services can benefit from NIOBE for their integrated workflow. However, standard interfaces of existing data aggregation services have limited information about the data such as how it will be accessed by following data consumers. Thus, NIOBE can only make the best effort to understand the access pattern of data consumers by profiling the access pattern during the first run with the I/O Profiler (see Section III-C2). NIOBE utilizes the repetitive nature of scientific workflows and assume a unified access pattern throughout all the iterations. With the access pattern obtained by profiling the first iteration, NIOBE can decide the optimal storage plan to provide the best overall performance for the entire workflow using the output of its I/O profiler.

Performance-prioritized Mode. To provide better decisions on data integration, NIOBE also introduces some new APIs to allow users to pass data access pattern information to NIOBE Lib without the overhead of the I/O Profiler. An XML file can be specified as the description of access pattern of the consequent phases in the workflow. For example, an HPC application can specify that the generated data will be analyzed by a K-means clustering application which expects the data to be in key-value pair of $< coordinate,\ data >$. The granularity of the data access will be in a grid size of $4 \times 4 \times 4$ when it is in the memory. With this information, NIOBE will decide to write the data into KVS as a better option compared to the default destination PFS. To match the access size, the raw data will be partitioned into multiple $4 \times 4 \times 4$ grids as key-value pairs. The data at each grid will be written to KVS with the coordinate as the key and the partitioned raw data as the value. Since this information is provided by the user, it can be more accurate than what is detected by the I/O Profiler. In addition, it can avoid the overhead of using the profiler, which may slow down the application. It does, however, require the user to know exactly what the I/O requirements are, which may not always be possible.

*2) **I/O Profiler**:* NIOBE's I/O Profiler is designed to capture how data is consumed to provide guidance for the Mapping Engine to determine the optimal storage policy. When the I/O Profiler is enabled, it may slightly affect the performance of the application that is being profiled by some overheads. Nevertheless, due to the repetitive nature of the workflow, the I/O Profiler only needs to be triggered once in the first iteration of execution. The access pattern of following iterations is highly possible to be the same as the first one. Furthermore, minimal information is caught to lower the overhead as much as possible. The I/O Profiler keeps track of the granularity of all the data accesses generated by data producer applications. It also records access information of data consumer applications, such as which portion of the data is accessed again and how the data is accessed. Since not all the data will be accessed after it is generated, profiling the reuse coverage of the data can avoid redundant conversion in NIOBE to minimize overhead. Profiling the access pattern in the data producer applications can help the decision of mapping in the Mapping Engine. For instance, I/O Profiler may understand a piece of data will be read 1MB each time with a 2MB strided pattern. With this knowledge, the Mapping Engine can convert the future write requests to 1MB objects in KVS to utilize the significant performance benefit of KVS in the read performance of small access over PFS. In contrary, the Mapping Engine may keep the original data path if a sequential access pattern is detected by the I/O Profiler since PFS can provide good enough read performance in such case. Conversion in such case loses too much performance in writing the data, which hurts the overall performance instead. The profiling result is saved in memory to avoid unnecessary disk I/O. Thus, the overhead of I/O Profiler can be minimized. The I/O Profiler sends the profiling result to the Mapping Engine to adjust the default storage plan for integrated data accesses.

*3) **Mapping Engine**:* The Mapping Engine is the decision maker in NIOBE. It determines whether a data access needs to be converted due to all the knowledge it perceives. This information comes from the workflow data dependency graph, which can be easily generated from the workflow script, profiling results from I/O Profiler and the request itself. If a piece of data needs to be accessed by another application with different semantics, the access will be converted to the corresponding format to speedup the anticipated access. If the data will not be touched again, NIOBE will ignore it and carry out the I/O as the default system. The decision is also made based on the trade off between current access and future ones. Using our previous work [15], [29], we have equipped NIOBE with a model that can intelligently select the best storage policy. NIOBE is also capable of buffering data into locally-attached high performance storage devices, such as NVMe SSDs, when the local memory is not large enough. These two main features of NIOBE are performance-oriented and can boost the overall performance experienced by applications.

Notice that such a mapping is different from IRIS, which is carried out on compute nodes. NIOBE utilizes the aggregation nodes on the I/O path and leaves all the compute resources to the applications. More importantly, NIOBE selects the optimal storage policy based on global access information of the data accesses. No redundant data from the data producer applications is converted as a result of the profiling phase in compatibility-prioritized mode or the user inputs in performance-prioritized mode. Moreover, generated data is optimally stored to match the access pattern of the data consumer applications to guarantee the best performance.

*4) **Metadata Manager**:* The Metadata Manager keeps track of all the integrated data accesses. It stores locations of all the

data mapped by NIOBE. It maintains a distributed namespace across all the nodes which run NIOBE services. Mappings between files and objects are stored as hash map entries in the Metadata Manager. For write access, entries will be inserted with the file name or the key of the data as the identifier of the metadata. For read access, the hash map is queried to return the location of the data. The metadata is stored in memory to maximize the insertion and query performance. It will be flushed to local storage periodically to guarantee persistency.

*5) **I/O Clients**:* An I/O client is the interface of NIOBE to communicate with an underlying storage system. It has modules for all supported storage systems, which include PFS such as IBM Spectrum Scale (used to be called *General Parallel File System* (GPFS), Lustre and OrangeFS, *Distributed File System* (DFS) such as HDFS, Ceph and GlusterFS, and KVS such as MongoDB, DynamoDB and Redis. After the data is converted by the Mapping Engine, the request will be forwarded to I/O clients to actually carry out the I/O operation. Internally, the operations will be described using standard interfaces, such as fopen/fwrite/fread/fclose for POSIX I/O and put/get/delete for key-value access. The corresponding module consequently translates the standard description to its native interface. Due to the modular design of I/O clients, NIOBE can be extended to other storage systems with minimal development of a corresponding module driver.

D. Implementation Details

The implementation quality has a big impact on the final performance of NIOBE.

1) Interceptor: As discussed in Section III-B, NIOBE intercepts standard I/O calls to maintain the compatibility with the existing software stack. Since we utilize the existing data aggregation services, we intercept the *recv()* call in aggregation services. Typically, these services receive I/O requests from the applications running on the compute nodes over the network. With the intercepted *recv()* on the network socket, we can obtain the access to the aggregated I/O data in a non-intrusive way. After the call is intercepted, we change the default memory allocation calls to reallocate the memory block to shared memory allocated by *shm_open* to enable data sharing between the aggregation service's original process and NIOBE's service process. Since the same amount of memory is allocated as the default aggregation services, NIOBE does not increase memory consumption. Resources on aggregation nodes can be provisioned as before without any change. The interception is achieved by using dynamic linking. It is a mechanism provided by Linux's dynamic loader to dynamically change the behavior of the program without the need of recompiling. It is used by setting the optional environmental variable *LD_PRELOAD* to point to a shared library file which contains the new implementation of functions used in the program. When using NIOBE, only one command needs to be added to set *LD_PRELOAD* before starting the data aggregation services. No code modification or compilation is needed for existing aggregation services.

The same technique is used to intercept standard I/O calls, such as fwrite/fread for HPC applications and put/get for BD applications. The intercept calls will be redirected to NIOBE where the metadata is queried to locate the required data which might have been re-directed to another storage system, or still reside on the native storage system.

Users can enable and disable the interception by setting another environmental variable *ENABLE_NIOBE* to manually switch on and off NIOBE. The interceptor will check this variable to behave accordingly.

2) Mapping Engine: The mapping engine of NIOBE is similar with IRIS [15]. We use the balanced mapping from IRIS in NIOBE. More sophisticated mapping can be extended to provide better performance for specific workflows. In this paper, we focus on the proof-of-concept implementation to validate the idea of NIOBE. When mapping file I/O to key-value pair access, tuples of file name, offset and length are converted to a unique key using a hash function. When the same data needs to be analyzed by BD applications, the same hash function is used to locate the key of the data. NIOBE's mapping engine is previously proven to be lightweight and robust while performing faster than other mappings such as that of the various connectors.

3) Metadata Manager: The Metadata Manager in NIOBE is a distributed service running on all data aggregation nodes. The metadata is managed across all participating nodes. Shared memory on each node is allocated to hold the metadata for fast access. The metadata will be periodically flushed to local disk to persistent the metadata. The partition of metadata namespace is achieved with hashing. A fixed size of key range is managed on each node. Once an entry needs to be accessed from/to the metadata, the client process firstly contacts the service process running locally. It consequently contacts the server which is in charge of the corresponding key range that the key belongs to. The hash function can guarantee a unique server for a specific key to actually store the entry. An RPC call will be sent from the local service process to the destination one to carry out the operation remotely.

4) Profiler: For profiling and analyzing of application's I/O patterns we use IOSIG [30]. IOSIG uses similar interception technique as we use in NIOBE to intercept POSIX I/O calls. We also extend IOSIG to profile data access from BD applications to IOSIG keeps track of all data access operations with their parameters to analyze the access pattern, such as file descriptor, offset and length for HPC applications, and file segment translated from the key and corresponding access size for BD applications. File descriptor is consequently converted to a file name that gets fed to the mapping engine to generate keys along with offset and length. The overhead of IOSIG is low enough to guarantee minimal performance impact of the profiling step in NIOBE. To further cut the overhead, we trim IOSIG to only use its I/O tracing functionality. Only minimal data is saved in a memory-mapped file to provide both high performance and persistency.

5) I/O Clients: NIOBE is designed to be modular and extensible. It is supposed to support arbitrary types of PFS

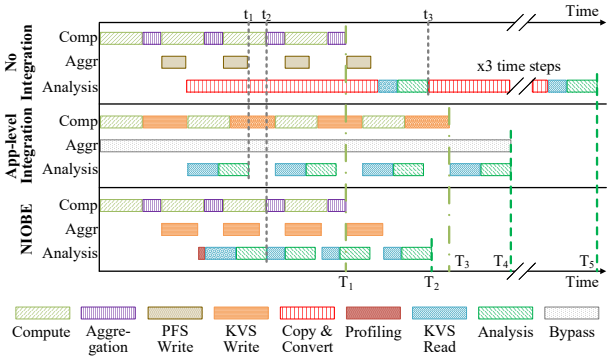


Fig. 2: Timeline comparison between NIOBE and existing solutions. (The time axis is cut after T_4 due to the page limit.)

and KVS. We define the basic functions of I/O clients in a base class. Users can extend the support of new storage system by following existing modules of storage systems and re-implement the corresponding functions.

E. NIOBE in Action

Figure 2 gives the timeline of an example use case for NIOBE and the other two types of existing solutions *No Integration* and *App-level Integration*. In this example, an integrated workflow with one HPC simulation and one BD analysis application is used. Four time steps of data is generated by the simulation and processed by the analysis application. Data conversion is required between two applications by default. All nodes are divided into three groups *Comp*, *Aggr* and *Analysis* which represents nodes for compute, aggregation, and analysis, respectively. *No Integration* represents the default system with data aggregation service enabled, such as systems with I/O forwarding layer and Burst Buffers. Data follows the traditional path without data integration. *App-level Integration* illustrates the existing solutions which integrate data access using resources on compute nodes, such as IRIS and SciDP.

For data generation, *No Integration* and *NIOBE* utilizes data aggregation nodes to overlap writing data of one iteration with compute of next iteration. In contrast, *App-level Integration* ignores these nodes and access the KVS directly from the compute nodes. It also makes the data access slower than that in *NIOBE* due to I/O interference generated by high I/O concurrency from the compute nodes, as observed from longer *KVS Write* times in the *App-level Integration* solution in Figure 2. For data consumption, *App-level Integration* and *NIOBE* addresses the mismatch between files on PFS and key-value pairs on KVS to avoid the extremely expensive and redundant I/O in data copy and conversion.

With data integration in *App-level Integration* and *NIOBE*, users can get the analysis result of the first time step much earlier than *No Integration* (t_1 and t_2 compared to t_3). No data copy and conversion is required as the data has been converted into key-value pairs and stored in KVS. However, the write to KVS in *App-level Integration* consumes compute resources and delays the following iterations of computation. In addition, the aforementioned slower write further delays the remaining iterations. Thus, the data analysis in following iterations starts later than *NIOBE* in *App-level Integration*.

As a workflow-aware data service, *NIOBE* accelerates the whole process and enhances the efficiency by reducing the execution time on both environments. *NIOBE* and *No Integration* are efficient on compute nodes and finishes their computation at T_1 on compute nodes (indicated by the vertical olive green dash-dotted lines). *App-level Integration*, on the other hand, occupies the compute nodes much longer (until T_3) because of the synchronous I/O on compute nodes. The total time of the whole workflow (presented by the vertical bright green dashed lines) shows the benefit of *NIOBE* from the perspective of analysis nodes. *No Integration* is encumbered by slow data copy and conversion and wastes a lot of resources on the analysis nodes. As a result, the finish time of the workflow is much later than the other two solutions (denoted as T_5, notice the break in the time axis). *App-level Integration* has a low utilization on the analysis nodes as the analysis application sits idle and waits for the data of the next iteration due to its ignorance of data aggregation. Visible time intervals can be found between the analysis of the results of each iteration, which pushes the end time to T_4. *NIOBE* can benefit from both the data integration and aggregation, and accelerates the workflow significantly. As it can be seen, *NIOBE* has the shortest time on both computation and analysis (shown as T_1 and T_2, respectively). With *NIOBE*, the resource utilization on both nodes is improved. The time waiting for data is significantly reduced on both clusters (denoted by smaller gaps between processing of data from each time step). Resources can be returned to the system and allocated to application processes of other scientific workloads.

IV. Evaluation

A. Experimental Setup

Testbed: We have implemented and evaluated NIOBE on the Ares cluster [31] at Illinois Institute of Technology. To emulate a real supercomputer equipped with I/O forwarders and/or Burst Buffers, we leveraged the topology of the network and configured the cluster as follows: 28 compute nodes as clients running HPC and BD applications, and four compute nodes as the I/O forwarding or Burst Buffer nodes. The mapping between compute nodes and I/O nodes is set to seven (i.e., I/O requests from every seven compute nodes get aggregated to one I/O node). We use MPICH 3.2.1 as the MPI library for HPC applications. We run OrangeFS 2.9.7 as the PFS and Redis 3.2.13 for KVS on 16 storage nodes. We chose these systems as representatives of a PFS and KVS architecture due to their popularity, ease-of-use, extended documentation, and their high performance in their respective ecosystems.

Workload: We use synthetic benchmarks to evaluate the pure I/O performance of NIOBE. Each test case reads and writes 128MB of data per process, which leads to a total of 140GB data size for the largest scale. We also use two scientific workflows, CM1 [32] and *Weather Research and Forecasting* (WRF) [33], to validate the benefit of NIOBE in terms of end-to-end performance. Up to 89GB and 26GB of

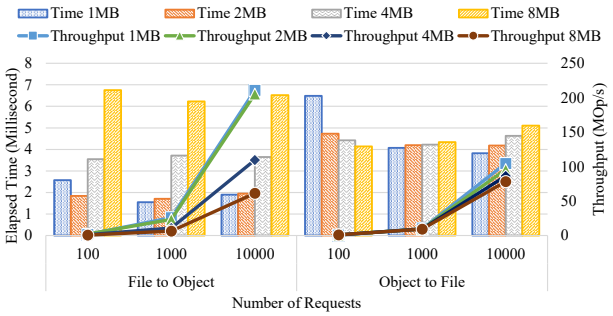


Fig. 3: Mapping overhead of NIOBE.

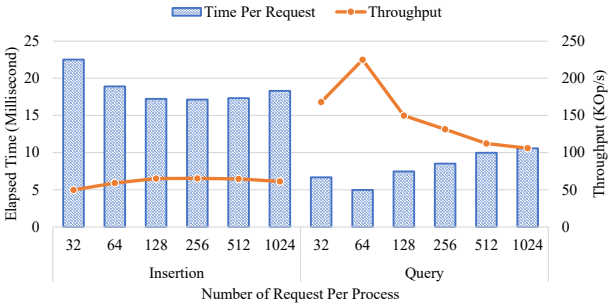
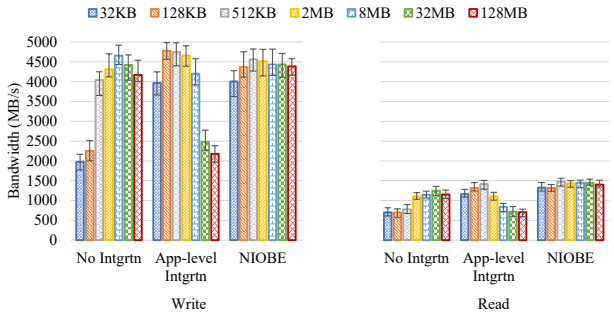
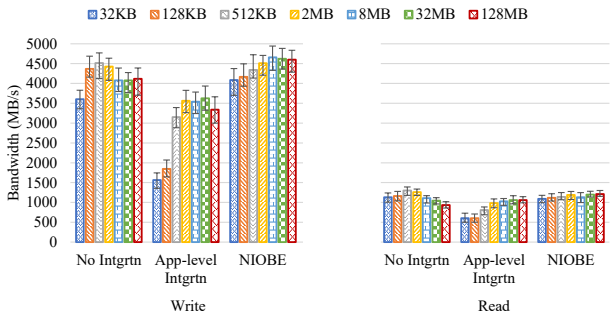


Fig. 4: Metadata overhead of NIOBE.



(a) File I/O



(b) Object I/O

Fig. 5: Synthetic I/O performance comparison of NIOBE and existing solutions.

data is produced and consumed in each iteration of CM1 and WRF workflow, respectively. We flush the page cache every time before the run to avoid OS caching. All the results come from an average of five runs to eliminate OS noise.

B. Overhead Evaluation

The data integration in NIOBE is achieved by mapping data access from one environment to another. Therefore, we test the overhead by measuring only the mapping time.

Figure 3 shows the mapping overhead per I/O request within NIOBE. Both file-to-object and object-to-file mappings are evaluated for HPC I/O and BD I/O, respectively. We supply NIOBE with multiple numbers of I/O requests, namely 100, 1000 and 10000 requests. They are mapped to the other storage system by the Mapping Engine. From the figure, we observe that the mapping time stays stable as the number of requests increases for both mapping directions. All the mappings are finished within 7ms. As discussed in Section III, the mapping is implemented using a hash function to map tuples of file name, offset, and request size into keys. The mapping has a time complexity of $O(1)$, hence is not dependent on the request size. That leads to fast and stable mapping time shown in Figure 3. We additionally calculate the throughput (defined by the number of mappings per second). As the number of requests increases, the throughput keeps increasing and reaches 200K mappings per second at 10000 requests.

Metadata is another critical internal component for NIOBE to sustain high throughput. The Metadata Manager keeps record of mapped I/O and retrieves the mapping when consequent requests upon the mapped data come to NIOBE. Figure 4 demonstrates the cost of metadata operations per I/O request. As we can observe, the average metadata operation time per request is about the same for all test cases. Due to the bucket-based design of metadata structure, which is similar to a distributed hashmap [34], query is faster than insertion. The average metadata operation time tends to increase as the number of requests increases (e.g., more than 512 requests per process for insertion and more than 64 requests per process for query). The aggregate throughput is quite stable for insertion operation which indicates NIOBE can provide stable performance for adding new metadata entries in its Metadata Manager. For the throughput of metadata query, which is the most dominant operation, NIOBE achieves up to over 200K operations/second to guarantee good performance for applications to query metadata of integrated data.

C. Synthetic I/O Performance Evaluation

As a native storage solution for HPC and BD environments, PFS and KVS can provide the best performance for representative I/O workloads in each environment. In NIOBE, mapping I/O requests between different storage systems sacrifices the native data access for better overall performance since the data will be accessed using another semantics. As a result, we evaluate this trade-off using synthetic workloads.

The file and object I/O performance of three different solutions are compared in Figure 5 for write and read, respectively. It is easy to observe that read performance is lower than write performance since the page cache is flushed before each run. Default configuration is used in both OrangeFS and Redis, which utilizes memory to buffer write accesses. In these tests, we use all the cores, which leads to 1120 application processes. For all the following figures, *No Intgrtn* stands for the default solution for systems with data aggregation

services, such as I/O forwarding layer and Burst Buffers. *App-level Intgrtn* is short for Application-level Integration, which represents existing integration solutions in compute nodes, such as IRIS. All the performance results are presented as bandwidths. They are measured from the clients of the storage system which issue the I/O requests. For example, bandwidths are calculated by dividing total data size over the time of writing from aggregation service processes and application processes to the storage systems for solutions with and without data aggregation services utilized, respectively. Overlapping between computation and I/O is not presented here. Workflow-related performance can be found in Section IV-D.

File I/O: As we can see from Figure 5a, *No Intgrtn* shows the best performance for large file write I/O as it has a highly optimized I/O stack for HPC workloads performing large write accesses. For small write accesses (i.e., write accesses smaller than 512KB), *No Intgrtn* is relatively slower, which can be attributed to the well-known low efficiency of smaller accesses on hard drives in our testbed. *App-level Intgrtn* and *NIOBE* map the file OrangeFS writes to put operations to Redis, which significantly improves the write performance as KVS is less sensitive to small accesses compared with PFS. Since the total data size for all test cases is the same (i.e., 128MB per application process), larger request size leads to less I/O concurrency. For large write accesses (i.e., write accesses larger than 8MB), *App-level Intgrtn* slows down as the total I/O bandwidth cannot be saturated by low I/O concurrency. *NIOBE* avoids the decline due to its better mapping policy on the aggregation nodes. For large write accesses, *NIOBE* splits the requests into multiple smaller ones to maintain proper I/O concurrency. Thus, *NIOBE* is less sensitive to request size variation from applications. *NIOBE* is 78% faster than *No Intgrtn* for write accesses smaller than 512KB, and 73% faster than *App-level Intgrtn* for write accesses larger than 8MB. The same pattern applies to the read accesses.

Object I/O: For object I/O, the same amount of data is generated and consumed as the file I/O tests. It is clear that *NIOBE* has the best overall write performance compared with the other two solutions. Comparing with *App-level Intgrtn* for file I/O, the performance of *No Intgrtn* does not drop since PFS favors large write accesses after the requests are mapped. *NIOBE* has 125% performance improvement over *App-level Intgrtn* for small writes (i.e., smaller than 512KB) due to larger request size in *NIOBE* after aggregation, which PFS prefers. Results for read test is analogous to write results.

Summary: Compared with *No Intgrtn* and *App-level Intgrtn*, *NIOBE* provides the most stable performance for various request sizes. It utilizes the benefits of both file and object I/O to sustain the performance when the request size is away from the optimal ones. *NIOBE* utilizes data aggregation services to avoid small requests. Meanwhile, large requests are divided to preserve I/O concurrency for good performance in *NIOBE*.

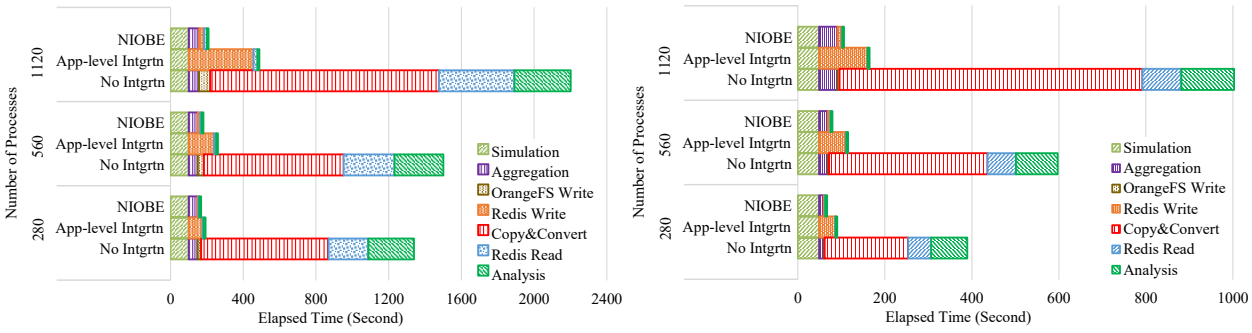
D. Workflow Performance Evaluation

NIOBE targets integrated scientific workflows. It utilizes the benefits of both data aggregation and integration to provide the optimal overall performance. We use two integrated workflows of real applications to evaluate the performance of NIOBE. Firstly, the CM1 simulation generates data which is analyzed by a K-means application expecting data from Redis. Secondly, a pair of WRF simulation and analysis applications are deployed with the same data dependency in-between. The number of iterations is set to ten. A total of up to 1120 processes are involved in both workflows.

Figure 6 shows the breakdown of the total execution time of two workflows for all three solutions. The total execution time is presented as wall time of the entire workflow starting from the first iteration of HPC simulation to the last iteration of data analysis. The total execution time is broken down into each phases. Since phases are overlapped with each other on all the nodes, only the time that is not overlapped with any processing on compute nodes is calculated. We present the results in this way to highlight the time that the workflow is prolonged due to the operations which unnecessarily fall behind the processing on compute nodes. The less time it takes, the sooner users can get the results, which consequently leads to better system utilization due to less idle time on compute nodes.

Similar to the timeline in Figure 2, *NIOBE* is significantly faster than *App-level Intgrtn* and *No Intgrtn*. The total execution time of *NIOBE* is 7.9x, 8.3x and 10.5x shorter than *No Intgrtn* for 280, 560 and 1120 processes in the CM1 workflow, respectively. The speedups come to 5.9x, 6.7x and 9.3x for the WRF workflow. The aggregation time is unique and the same to *NIOBE* and *No Intgrtn*. *Copy&Convert* dominates and significantly prolongs the execution time of *No Intgrtn*. It is interesting to notice that *No Intgrtn* is explicitly optimized in our evaluations. In reality, the performance could be worse due to three reasons. 1) The data copy and conversion are carried out in parallel to be the optimal by our effort. It is much faster than a simple *cp* or *rsync* between two storage systems. All compute nodes are involved in data access and format conversion. However, it requires much more programming experience, which puts a big burden on the users. 2) The number of iterations is relatively small. The phase of *Copy&Convert* is too long to be overlapped with the simulation. It also heavily delays the start of the analysis of the data generated in each iteration, which results in the non-overlapped KVS I/O (*Redis Read*) and analysis computation (*Analysis*) in our tests. 3) *No Intgrtn* is implemented with the assumption of an iteration-based data analysis application in Figure 6. In practice, scientists often analyze the data in an offline fashion. They wait for the completion of the entire HPC application and readiness of all the data before they start the analysis. No overlapping is utilized to reduce the total execution time.

When compared with *App-level Intgrtn*, *NIOBE* speeds up the workflow by 14%, 44% and 233% for 280, 560 and 1120 processes for the CM1 workflow, respectively. For the WRF workflow, *NIOBE* reduces the execution time by 51%, 45% and 69% for different scales, respectively. *App-level Intgrtn* avoids the aggregation time by writing to the KVS directly, which is represented by *Redis Write* in Figure 6. The write operation is noticeably longer than that in *NIOBE* because



(a) CM1 (b) WRF

Fig. 6: Execution time breakdown of CM1 and WRF workflow for NIOBE and existing solutions. (Only execution time that cannot be overlapped with operations on compute nodes are presented.)

it is done synchoronously on compute nodes, which cannot be overlapped by computing on compute nodes as *NIOBE*. It utilizes the existing data aggregation service to enable asynchronous I/O with the cost of the data aggregation time (*Aggregation* in the figure). According to the evaluation results, the cost pays off to provide a better overall performance. The overlapping benefit and integrated I/O gives NIOBE the capability to significantly accelerate the integrated scientific workflow.

V. Related Work

In the HPC community, data aggregation has been proposed to reduce the pressure on external storage. Two-phase I/O [35] aggregates small requests via a network communication phase and carries out larger batches of contiguous I/O to the PFS. Liao et al. proposed an improved collective I/O to consider the locking mechanism in the underlying file system [36]. Tessier et al. developed the TAPIOCA framework to adaptively aggregate I/O with a network topology-aware design [37]. Following the development of SSD, He et al. developed a heterogeneity-aware collective I/O to optimize for heterogeneous PFS [38]. Most importantly, I/O forwarding layer [18] has been proposed to alleviate the stress to PFS by aggregating I/O requests from all compute nodes to dedicated I/O nodes. As a result, the number of I/O clients that PFS serves reduces. The request size is increased to favor hard drive-based PFS. Burst Buffers [21] was designed to buffer data in fast local storage. Cray's DataWarp [39] technology is an example of such service in production systems such as Cori [28]. Data staging services are consequently introduced to utilize the I/O nodes. Datastager [19] and Dataspace [20] provide interfaces to allow better scheduling of staging in and out data.

All these works only target HPC applications without addressing the general need for data integration. NIOBE enhances the data aggregation services to better support integrated data access for extreme-scale computing.

Object stores have been explored in HPC due to its advantage in scalability and performance in access latency. However, no production HPC system is using object stores as the main storage solution for its native workloads rather to enhance the current PFS. Solutions such as IndexFS [40], BatchFS [41], and FusionFS [42] aim to solve the scalability issue of metadata operations by offloading them to an object store. In contrast, MarFS [43] and Ceph [17] propose to use object store as the data storage while maintaining POSIX-compatible interface. In MarFS and Ceph, object store replaces the traditional PFS in HPC and becomes the only storage system. However, access to an object store natively from a scientific application is still required and NIOBE fills this need.

Scientific applications generate large volumes of data only from which scientists can gain useful insights. BD frameworks are proven to be extremely capable in handling huge datasets. As mentioned in Section II, IBM has built a Hadoop connector to allow BD applications to communicate with GPFS directly [12]. Multiple HDFS connectors have been similarly developed for Lustre [13], [44]. However, these solutions are either proprietary or designed for a specific PFS restricting the flexibility, programmability, and transparency that NIOBE offers.

NIOBE bridges multiple storage systems to provide the best storage service for an integrated workflow which needs services from these systems. No one storage system is the winner, but each is great for its targeted workloads. The benefit of having transparent access to any storage is where NIOBE shines.

VI. Conclusions and Future Work

Combining the power of HPC and BD is becoming increasingly interesting for scientists to gain insights in a more efficient way. The inherent difference between data access API and semantics brings an obstacle to an integrated scientific workflow system, which involves applications from both environments. NIOBE tackles this issue by enabling integrated I/O on existing data aggregation hardware and services to maximize the overlapping between computation and I/O to accelerate the workflow. With NIOBE, these workflows can be accelerated by up to 10.5x compared with traditional solutions involving data movements due to optimized I/O. When compared with other state-of-the-art integration solutions, NIOBE can also provide a speedup of up to 133% due to overlapped I/O and data aggregation. In the future, we plan to extend NIOBE to support more data aggregation services and storage systems to test it to larger scales and with more workflows.

REFERENCES

[1] R. L. Rob Ross Rajeev Thakur,, Marc Unangst, and B. Welch, "Parallel I/O in Practice," in *Supercomputing*, 2009.

[2] M. Folk, A. Cheng, and K. Yates, "Hdf5: A File Format and I/O Library for High Performance Computing Applications," in *Proceedings of Supercomputing*, 1999.

[3] R. Rew and G. Davis, "NetCDF: an interface for scientific data access," *IEEE Computer Graphics and Applications*, vol. 10, pp. 76–82, jul 1990.

[4] Apache Software Foundation, "Apache Hadoop." http://hadoop.apache.org/, 2011. Accessed: 2017-07-13.

[5] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark : Cluster Computing with Working Sets," *HotCloud'10 Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, p. 10, 2010.

[6] N. Chaimov, A. Malony, S. Canon, C. Iancu, K. Z. Ibrahim, and J. Srinivasan, "Scaling Spark on HPC Systems," in *Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing - HPDC '16*, (New York, New York, USA), pp. 97–110, ACM Press, 2016.

[7] D. A. Reed and J. Dongarra, "Exascale computing and big data," *Communications of the ACM*, vol. 58, pp. 56–68, jun 2015.

[8] S. Zhou, B. H. Van Aartsen, and T. L. Clune, "A lightweight scalable i/o utility for optimizing high-end computing applications," in *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, (Miami, FL, USA), pp. 1–7, IEEE, 2008.

[9] E. Salmon, "Evolving Storage and Cyber Infrastructure at the NASA Center for Climate Simulation," in *Mass Storage Systems and Technologies (MSST), 2017 IEEE 33th Symposium on*, (Santa Clara, CA), 2017.

[10] D. Tiwari, S. Boboila, S. S. Vazhkudai, Y. Kim, X. Ma, P. J. Desnoyers, and Y. Solihin, "Active Flash: Towards Energy-Efficient, In-Situ Data Analytics on Extreme-Scale Machines," in *Proceedings of the 11th USENIX Conference on File and Storage Technologies (FAST'13)*, 2013.

[11] F. Schmuck and R. Haskin, "GPFS: A shared-disk File System for Large Computing Clusters," in *Proceedings of the 1st USENIX Conference on File and Storage Technologies*, pp. 231–244, 2002.

[12] IBM, "HDFS Transparency." https://www.ibm.com/support/knowledgecenter/en/STXKQY_4.2.1/com.ibm.spectrum.scale.v4r21.doc/bl1adv_Overview.htm, 2012. Accessed: 2018-04-10.

[13] Seagate, "Diskless Hadoop 2 (YARN) on Lustre." https://github.com/Seagate/hadoop-on-lustre2. Accessed: 2017-09-10.

[14] W. Tantisiriroj, S. W. Son, S. Patil, S. J. Lang, G. Gibson, R. B. Ross, S. Seung Woo, S. J. Lang, and R. B. Ross, "On the duality of data-intensive file system design: Reconciling HDFS and PVFS," in *2011 International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pp. 1–12, 2011.

[15] A. Kougkas, H. Devarajan, and X.-H. Sun, "IRIS: I/O Redirection via Integrated Storage," in *Proceedings of the 2018 International Conference on Supercomputing - ICS '18*, (New York, New York, USA), pp. 33–42, ACM Press, 2018.

[16] K. Feng, X.-H. Sun, X. Yang, and S. Zhou, "SciDP: Support HPC and Big Data Applications via Integrated Scientific Data Processing," in *2018 IEEE International Conference on Cluster Computing (CLUSTER)*, pp. 114–123, IEEE, sep 2018.

[17] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. E. Long, and C. Maltzahn, "Ceph: A Scalable, High-performance Distributed File System," in *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, pp. 307–320, 2006.

[18] K. Iskra, J. W. Romein, K. Yoshii, and P. Beckman, "ZOID: I/O-forwarding infrastructure for petascale architectures," in *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming - PPoPP '08*, (New York, New York, USA), p. 153, ACM Press, 2008.

[19] H. Abbasi, M. Wolf, G. Eisenhauer, S. Klasky, K. Schwan, and F. Zheng, "DataStager: Scalable Data Staging Services for Petascale Applications," in *Proceedings of the 18th ACM international symposium on High performance distributed computing - HPDC '09*, (New York, New York, USA), p. 39, ACM Press, 2009.

[20] C. Docan, M. Parashar, and S. Klasky, "DataSpaces: An Interaction and Coordination Framework for Ccoupled Simulation Workflows," in *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing - HPDC '10*, (New York, New York, USA), p. 25, ACM Press, jun 2010.

[21] N. Liu, J. Cope, P. Carns, C. Carothers, R. Ross, G. Grider, A. Crume, and C. Maltzahn, "On the role of burst buffers in leadership-class storage systems," in *2012 IEEE 28th Symposium on Mass Storage Systems and Technologies (MSST)*, pp. 1–11, IEEE, apr 2012.

[22] Y. Chen, C. Chen, X.-H. Sun, W. D. Gropp, and R. Thakur, "A Decoupled Execution Paradigm for Data-Intensive High-End Computing," in *2012 IEEE International Conference on Cluster Computing*, pp. 200–208, IEEE, sep 2012.

[23] Oak Ridge National Lab, "Leadership Computing Requirements for Computational Science." https://www.olcf.ornl.gov/wp-content/uploads/2010/03/ORNL_TM-2007_44.pdf. Accessed: 2018-08-27.

[24] Apache Software Foundation, "Design - Apache Hive," 2015.

[25] Intel, "Intel Enterprise Edition for Lustre Software." http://www.comnetco.com/wp-content/uploads/2015/01/ieel_product_brief.pdf, 2015.

[26] N. Ali, P. Carns, K. Iskra, D. Kimpe, S. Lang, R. Latham, R. Ross, L. Ward, and P. Sadayappan, "Scalable I/O forwarding framework for high-performance computing systems," in *2009 IEEE International Conference on Cluster Computing and Workshops*, pp. 1–10, IEEE, 2009.

[27] H. Abbasi, M. Wolf, G. Eisenhauer, S. Klasky, K. Schwan, and F. Zheng, "DataStager: scalable data staging services for petascale applications," *Cluster Computing*, vol. 13, pp. 277–290, sep 2010.

[28] NERSC, "Cori." https://www.nersc.gov/users/computational-systems/cori/. Accessed: 2019-04-30.

[29] A. Kougkas, H. Eslami, X.-H. Sun, R. Thakur, and W. Gropp, "Rethinking key–value store for parallel i/o optimization," *The International Journal of High Performance Computing Applications*, vol. 31, no. 4, pp. 335–356, 2017.

[30] Y. Yin, S. Byna, H. Song, X.-H. Sun, and R. Thakur, "Boosting Application-Specific Parallel I/O Optimization Using IOSIG," in *2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (ccgrid 2012)*, pp. 196–203, IEEE, may 2012.

[31] "Ares cluster." http://www.cs.iit.edu/{~}scs/resources.html{#}content6-8p, 2019. Accessed: 2019-04-24.

[32] G. H. Bryan, "CM1 Homepage." http://www2.mmm.ucar.edu/people/bryan/cm1/, 2019. Accessed: 2019-04-24.

[33] UCAR, "About the Weather Research & Forecasting Model." http://www2.mmm.ucar.edu/wrf/index.php, 2019. Accessed: 2019-04-24.

[34] "Basket." https://github.com/hariharan-devarajan/basket. Accessed: 2019-04-24.

[35] R. Thakur, W. Gropp, and E. Lusk, "Data Sieving and Collective I/O in ROMIO," in *The Seventh Symposium on the Frontiers of Massively Parallel Computation*, pp. 182–189, 1999.

[36] W.-k. Liao and A. Choudhary, "Dynamically adapting file domain partitioning methods for collective I/O based on underlying parallel file system locking protocols," in *High Performance Computing, Networking, Storage and Analysis, 2008. SC 2008. International Conference for*, pp. 1–12, IEEE, 2008.

[37] F. Tessier, V. Vishwanath, and E. Jeannot, "TAPIOCA: An I/O Library for Optimized Topology-Aware Data Aggregation on Large-Scale Supercomputers," in *2017 IEEE International Conference on Cluster Computing (CLUSTER)*, vol. 2017-Septe, pp. 70–80, IEEE, sep 2017.

[38] S. He, Y. Wang, X.-H. Sun, C. Huang, and C. Xu, "Heterogeneity-Aware Collective I/O for Parallel I/O Systems with Hybrid HDD/SSD Servers," *IEEE Transactions on Computers*, vol. 9340, no. c, pp. 1–1, 2016.

[39] Cray, "DataWarp Cray DataWarp Applications I/O Accelerator." https://www.cray.com/products/storage/datawarp. Accessed: 2019-05-18.

[40] K. Ren, Q. Zheng, S. Patil, and G. Gibson, "IndexFS: Scaling File System Metadata Performance with Stateless Caching and Bulk Insertion," in *SC14: International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 237–248, IEEE, nov 2014.

[41] Q. Zheng, K. Ren, and G. Gibson, "BatchFS: Scaling the File System Control Plane with Client-Funded Metadata Servers," in *2014 9th Parallel Data Storage Workshop*, pp. 1–6, IEEE, nov 2014.

[42] I. R. Dongfang Zhao, Chen Shou, Zhao Zhang, Iman Sadooghi, Xiaobing Zhou, Tonglin Li, "FusionFS: a distributed file system for large scale data-intensive computing," in *2nd Greater Chicago Area System Research Workshop (GCASR)*, 2013.

[43] J. Inman, W. Vining, G. Ransom, and G. Grider, "MarFS, a Near-POSIX Interface to Cloud Objects," *USENIX ;Login:*, vol. 42, no. 1, pp. 26–31, 2017.

[44] Intel, "Hadoop Adapter for Lustre (HAL)." https://github.com/intel-hpdd/lustre-connector-for-hadoop. Accessed: 2017-09-10.