PortHadoop: Support Direct HPC Data Processing in Hadoop

Xi Yang*, Ning Liu*, Bo Feng*, Xian-He Sun* and Shujia Zhou†
*Department of Computer Science
Illinois Institute of Technology, Chicago, IL
{xyang34, nliu8, bfeng5}@hawk.iit.edu, sun@iit.edu
†Northrop Grumman Corporation, USA
shujia.zhou@ngc.com

***Abstract*—The success of the Hadoop MapReduce programming model has greatly propelled research in big data analytics. In recent years, there is a growing interest in the High Performance Computing (HPC) community to use Hadoop-based tools for processing scientific data. This interest is due to the facts that data movement becomes prohibitively expensive, high-performance data analytic becomes an important part of HPC, and Hadoop-based tools can perform large-scale data processing in a time and budget efficient manner. In this study, we propose PortHadoop, an enhanced Hadoop architecture that enables MapReduce applications reading data directly from HPC parallel file systems (PFS). PortHadoop saves HDFS storage space, and, more importantly, avoids the otherwise costly data copying. PortHadoop keeps all the semantics in the original Hadoop system and PFS. Therefore, Hadoop MapReduce applications can run on PortHadoop without code change except that the input file location is in PFS rather than HDFS. Our experimental results show that PortHadoop can operate effectively and efficiently with the PVFS2 and Ceph file systems.**

***Keywords*—Hadoop; MapReduce; I/O; Data Analysis**

I. Introduction

Hadoop MapReduce has been a popular framework for distributed large-scale data processing [1] [2]. Hadoop MapReduce handles fault tolerance in a transparent manner and can achieve embarrassingly parallelism on commodity devices. Its programming model is easy to understand and use, which has significantly lowered the bar of large-scale data processing.

In recent years, researchers have proposed the concept of "HPC in the Cloud" [3] [4] [5] and various programming models and platforms to address big data challenges in data center [6] [7] [8]. The motivation behind "HPC in the Cloud" is twofold. Firstly, the growth of computing capacity of the HPC systems has followed Moore's law in the past few decades, and this trend will continue in the near future. On one the hand, the increasing of computing capacity has enhanced the abilities of scientific discoveries. On the other hand, those empowered scientific applications have also generated skyrocketing data, putting even more pressure on the ever-overloaded file systems. The development of storage systems has severely lagged behind its computing counterpart, in terms of both capacity and accessing speed, thus limiting the overall performance of HPC systems. Offloading the onerous data-intensive tasks to a cheaper and yet powerful system like Hadoop MapReduce system is one viable approach to alleviate the burden on HPC storage system. Secondly, Hadoop MapReduce has gained wide success in many fields over the past few years. Utilizing Hadoop for scientific application data management and processing becomes convenient as well as productive.

Typical tasks of an HPC storage system fall into two categories: (1) Input/Output from a computing system; (2) query and data processing requests from external users. Traditionally, task (1) has been the major concern of HPC, and the design and optimization of HPC systems have focused on achieving high-performance I/O rate and dealing with irregular and bursty I/O requests [9] [10]. Parallel File Systems (PFS), e.g., PVFS2 [11], GPFS [12], CephFS [13], Lustre [14] etc, are developed for the purpose of solving task (1). While task (2) is becoming more and more an integrated part of HPC, PFS are not designed to handle it. The query and data processing tasks often take a large portion of the performance time and impair the I/O performance of running applications. In recognizing the problem, there is a growing interest in using big data processing frameworks such as Hadoop MapReduce for large-volume data query and processing. Another reason for this interest, of course, is that many data processing and analytic tools/software are developed under Hadoop MapReduce environments.

Hadoop MapReduce manages its data through the Hadoop Distributed File System (HDFS). Unlike PFS, HDFS releases data consistency and consequently has a better data concurrency than PFS. It is extremely effective for applications that do not require strong consistency, such as Internet search, information retrieval, and visualization, etc. For example, in the NASA Center for Climate Simulation (NCCS), climate and weather simulations can create a few Terabyte simulation datasets [15]. Only the data items with interesting events such as hurricane center and thunderstorms, which could be one order of magnitude smaller than the original data size, are further analyzed. To visualize the reduced, but still huge, data effectively, a visualization tool has been developed under Hadoop environment at NCCS. Hence, it is highly desirable to have a reading tool that is able to read a subset of data in PFS to HDFS efficiently for visualization. In fact, this cross-platform reading tool is necessary and essential for on-time visualization for NASA climate and weather simulation results.

A straightforward way to use Hadoop MapReduce is to copy data from PFS to HDFS. However, this approach has several issues. The first design challenge is to maintain the efficiency of the data storage. Typically, PFS keeps one copy of the data, and HDFS keeps multiple copies for high availability and fault tolerance. This multiple copying is costly and unnec-

978-1-4799-9926-2/15/$31.00 ©2015 IEEE

essary for data already available in PFS. The second design challenge is the performance of data movement. HDFS does not support concurrent write access. Thus, data blocks have to be first aggregated into an assigned aggregator and then saved onto HDFS. This copy procedure is very time-consuming, and the users have to experience a long, unpredictable delay before accessing the data in HDFS. The biggest challenge yet is to reconcile the semantic gap between the two file systems. This semantic gap is not just in terms of interface, but the optimization and organization under its semantic interface. For instance, In handling fault tolerance, PFS relies on application checkpointing and restart while HDFS relies on data replication and task rescheduling. Thus, their scalability has to be achieved with two different philosophies. There are numerous community efforts trying to reconcile the I/O middleware layer such as the MR-MPI [16]. Such efforts usually involve several trade-offs in design and deployment and sacrifice performance and functionalities. Therefore, it is extremely challenging to develop a full-fledged file system that is well accepted by both communities.

After considering those issues, we propose PortHadoop, a portable Hadoop framework that supports direct data access to PFS from a Hadoop environment. PortHadoop supports multiple PFS including PVFS2 and CephFS. PortHadoop does not intend to reconcile PFS and HDFS. Instead, it supports direct data access between these two file systems. PortHadoop reads PFS data directly from a (remote) PFS and stores the data onto PortHadoop memory system. This data path and its management involve a significant change in the current Hadoop implementation, and we will elaborate them in detail in Section II. The benefit of the PortHadoop design is significant: both the interface of PFS and HDFS are unchanged. Thus, the HPC application code does not need any change, and can still take the full advantages provided by PFS, e.g. enhancing application performance through maximizing the parallelism and I/O bandwidth. In the meantime, there is no need to change the Hadoop applications as well. In Figure 1, we illustrate the design behind PortHadoop. Here, the lines labeled with "a" represent the data path of the naive approach, and the lines labeled with "b" represent the data path of PortHadoop. The naive approach copies the entire data onto HDFS and then, after copying, read data for post-processing. PortHadoop, read data directly to Hadoop memory, without copying from PFS to HDFS first, for data processing. Since PortHadoop reads data directly for processing, it has avoided the multiple copying issues onto HDFS. Also, since it reads directly into the memory, it further can pipeline the data processing and data transfer. As illustrated in Figure 1, the advantage of PortHadoop is tremendous in that it avoids the entire copy and read on HDFS, and only reads the needed data to Hadoop memory system directly.

The contributions of this paper are summarized as follows:

- PortHadoop has built an efficient mechanism for processing data stored in PFS in a Hadoop environment via direct data access;
- PortHadoop is portable to various PFS. The current PortHadoop implementation works effectively and efficiently on PVFS2 and CephFS;
- PortHadoop keeps the semantics of native Hadoop and PFS. Therefore, existing Hadoop applications can run directly on PortHadoop, and HPC application codes do not need any modification.

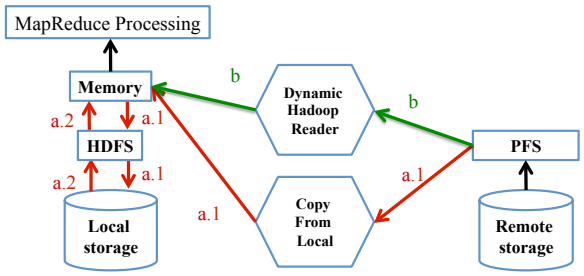


Fig. 1. Data paths for supporting data processing in Hadoop. The data path of the naive approach is labeled as a, and colored in red while the data path of PortHadoop is labeled as b, and colored in green.

The rest of the paper is organized as follows. Section II mainly focuses on the design and implementation of PortHadoop. The experimental evaluation of PortHadoop is presented in Section III. The related work is discussed in Section IV. Finally, we conclude the paper in Section V.

II. Design and Implementation

A. Overview

PortHadoop is an extension of the Hadoop framework. Our current implementation of PortHadoop is based on Hadoop version 1.0.0, which is a stable version and widely deployed. Our target application is the NASA climate modeling application, which is implemented under Hadoop version 1. It serves our purpose well. Cross-platform data access does not involve advanced features, such as advanced resource allocation for multi-tenancy, provided by Apache YARN. HDFS has not changed much from Hadoop version 1 to YARN, except some additional features. PortHadoop can be extended to YARN with little effort. PortHadoop supports the native MapReduce workloads and also enables map tasks to read directly from an external PFS. Our work is portable in the sense that it is compatible with multiple PFS.

To achieve our design goal, we have modified multiple existing components in Hadoop system and created some new components. We introduce the communication and data transfer protocol between Hadoop and PFS in Subsection II-B, introduce the virtual block concept for cross-platform data processing in Subsection II-C, present the task scheduling support in Subsection II-D. We discuss the PortHadoop data alignment and data integrity design in Subsection II-E and propose data prefetching mechanism to further utilize the bandwidth between PFS and Hadoop in Subsection II-F. Finally, fault PortHadoop tolerant mechanisms are discussed in Subsection II-G. Figure 2 illustrates the PortHadoop system architecture and its corresponding job and task events. A typical MapReduce job splits data into multiple independent chunks that are processed by multiple map tasks in a parallel manner. PortHadoop intercepts data-read from map tasks by mapping the virtual blocks in HDFS to real blocks in remote PFS. PortHadoop manages the virtual blocks in a virtual block table in the name node. A job starts with mapping the virtual blocks to PFS files. This step is crucial as PortHadoop uses

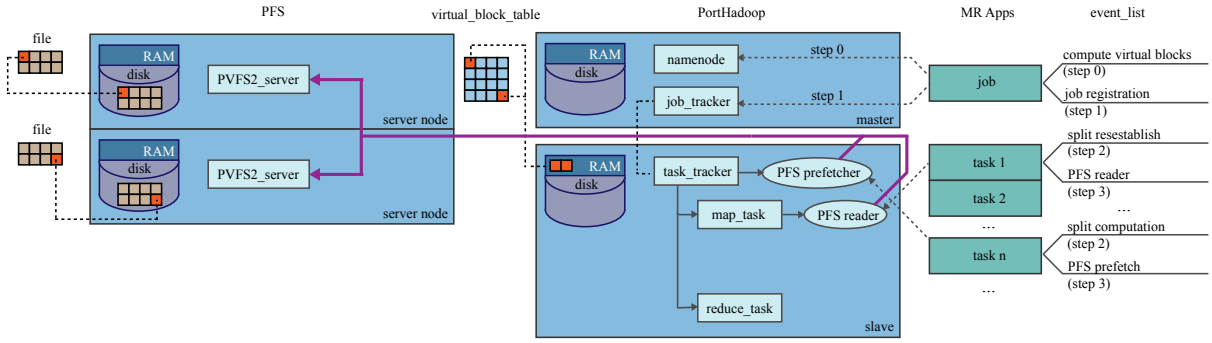


Fig. 2. PortHadoop System Architecture: We take PVFS2 as an exmaple for PFS, PortHadoop job_tracker and task_tracker run on master node, and task_tracker and map_task run on PortHadoop slave node. (There are many slaves, take one as example here.) Virtual blocks are managed in the virtual_block_table in namenode. Task_tracker initiates the PFS_prefetcher thread, which is responsible for prefetching data directly from remote PFS. Map_task initiates the PFS_reader thread, which reads data directly from remote PFS.

this block metadata information for data block retrieval from the remote PFS. In Figure 3, we further illustrate the main algorithms used in PortHadoop. The PFS_reader procedure represents the thread spawned by the map task process. They are initiated when the target blocks are not in the virtual block table. Those threads further interact directly with the remote PFS file server. The PFS_prefetcher represents the thread spawn by task tracker process. It communicates with the remote PFS. This thread can utilize the PFS network and prefetch data from the remote PFS to memory, thus achieve the goal of overlapping computation with I/O. Both PFS_reader and PFS_prefetcher fetch the target data into Cached File(CF), and then set this CF available to map task.

B. Portable APIs

A communication protocol between Hadoop and PFS is established via our portable APIs. We implement this protocol according to Message Passing Interfaces (MPI). In the implementation of PortHadoop, we select MPI-IO [17] as the direct access tool for reading data from remote PFS. Thus, the MPI APIs are the standard APIs for PortHadoop. We choose MPI because it meets our three key requirements: 1) efficient data transmission, 2) data integrity, and 3) ease of access to PFS namespace and metadata information.

MPI-IO is widely adopted in many supercomputer systems as the intermediate tool for communicating with the storage system. As the most popular implementation, ROMIO [18] has enabled MPI-IO to work seamlessly with many PFS, including GPFS, PVFS2, Lustre, and CephFS. It is an efficient and compatible data transmission tool to PFS.

Because PFS and HDFS have different views of data, we need to ensure data integrity while transferring data. In PFS, data are stored and accessed in certain types, such as int, long, float, etc. However, HDFS data is unstructured and untyped data, which is Byte oriented. The user defined RecordRead within map function will parse the data as it is expected. Thus, the granularity of data transfer is per byte, which is consistent between the data source and sink. In the implementation of PortHadoop, MPI_CHAR and MPI_BYTE are used as the basic data type for data access. When loading the data in nodes at Hadoop cluster, they are transferred as they were in PFS.

In order to intelligently read data from PFS, PortHadoop uses MPI_File_get_size() function to get the size of the target file. PortHadoop uses this file size as well as the default block size to compute the exact offset on the data for each task in HDFS namespace system. This calculation ensures that PortHadoop provides a consistent semantic as that of HDFS for the upper-level MapReduce applications.

C. Virtual Block

As mentioned in the system overview, virtual blocks are used to build the data mapping between HDFS and the remote PFS. This mechanism is illustrated in Figure 1. PortHadoop bypasses the HDFS layer in the Hadoop environment by placing data directly into memory. It has skipped the file copying process, which is extremely costly. However, the skipping creates a technical hurdle. PortHadoop needs to maintain the integrity of the upper-level MapReduce applications, providing a virtual HDFS environment, even the data is in a remote PFS.

In order to provide the virtual environment, we introduce the concept of "virtual block" in PortHadoop. The management of HDFS is based on a centralized namespace in NameNode. This namespace provides the view of all the files and blocks saved in HDFS and is the base for JobTracker (the first generation of Hadoop) and MapReduceAppMaster (Hadoop YARN) to make scheduling decision. Conventionally, tasks are scheduled to nodes where the input splits or data blocks reside. To make this procedure consistent, we added functionalities in the namespace system of NameNode by providing virtual blocks and files without actually saving the data blocks to the nodes. In other words, virtual blocks only exist in the centralized namespace. The data operation is hidden from the MapReduce application, and a map task triggers the MPI file read procedure and fetches the data from the remote PFS before its Mapper function processes its data. The details about a map task and its processing data are presented in Subsection II-E. The mapping from remote PFS file to virtual blocks is computed by the call of MPI_File_get_size(). The functionalities are provided in a created function named VirtualBlockAllocation(). This function can create the virtual blocks in the namespace, map files to blocks and close the directory. We purposely exclude DataNode in this procedure. Thus, PortHadoop can avoid the I/O operations and data replications that are necessary parts of the original HDFS. In Subsection

VBT : Virtual block table
S : Split
SQ : Split queue
CF : Cached file
PC : Capacity for prefetching data

procedure SCHEDULE_TASK
▷ Enable PortHadoop to work with virtual block. Create VBT for PFS file/files by client.
Job and splits are initialized. SQ is initialiazed.
while true **do** ▷ The master processes RPC from slaves.
if the slave asks for map task **then**
if this slave has split prefefetched in SQ **then**
assign a task with prefetched data to the slave, update SQ
else
assign a map task to the slave, update SQ
end if
end if
if the slave asks for prefetch **then**
assign prefetch task to the slave, update SQ
end if
process other requests
end while
end procedure

procedure POLL_TASKS
while true **do** ▷ RPC from a slave to the scheduler.
poll task when resource avaiable
request prefetch task when PC avaiable
procedure Process_actions
end while
end procedure

procedure PROCESS_ACTIONS
if action is to assign task **then**
initialize Map_task(S)
end if
if action is to prefetch **then**
initialize PFS_Prefetch(S), update PC
end if
process other actions
end procedure

procedure MAP_TASK
compute CF, re-establish split to CF
if CF is not in available **then**
initialize PFS_read(S)
end if
wait until CF is available
conduct Mapper, update PC when cleanup
end procedure

Fig. 3. Main algorithms used in PortHadoop.

II-D, we illustrate these virtual blocks are sufficient for cross-platform task scheduling and execution.

D. Task Scheduling Support

PortHadoop schedules tasks based on virtual blocks to initialize the aligned mapping between map tasks and their

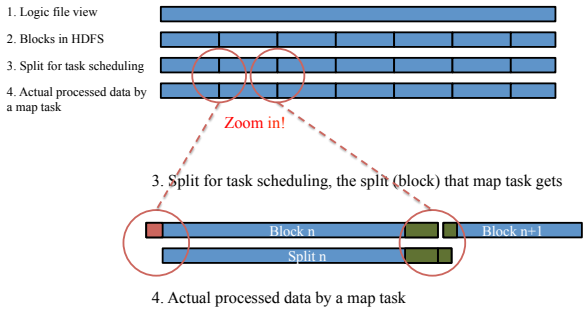
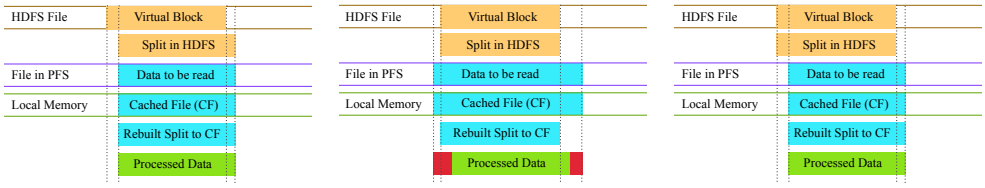


Fig. 4. Cross-block-boundary record reads in Hadoop MapReduce: this issue is transparently handled by RecordReader and HDFS.

target data. The default scheduler in Hadoop schedules tasks based on the data locality information. Tasks are scheduled to the local slave node where the corresponding data block resides. The scheduler only assigns a task onto a remote node when there is no slot or container resource available on a local node. The scheduler assigns each task an input split, which is a reference to an actual data block or consecutive blocks. The actual data block is only retrieved when the task is assigned to the slave node and starts execution. Therefore, PortHadoop intercepts the input split and redirect the data block retrieval from HDFS or local storage system to memory. The centralized namespace provides sufficient information for the scheduler to assign splits to tasks. Thus, we can avoid the communication to DataNode. In fact, we have built an HDFS equivalent in the memory system as introduced in Subsection II-C. The offset and length of the file for each task is identical as they would be in the HDFS. To support this mechanism, PortHadoop has to initiate the necessary MPI function calls to get the file metadata and data from the remote PFS, update the namespace and save the corresponding data to memory system before scheduling the tasks. This procedure will be triggered while the client is adding the input path with a PFS specified prefix, such as "pvfs2://," which indicating the input data are in remote PFS. In implementation, the function is integrated in addInputPath method of FileInputFormat. The hint of PFS will be reserved in the path, to inform the consequent map task to process a remote data split at a specific PFS. An RPC protocol is established between the client and NameNode to establish virtual blocks according to the input data.

E. Split Alignment

It is essential to guarantee data integrity such that map task in PortHadoop can process its target data similar to that of the conventional Hadoop. As discussed in Subsection II-D, a split is a unit of input data for a map task and is a reference to the target data rather than the actual data block. We illustrate the virtual block mechanism in Figure 4. Specifically, a split contains the file name, start and end positions in a file. Each split will be calculated while user submits the job. In general, a contiguous file will be chunked up to multiple splits for processing. A split corresponds to a data block. Because the block size is pre-determined, it is common that a record will cross the boundary of two blocks. In this scenario, a split usually contains incomplete information for the first and the last record. A map function has its RecordReader to read the input data in its expected format. The RecordReader reads



(a) Perfectly-aligned read strategy with predicatable split sizes

(b) Redundant read strategy

(c) Perfectly-aligned read strategy with dynamic split boundary detection

Fig. 5. Read strategies and actual data processing in PortHadoop.

records in a while loop, from the start offset of the split, also the last record in the split if the very record cross block boundary.

For a given split, the data of its beginning part may belong to the task of the previous split. If the start offset in the split is non-zero, which indicates the current split is not the first block, RecordReader will skip that piece of data and adjust the start offset data for appropriate task processing. In a special case, the split is perfectly aligned, the first record in the split, therefore, should be processed by the current map task. In Hadoop, LineRecordReader reads the input data backward one byte from the start offset of assigned split and then skips the first line from a synchronization label, namely sync-marker, such as a new line character. This strategy avoids processing an incomplete record that at the beginning of a given split. HDFS can handle the split alignment automatically if data are copied into HDFS first. The question, then, is how do handle the alignment if we skip the HDFS copying.

PortHadoop aligns the split boundary and guarantees the data integrity provided by HDFS. Before initializing the Mapper function, as Figure 5 shows, the range of target data is parsed from the given split, the fetcher then read the data in PFS and save it to the local memory of the map tasks hosting node. We re-establish the split by pointing the split file at the cached file and setting the start offset the same as the length of sync-marker and keeping the split length remain.

In PortHadoop, the splits are categorized into two classes: the ones with predictable size and patterns, and the ones with unpredictable sizes. The solution to the first category is straightforward. As illustrated in Figure 5(a), we can set the boundary of two input splits at the right place where no records are broken. And the data processed is the same as the assigned split. HDFS uses fixed block size for easy namespace and conducts metadata operations in name node, allowing for customizable block size that is multiples of 1024 bytes. Though Hadoop computes splits based on blocks sizes, we cannot change the block sizes based on record boundaries. Alternatively, we adjust split sizes. In other words, once the split boundaries are calculated, we can allocate the virtual blocks, each of which indicating the approximate amount of data to be processed by a map task, and then set up preferred split sizes accordingly to match boundaries. Therefore, the consequent splits will be perfectly aligned, and there is no cross-block-boundary record read.

For the second category, a straightforward approach is to adopt the default Hadoop mechanism that adjusts the split boundaries during task executions. Recall that the input data are not kept in HDFS, so there is no data locality information in the splits; the splits will be sorted according to their sizes in descent order before task scheduling. Hence, we have to reserve the split dependency during task execution, which is a pointer from the previous block to the following block. Moreover, it is necessary to dynamically re-establish input stream at runtime while the RecordReader reaches the end of the local split file. PortHadoop implementation is based on the second and the third mechanisms.

The second method is to acquire more data than the actual range of the split. We name it redundant-read mechanism, as shown in Figure 5(b). If the current split is not the first one, it reads a unit of synchronized marker length data at the beginning of the split. Also, if the current split is not the last one, it also reads certain redundant bytes to ensure the data integrity of the last record. While the current split is not the first one, a backward read for a sync-marker length data is a must in Hadoop, although it may introduce more latency. Redundant-read in PortHadoop will read all the data once that avoids the latency of backward reading. However, while the record size pattern is unknown, it is hard to predetermine the size of the read. This method suffers uncertainty for both overhead and integrity guarantee for the last record in that split. If the size is too small, then the record may still be incomplete. If the size is too large, the I/O overhead will dominate.

The third one is to detect the split boundary by detecting the sync-marker that is close to block boundary, and then fetch the target data accordingly. We name this method perfectly aligned read with dynamic split boundary detection, as shown in Figure 5(c). Hence, the fetched data are exactly the same as the data to be processed for an assigned split. Different from predictable record patterns, PortHadoop acquires the split boundaries first; instead of creating splits accordingly, it then passes the hints of split range to MPI data reader to fetch the perfectly aligned target data. Therefore, this procedure guarantees both data integrity and I/O efficiency.

In summary, PortHadoop implements three split alignment strategies as shown in Figure 5, to support split alignment. In Figure 5, the yellow rectangle represents a virtual data block in HDFS, and the corresponding split that is assigned to a map task. The light aqua one represents the data mapping from PFS to memory, a cached data file (CF) and its rebuilt split to that CF. The Green ones indicate the actual data to be processed by map tasks. The redundant-read strategy, as shown in Figure 5(b), is efficient in practical because a redundant 2K bytes are small to a 64MB block and sufficient to ensure data integrity. However, it may suffer data integrity or redundant I/O when the record size or synchronized unit is too large in that split. We have two perfectly-aligned read strategies, which are shown

in Figure 5(a) and Figure 5(c), respectively. Both of them have no redundant read for the cached file (CF) and their map tasks process the exact amount of data that cached in memory. The first one prefers predictable split sizes and determines the alignment for data transfer in the split creation phase. The second one has no assumption on the split boundaries, it adjusts the boundaries at runtime, calculates the data (CF) to be transferred in map task runtime.

F. Data Prefetching

As large memory enhances performance and the RAM card becomes cheaper and has larger capacity, it is conservative that PortHadoop uses memory as a staging stack of the input data splits. Map function begins once its input data are cached in memory. Hence, the I/O request time of map task input will be exposed. The causes are two-fold. First, as same as the conventional Hadoop, PortHadoop adopts a poll mechanism for task executions. The slave nodes are periodically communicating with the scheduler, for reporting task status and asking for tasks when the resource is available. Second, the execution time of tasks in PortHadoop is composed of the time for data-fetching and processing manner. Hence, when the bandwidth of both network and PFS disk I/O is underutilized, prefetching target data blocks helps to reduce exposed data access time. However, prefetching more blocks will overflow the capacity of the memory at slave nodes and causes failure. Hence, in each node, we define an upper-limit capacity threshold for these in-memory blocks. One monitoring thread maintains the status of capacity consumption and clean ups the useless data for the killed or failure tasks.

We design a fine-grained pipeline data processing mechanism in PortHadoop, which coordinates data prefetching and task scheduling. Since there is no data locality for all the map tasks, the task will be dispatched in FIFO order in the task queue. JobInProgress is for a job and decoupled with job tracker. We utilize the existing status of tasks and splits information that is kept in JobInProgress. The PortHadoop scheduler uses JobInProgress information to assign map tasks, and prefetching task. The pipelining is conducted in three steps. The first one is the job tracker decides whether to conduct pipelining based on if the job has a potential to overlap the data processing and data fetching. Second, once the job is initialized, the split information is ready, we maintain an array that indicates the split status in three categories: prefetching, corresponding task finished, and not assigned. Different from map task scheduling, the prefetching task will prefetch the split with a state of not assigned, from the rear of the split queue for avoiding redundant fetching data blocks. Consequently, if the current job needs block prefetching, then the job tracker assigns block prefetch tasks to the task trackers. Task tracker is responsible for launching prefetching input data functions. We utilize heartbeat routine between task tracker and job tracker to coordinate the data prefetch and pipelining task execution. We define a new action named PrefetchAction and encapsulate it into actions that assigned split to the task tracker. Each PrefetchAction carries the file name, the split start offset, and the split length for prefetching. We customize TaskTrackerStatus to provide available capacity for prefetching input splits, report to the job tracker. When the scheduler assigns prefetching tasks to a task tracker, it records the name of task tracker and the corresponding split and the state of that split is set to prefetching. Correspondingly, when a task tracker asks for map tasks, the job tracker will look up the assigned prefetching splits and assign corresponding map tasks to that slave.

G. Fault Tolerance Support

Hadoop is popular due to its transparent fault tolerance on commodity machines. As PortHadoop is an extended Hadoop, its map tasks will store their intermediate results onto local file system as the default Hadoop does. Therefore, the fault tolerance feature of reduce phase is inherent in the conventional Hadoop after map tasks have finished their jobs. However, if a map task fails during its execution, the failure task needs to fetch its input data again from the remote PFS. In this case, PortHadoop has designed to reschedule the failure task and re-fetch the corresponding input data. PortHadoop optimizes data transfer by pipelining the transfer and processing prior to alleviating map task skewness and stragglers. We observed the skewnesses in multiple waves of map task processing in Section III. To optimize task processing, PortHadoop also supports backup tasks [19], as that in the conventional Hadoop.

III. Evaluation

In this section, we present the experimental evaluations of PortHadoop and verify its performance and functionalities from a set of tests cases. PortHadoop bridges two domains of file systems and supports multiple types of Hadoop applications. Thus, there are multiple factors which affect PortHadoop performance. We select the most important factors and illustrate the details in the following discussions.

A. File Systems and Platforms

PortHadoop is portable meaning that it works seamlessly with multiple PFSes. The current implementation supports PVFS2 and CephFS. Specifically, we use OrangeFS version 2.8.6 that is the commercial version of PVFS2 and is well maintained by a team from Clemson University. We also used CephFS version 0.80.7 in the experiments. In our test environment, the network connection is one Gigabit Ethernet. To further prove that PortHadoop can operate on multiple platforms, we use the two clusters in IIT SCS Lab: Craysun and HEC. We use 17 nodes in Craysun and 25 nodes in HEC. The head node is a Dell PowerEdge 2850 Server, which has a dual-core Intel Xeon CPU 2.80GHz with 6GB of memory. All slave nodes are Dell PowerEdge SC 1425 Server, each of which has an Intel Xeon CPU 3.40GHz processor, 1GB of memory, and a 36GB (15,000 rpm) SCSI hard drive. HEC is an SUN Fire Linux cluster, in which each node has two AMD Opteron processors, 8GB memory, and a 250GB SEAGATE HDD. All nodes run Ubuntu 14.04 with the latest Linux kernel. The MPI installed in both systems is MPICH 3.0.4.

B. Metadata Operation

In Subsection II-C, we introduced the concept of virtual blocks. The NameNode in PortHadoop manages virtual blocks in a centralized manner similar to managing regular blocks. Intuitively, the overhead for managing the virtual blocks should be small because all operations are within the memory. In Figure 6, we present the overhead under a series of experiments

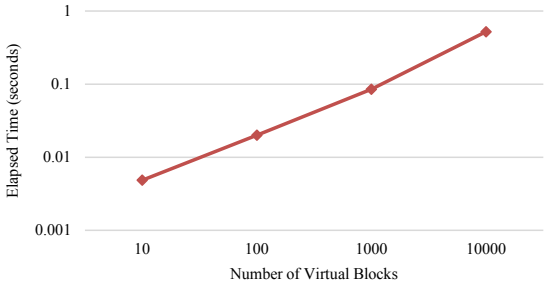


Fig. 6. Duration of virtual block allocation in PortHadoop.

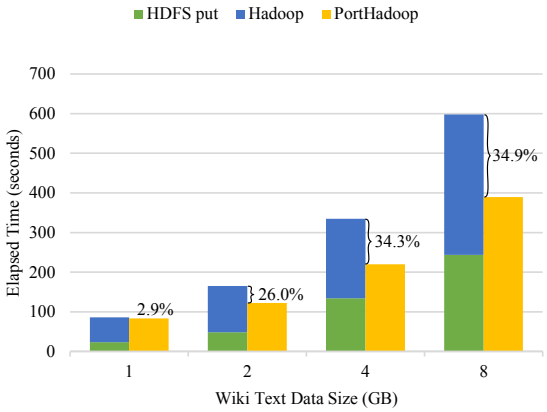


Fig. 7. Performance comparison between conventional Hadoop and PortHadoop with WordCount on wiki text dataset. "HDFS put" copies an input data file from CephFS to HDFS and then "Hadoop" processes the dataset. "PortHadoop" directly reads and processes the dataset residing in CephFS.

on the creation of virtual blocks. The data block size is set as 64MB, and we vary the total number of blocks from 10 to 10,000. As we can see, the total execution time grows linearly with the number of operated virtual blocks. In the largest test case, the virtual blocks for a total of 625 GB of data are allocated in about half a second. The overhead compared to the actual data fetching from the remote PFS is negligible. Thus, in the following experiments, we mainly focus on analyzing the overhead of I/O and task execution.

C. Evaluation on Hadoop Benchmarks

We select Wordcount and Grep as the baseline Hadoop benchmarks for evaluating the performance of PortHadoop. They are representative applications for old fashion (runOldMapper) and new fashion (runNewMapper) MapReduce APIs, respectively. To further show the performance improvement of PortHadoop over the original Hadoop system, we conducted experiments on both Hadoop and PortHadoop. As discussed in Section II, PortHadoop is based on Hadoop version 1.0.0; thus we also use the same version for a fair comparison.

We measure and compare the costs of reading data in the conventional copy-then-read manner and PortHadoop. CephFS is used in this evaluation. We use raw wiki text dataset from Wikipedia.org as the input data. Recall the conventional method has a "copy" step and a read step. The data copy overhead from PFS to HDFS is marked as HDFS put in Figure 7. After that, the conventional Hadoop application reads the target data from HDFS. The read (processing) time is labeled as Hadoop in Figure 8. By contrast, our PortHadoop directly accesses and processes the data in CephFS. Compared with "HDFS Put" and "Hadoop" processing approach, the input dataset of PortHadoop reside in the remote PFS, and input I/O occurs at separate nodes from Hadoop slave nodes. Recall that PortHadoop conducts a remote data access, reading data from a PFS system to an HDFS system. This remote data access process can be overlapped and pipelined with data processing. In this direct data processing cost, we did not perform any prefetching in this experiment since prefetching is application dependent, and here we want to show the advantage of avoiding copying. As shown in Figure 7, for the 8 GB case, the HDFS put + Hadoop time is 243 seconds + 355 seconds = 598 seconds, and the PortHadoop is 389 seconds. PortHadoop improves the performance by 34.9%. Figure 7 demonstrates the correctness of PortHadoop design and its potential, but not its best possible performance.

We also conduct our experiments with PVFS2. Eight I/O server nodes are set up to test the scalability for PortHadoop. PortHadoop is configured with 4, 8, and 16 slaves, respectively. As Figure 8 shows, PortHadoop significantly reduces the overall execution time while the number of concurrent map tasks varies from 2 to 8 per slave, and the target data size from 1GB to 8GB. While the number of the concurrent maps task is small, e.g. 2, the performance improvement by PortHadoop is limited, especially when the target data size is also small. This is attributed to the fact that PortHadoop uses memory as a stage and adopts blocking data access. Map task starts processing only when the data transfer completes. Hence, the data transfer time is exposed to the overall elapsed time. In contrast, after data copy phase, the consequent job of Hadoop can achieve data locality and task processing can overlap the disk I/O. Nevertheless, PortHadoop always outperforms the copy approach by standard Hadoop.

To further investigate the scalability of PortHadoop and evaluate the potential bottleneck, we employ up to 16 slaves with 8 concurrent map task processing per node, to saturate the network bandwidth between PFS and PortHadoop system. Figure 9 shows that PortHadoop improves the overall performance but suffers from the limited network bandwidth for further scalability. However, the copy phase in naive approach is still costly due to the constraint of the network bandwidth.

Figure 10 shows the performance of PortHadoop with different applications with 4GB input data. Copying these data from CephFS to HDFS costs roughly 134 seconds. The performance of both TeraSort and TestDFSIO-read outperform that of the original Hadoop because CephFS provides extra and better I/O bandwidths for data read. Particularly, TestDFSIO-read is a pure read workload, it costs 60 seconds in PortHadoop while 190 seconds in Hadoop without data copy cost, since CephFS can provide better read bandwidth than HDFS.

D. Evaluation on a Bio-application

We use a bio application named OCTree to further demonstrate the portability and functionalities of PortHadoop. Octree is used for classifying protein-ligand binding geometries [19]. The proposed method is implemented as a two-phase Hadoop MapReduce job. The geometry reduction and key generation

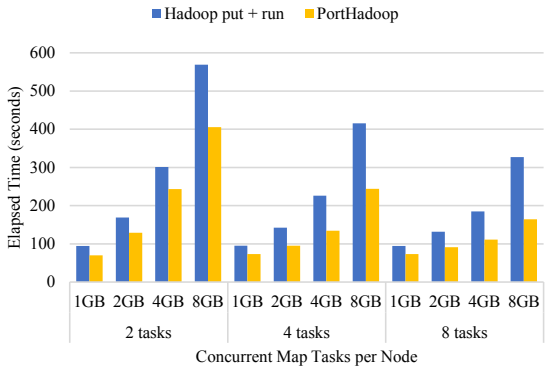


Fig. 8. Performance as a function of number of concurrent map task and input data size (PortHadoop integrates with PVFS2)

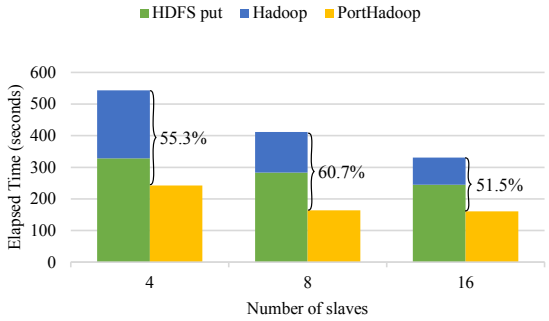


Fig. 9. Performance comparison between Hadoop and PortHadoop for 3 number of slaves. Each WordCount job processes 8 GB wiki text data with 8 concurrent map tasks per slave.

constitute the first-phase MapReduce job, where map tasks read large datasets as the input for further iterative computation. The second phase takes the output of the first phase as input, and this constitutes one round of iteration. This iterative octree-based clustering algorithm is implemented as a chain of MapReduce jobs and the I/O amount in the iterative phase is usually quite small compared to the first phase. In this paper, we only focus on the first phase computation and report its performance in PortHadoop. This application has strict semantic for its input data. The target file consists of several input units. Each input unit is about 57MB. We conduct this experiment on eight Hadoop slaves, each slave with two concurrent map tasks. As Figure 11 shows that PortHadoop

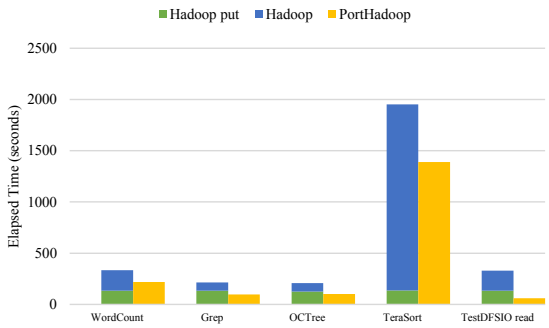


Fig. 10. Performance of PortHadoop with different applications under CephFS.

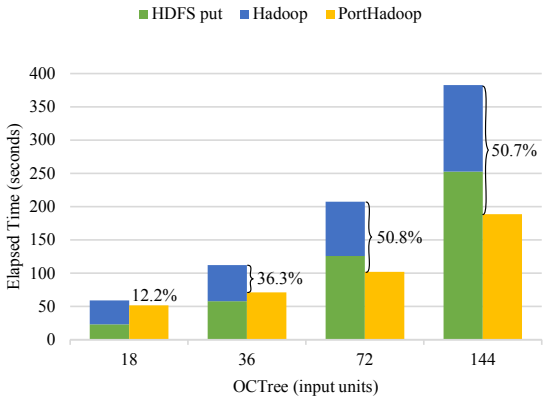


Fig. 11. Performance comparison with bio application Octree under CephFS.

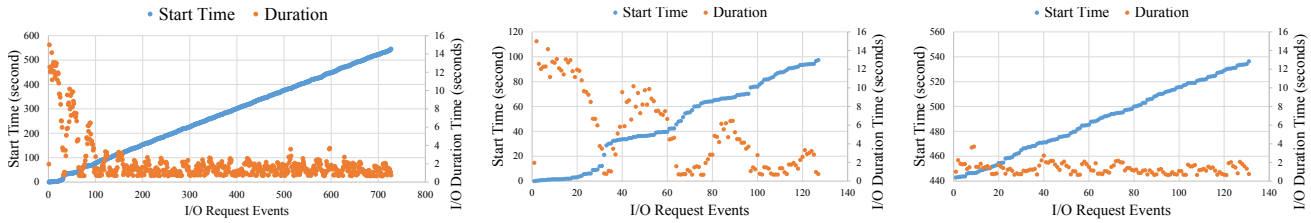
has about 10% more time compared with Hadoop framework, similar to the previous experiments. However, it avoids the data migration phase. Therefore, it always outperforms the native copy approach and improves overall performance roughly by 50% when the job is large.

E. Profiled I/O Analysis

To illustrate the potential of overlapping the computation and I/O in PortHadoop, we profiled the I/O events of map phase in Figure 12(a). To clearly show the I/O behavior and data transmission cost, we conduct WordCount on a weather simulation data set of 48GB, which needs multiple waves of map tasks to complete. The experiment run on 8 PVFS2 server nodes and 8 slaves in PortHadoop, each slave is configured with four concurrent map tasks. Figure 12(b) shows the first 128 I/O events. At the very beginning, the I/O requests from a wave, and such bursty I/O requests results in I/O contention. It is because map tasks issued I/O in a wave, the start time of current event depends on the completion time of previous map task. I/O contention is alleviated later when the I/O requests vary. As shown in Figure 12(c), I/O duration and map task start time became stable. And comparing two Figure 12(b) and Figure 12(c), the 128 I/O events in both cases cost about 100 seconds. The computation and I/O are overlapped. The performance of data transfer will be enhanced as I/O bandwidth increased. Both PortHadoop and Hadoop will suffer from the network overhead caused by reading from a remote site. However, the consequent performance of PortHadoop is alleviated by processing data in pipeline manner that overlaps data transfer and processing.

F. Evaluation on Performance of Data Prefetching

We evaluate data prefetching feature of PortHadoop under PVFS2 environment with 8 server nodes by conducting Wordcount on 8 Hadoop slaves. Each slave is configured with two concurrent map tasks. The maximum number for prefetching data blocks in PortHadoop is 2. Figure 13 shows the performance improvement of PortHadoop from data prefetch feature by 5% on average. This is reasonable because as Figure 12(c) shows, the average I/O duration is around 2 seconds when the contention reduced, considering the execution time of the map tasks is around 20 seconds that is multiple compared with the I/O time. Therefore, the overlapped I/O time is a related small portion to overall elapsed time.



(a) I/O Profiled overview. (b) I/O Profiled for the first 128 I/O requests. (c) I/O Profiled for the last 128 I/O requests.

Fig. 12. Profiled I/O events of map phase of WordCount on 48GB scientific data in PortHadoop. The start time labels the I/O request time, and the I/O duration means the elapsed time between the start time and the completion time for I/O request event.

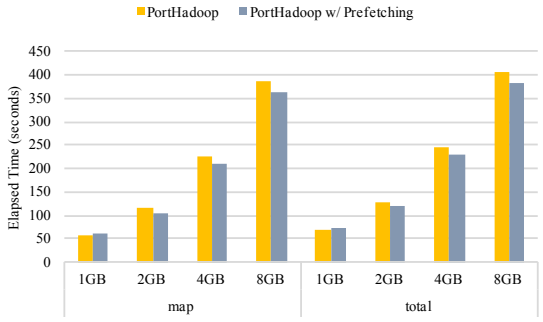


Fig. 13. Performance of PortHadoop with and without data prefetching under PVFS2.

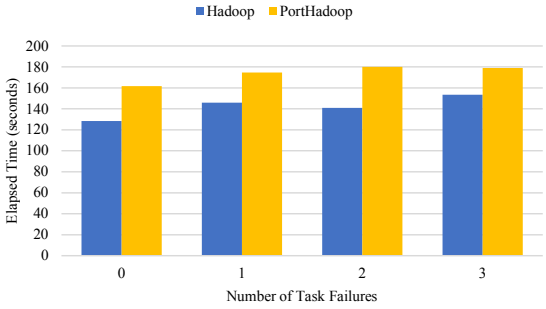


Fig. 14. Performance under task failure: Hadoop processes data reside at HDFS whereas PortHadoop processes data reside at CephFS.

G. Performance under Failures

We inject failures to map tasks in order to evaluate the fault tolerance feature and overhead of PortHadoop. We run WordCount on 8 slaves, each with 8 concurrent map tasks. The input dataset is 8 GB wiki text data. In Figure 14, we randomly inject failures to map tasks that trigger failure-redo attempts. For a fair comparison, failure injects time is at about one- third of map phase, and testing was run five times. The performance under failure is slightly higher than the failure-free runs for both native Hadoop and PortHadoop. The failure penalty of PortHadoop is slightly higher than one of native Hadoop. We think that this is because its input data reading from remote PFS is a blocking procedure through the network.

IV. Related Work

There are several research works for integrating MapReduce and HPC data processing power. MRAP [20] proposed to bridge the semantic gap between HPC data models (e.g., NetCDF) and MapReduce data formats. It significantly reduces data migration and alleviates data preprocessing workloads for MapReduce applications. In contrast, our proposal is a copy-free, data transferring and processing pipeline approach. SciHadoop [5] attempted to fill the gap between byte stream data model (e.g., Hadoop) and the highly structured, array-based binary data model (e.g., NetCDF), supporting Hadoop MapReduce framework to store and process NetCDF data. SciHadoop manages the physical-to-logical mapping for MapReduce applications and introduces several optimization techniques such as reduction of data transfer and remote reads, avoidance of unnecessary scan operations. Our work focuses on making Hadoop efficiently access and process remote data, and it will be extended to support high-level data processing that is built on top of Hadoop.

Researchers in HPC proposed to extend existing PFS to support its non-native MapReduce workloads. PVFS-shim-layer [21] enables PVFS to have a competitive performance with default HDFS. GPFS-FPO [22] is proposed to replace HDFS and extend GPFS for Hadoop MapReduce. However, these approaches require modifications on raw implementations, conventional system configurations, and data placements, which are not transparent to underlying native parallel file systems. In particular, different from raw GPFS, GPFS-FPO is designed for distributed environment where each node is equipped with local disks and store big size chunk. Lustre community dedicatedly to support MapReduce applications [23]. Lustre is naturally POSIX-compliant and designed with high-speed networks but does not support replication, and the fault tolerance features to MapReduce applications are unknown. Ceph also proposed to support both HPC and MapReduce applications [24]. Moreover, these efforts are tightly coupled implementations with a specific PFS and can hardly be used by other in-production file systems to support their native workloads. Our method is a portable solution and does not need to modify the existing PFS; MapReduce application will run on its native cluster.

The MapReduce community tried to improve the response time of applications, adopting pipelining or in-memory processing strategies, such as MapReduce Online [25], Themis [26], M3R [27]. These pipelining strategies are adopted within MapReduce cluster. Themis [26] and M3R [27] trade reliability for performance by avoiding materializing the intermediate data into disks. PortHadoop processes data from external PFS in a pipelining manner. It may achieve a finer-grain pipeline processing by adopting data prefetching. Our proposal will inherit the fault tolerance mechanism as default Hadoop does, such as fail-redo mechanism and intermediate

result materialization mechanism, without significantly scarifying the reliability for achieving performance.

V. Conclusion

We have presented PortHadoop, a portable Hadoop architecture supporting direct data fetching from parallel file systems, in this study. Data access has become the premier performance bottleneck of HPC. This is especially true for advanced HPC applications where data analysis is performed, in addition to scientific simulations. However, due to historical and technical reasons, data analytic software traditionally are developed and available under Hadoop systems, whereas HPC applications are developed under MPI environments. The cross-platform data access between Hadoop and MPI is extremely costly. PortHadoop is designed to solve the cross-platform data access issue. With PortHadoop, researchers can promptly analyze identified events without copying the entire data set from PFS to Hadoop, and consequently accelerate scientific discovery and significantly reduce costs in computation and storage. Our experimental results show that PortHadoop is effective and compatible with existing PFS such as PVFS2 and CephFS. In particular, under PortHadoop: 1) MapReduce can read data directly from PFS without data copying. The target blocks processed by map tasks reside in memory rather than on disk. 2) Only the needed data at PFS are taken to Hadoop at runtime. 3) Blocks in a PFS files can be accessed concurrently. 4) According to the amount of data required by map tasks, the data transfer operations from PFS to HDFS can overlap with MapReduce data processing. 5) PortHadoop supports fault tolerance as default Hadoop does. In the future, we plan to improve PortHadoop in three areas: (1) Apply PortHadoop to Apache YARN and support more parallel file systems, including Lustre and GPFS. (2) To support various data formats, including SequenceFile and compressed datasets. (3) Deploy, test, and optimize PortHadoop on a large-scale platform, such as Amazon EC2.

Acknowledgement

The authors would like to thank the partial support from NASA AIST 2015 funding. This research is also supported in part by NSF under NSF grants CNS-0751200, CCF-0937877, and CNS-1162540.

References

[1] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The Hadoop Distributed File System," in *Proceedings of IEEE Symposium on Mass Storage Systems and Technologies*, 2010.

[2] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth *et al.*, "Apache Hadoop Yarn: Yet Another Resource Negotiator," in *Proceedings of the 4th annual Symposium on Cloud Computing*, 2013.

[3] D. Nurmi, R. Wolski, C. Grzegorczyk, G. Obertelli, S. Soman, L. Youseff, and D. Zagorodnov, "The eucalyptus open-source cloud-computing system," in *Cluster Computing and the Grid, 2009. CCGRID'09. 9th IEEE/ACM International Symposium on*. IEEE, 2009, pp. 124–131.

[4] C. Evangelinos and C. Hill, "Cloud computing for parallel scientific hpc applications: Feasibility of running coupled atmosphere-ocean climate models on amazons ec2," *ratio*, vol. 2, no. 2.40, pp. 2–34, 2008.

[5] J. B. Buck, N. Watkins, J. LeFevre, K. Ioannidou, C. Maltzahn, N. Polyzotis, and S. Brandt, "Scihadoop: Array-based query processing in hadoop," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, 2011, p. 66.

[6] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. H. Katz, S. Shenker, and I. Stoica, "Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center," in *Proceedings of USENIX Symposium on Networked Systems Design and Implementation*, 2011.

[7] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes, "Large-scale cluster management at google with borg," in *Proceedings of the Tenth European Conference on Computer Systems*. ACM, 2015, p. 18.

[8] P. Wang, H. Jiang, X. Liu, and J. Han, "Towards Hybrid Programming in Big Data," in *7th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 15)*. Santa Clara, CA: USENIX Association, Jul. 2015.

[9] 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 *In Proceedings of the 2012 IEEE Conference on Massive Data Storage*, 2012.

[10] N. Liu, J. Fu, C. Carothers, O. Sahni, K. Jansen, and M. Shephard, "Massively parallel I/O for partitioned solver systems," *Parallel Processing Letters*, vol. 20, no. 4, pp. 377–395, 2010.

[11] Philip H. Carns and Walter B. Ligon III and Ross, Robert B and Thakur, Rajeev, "PVFS: A Parallel File System for Linux Clusters," in *Proceedings of the 4th Annual Linux Showcase and Conference*, 2000.

[12] F. B. Schmuck and R. L. Haskin, "GPFS: A Shared-Disk File System for Large Computing Clusters," in *Proceedings of USENIX Conference on File and Storage Technologies*, 2002.

[13] S. Weil, S. Brandt, E. Miller, D. Long, and C. Maltzahn, "Ceph: A Scalable, High-Performance Distributed File System," in *Proceedings of USENIX Symposium on Operating Systems Design and Implementation*, 2006.

[14] P. J. Braam *et al.* (2014, Mar.) The Lustre Storage Architecture. ftp://ftp.uni-duisburg.de/linux/filesys/Lustre/lustre.pdf.

[15] NASA. NASA Center for Climate Simulation. http://www.nasa.gov/topics/earth/features/climate-sim-center.html.

[16] S. J. Plimpton and K. D. Devine, "MapReduce in MPI for large-scale graph algorithms," vol. 37, no. 9, pp. 610–632.

[17] R. Thakur, W. Gropp, and E. Lusk, "On Implementing MPI-IO Portably and with High Performance," in *Proceedings of the sixth workshop on I/O in parallel and distributed systems*. ACM, 1999, pp. 23–32.

[18] ——, "Data Sieving and Collective I/O in ROMIO," in *Proceedings of the Symposium on the Frontiers of Massively Parallel Computation*, 1999.

[19] B. Zhang, D. T. Yehdego, K. L. Johnson, M.-Y. Leung, and M. Taufer, "Enhancement of accuracy and efficiency for RNA secondary structure prediction by sequence segmentation and MapReduce," *BMC Structural Biology*, vol. 13, no. Suppl 1, p. S3, Nov. 2013.

[20] S. Sehrish, G. Mackey, J. Wang, and J. Bent, "MRAP: A Novel MapReduce-based Framework to Support HPC Analytics Applications with Access Patterns," in *Proceedings of the ACM International Symposium on High Performance Distributed Computing*, 2010.

[21] W. Tantisiriroj, S. W. Son, S. Patil, S. J. Lang, G. Gibson, and R. B. Ross, "On the Duality of Data-intensive File System Design: Reconciling HDFS and PVFS," in *Proceedings of the International Conference for High Performance Computing, Networks, Storage and Analysis (Supercomputing)*, 2011.

[22] IBM. Deploying a big data solution using IBM GPFS-FPO. http://public.dhe.ibm.com/common

[23] Sun Microsystems Inc. (2014, Mar.) Using Lustre with Apache Hadoop. http://wiki.lustre.org/images/1/1b/Hadoop_wp_v0.4.2.pdf.

[24] C. Maltzahn, E. Molina-Estolano, A. Khurana, A. J. Nelson, S. A. Brandt, and S. Weil, "Ceph as a scalable alternative to the hadoop distributed file system," *login: The USENIX Magazine*, vol. 35, pp. 38–49, 2010.

[25] T. Condie, N. Conway, P. Alvaro, J. M. Hellerstein, K. Elmeleegy, and R. Sears, "Mapreduce online." in *NSDI*, vol. 10, no. 4, 2010, p. 20.

[26] A. Rasmussen, M. Conley, G. Porter, R. Kapoor, A. Vahdat *et al.*, "Themis: An I/O-Efficient MapReduce," in *Proceedings of the Third ACM Symposium on Cloud Computing*. ACM, 2012, p. 13.

[27] A. Shinnar, D. Cunningham, V. Saraswat, and B. Herta, "M3R: increased performance for in-memory Hadoop jobs," *Proceedings of the VLDB Endowment*, vol. 5, no. 12, pp. 1736–1747, 2012.