SCALER: Scalable Parallel File Write in HDFS

Xi Yang, Yanlong Yin
Department of Computer Science
Illinois Institute of Technology
{xyang34, yyin2}@iit.edu

Hui Jin
Parallel Execution Group
Oracle Corporation
hui.x.jin@oracle.com

Xian-He Sun
Department of Computer Science
Illinois Institute of Technology
sun@iit.edu

***Abstract*—Two camps of file systems exist: parallel file systems designed for conventional high performance computing (HPC) and distributed file systems designed for newly emerged data-intensive applications. Addressing the big data challenge requires an approach that utilizes both high performance computing and data-intensive computing power. Thus, HPC applications may need to interact with distributed file systems, such as HDFS. The N-1 (N-to-1) parallel file write is a critical technical challenge, because it is very common for HPC applications but HDFS does not allow it. This study introduces a system solution, named SCALER, which allows MPI based applications to directly access HDFS without extra data movement. SCALER supports N-1 file write at both the inter-block level and intra-block level. Experimental results confirm that SCALER achieves the design goal efficiently.**

***Keywords*—Parallel I/O, Distributed file systems, Optimization, HDFS**

I. Introduction

A. Two Camps of Large-scale Parallel Processing Systems

Supercomputers (e.g. the IBM Blue Gene system) [1] and MapReduce [2] based datacenters (e.g. Hadoop powered cloud computing systems) represent two different camps in the field of large-scale parallel processing. Many terms exist to illustrate their differences, such as parallel computing system versus distributed computing system, scale-up system versus scale-out system, or HPC system versus datacenter or Cloud computing system. In this paper, we use the terms HPC system and datacenter, respectively, to represent these two types of parallel processing architectures. These systems have evolved from separate application domains, and have very different strengths.

HPC systems are mainly used for large-scale computation-intensive applications, such as scientific simulations. These applications create many threads or processes to simulate massive components or solve massive subtasks. As these threads and processes frequently interact with each other, an HPC system consists of many specially manufactured processing cores that are tightly coupled with high-speed multi-dimensional networks. The corresponding programming models are based on message passing interface (MPI) [3] or multithreading implementations (e.g. OpenMP) [4] for efficient data sharing and interaction among processes/threads. The underlying storage system supporting HPC application is usually segregated from the computing cores. HPC storage systems utilize parallel file systems (PFS), e.g. PVFS2 [5], GPFS [6], and Lustre [7], to manage massive disks that are remotely accessed through the network. HPC systems are well-suited to computation-intensive workloads but often have difficulties coping with data-intensive applications.

On the other hand, datacenters are designed for large-scale data processing applications, such as the massive Internet services and statistical data analysis. For these data oriented applications, the subtask partitioning is based on data partitioning. The subtasks are loosely coupled and relatively independent of each other; thus, inter-subtask message passing and synchronization are often not required. This programming model is referred to as embarrassingly parallel programming and is well supported by MapReduce frameworks like Hadoop. To this end, a datacenter usually consists of many commodity computer nodes that are connected with regular local area networks. Each node has local disks that are utilized by the cluster storage system; thus, the computing system is co-located with the storage in the same cluster. These clusters are managed by distributed file systems (DFS), e.g. GoogleFS [8] and HDFS [9]. One major feature that DFS has but PFS lacks is data locality — with data location information, the scheduler is able to distribute tasks onto compute nodes where the required data reside.

B. The N-1 Parallel File Write from MPI Applications to HDFS

Many HPC applications are data-intensive, such as data mining and checkpointing, or have data-intensive phases, such as the database preparation phases before DNA sequence searches [10] and the data dumping phases at the end of scientific simulations. As mentioned above, HPC systems lack a datacenter's efficiency in handling data-intensive workloads [11]. Big data applications require the linkage of high performance computing and data processing power. Therefore, there is a desire to run data-intensive HPC applications on datacenters [12]. Thus, the paradigm "HPC in the Cloud/datacenter" is getting more and more attention under this context. YARN [13] and Mesos [14] represent this trend of supporting various workloads including both MPI and MapReduce on a shared cluster. However, the HPC-datacenter merge has many difficulties, due to some semantic gaps. One critical obstacle is that their underlying file systems support different levels of parallelism for writing files. In DFS and PFS, a common strategy is that large data files are partitioned into fixed-size blocks and those blocks are stored onto multiple storage nodes (e.g., Datanodes in HDFS

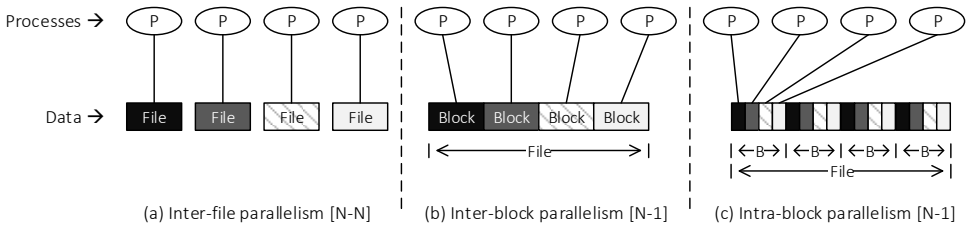

Fig. 1. Three levels of data access parallelism.

and storage nodes in PVFS2). We categorize the data access parallelism in such parallel I/O systems into three categories: inter-file, inter-block, and intra-block parallelism, illustrated in Fig. 1. Inter-file parallelism represents the N-N parallel file access, where N processes access N files and each process has one independent file for itself. Both inter-block and intra-block parallelism are of N-1 parallel file access where N processes share one file and read/write different parts of that file concurrently.

In terms of data write, the problem is that all the three types of parallelism are common for HPC applications but HDFS only supports the first type — inter-file parallel write.

Due to this limit, the following performance characteristics are observed. For N-N file writes, each process writes its own file and there is no file-level lock contention. As the number of processes or the number of Datanodes increases, the aggregated bandwidth grows almost linearly. However for N-1 file writes, N processes share one file and compete for the file-level lock. Thus for a given file in HDFS, only one process can get the lock and successfully open the file in WRITE mode; any other requests are rejected. In other words, parallel file write is not supported. The HDFS N-1 write is sequential and its performance does not increase with the system scale.

HPC relies on data sharing and communication extensively. N-1 file writes are a must for HPC applications. The poor performance of HDFS in N-1 file writes motivates this study. We propose SCALER — a solution to enable and optimize MPI based N-1 parallel file writes on HDFS (the word SCALER is short for "SCalable pArallel fiLE wRite.")

C. SCALER: Enabling and Optimizing Parallel File Write in HDFS

As mentioned in the previous subsection, HDFS only supports inter-file parallel writes. SCALER improves upon this by adding HDFS inter-block and intra-block parallel file write support. **To enable inter-block parallel write**, where client processes share files but not blocks, multiple processes must be allowed to obtain the same write lock for a shared file. We redesigned the lock control mechanism in HDFS to achieve this. Also to ensure the correct logical order of blocks in a file, we also designed a metadata management mechanism that allows the scenarios where multiple processes commit multiple block metadata entries concurrently. **To enable intra-block parallel write**, where multiple processes concurrently write a given block, we also need to handle the block-level write contention, besides the above lock and metadata management. We designed the block aggregation mechanism. It is similar to the Collective I/O scheme in MPI-IO [15], but specially adapted to HDFS architecture; the algorithm is aware of block locality during aggregation.

The SCALER I/O scheme requires two major parts of implementation — the client side in MPI-IO and the server side in HDFS. The client implementation in MPI-IO is an ADIO driver [15] that performs a selection algorithm that assigns aggregators to target blocks and implements client-side block aggregation. On the HDFS server, the above mentioned lock control and metadata management are implemented.

SCALER provides two types of parallel file write functions: synchronous and asynchronous. The basic synchronous function efficiently utilizes the parallel architecture and improves the overall I/O bandwidth. To reduce the I/O response time, an asynchronous parallel file write function is designed and implemented which handles all data marshaling and parallel writes in the background. In this paper, by response time, we mean the time between client side's I/O being issued and the I/O function call being returned.

This implementation is evaluated with various experiments to confirm our contributions in:

1) **Scalable parallel file write performance.** The overall HDFS bandwidth of parallel N-1 file writes should increase linearly with the number of Datanodes.
2) **Reducing I/O response time.** By moving inter-Datanode data exchange to the background, our optimized asynchronous parallel write function should respond fast and works efficiently in HDFS.
3) **Effective buffer for burst write workload.** In the asynchronous write mode, SCALER buffers data locally in the requests' hosting Datanodes. This mechanism creates an effective burst write buffer for HDFS.

The rest of this paper is organized as follows. In Section II, we first discuss the key design considerations, design details, and implementation of SCALER, including lock sharing mechanism, synchronous and asynchronous data write, and metadata management. Section III presents the evaluation results to confirm the effectiveness of our design. Section IV presents the related work. Finally, Section V concludes this paper.

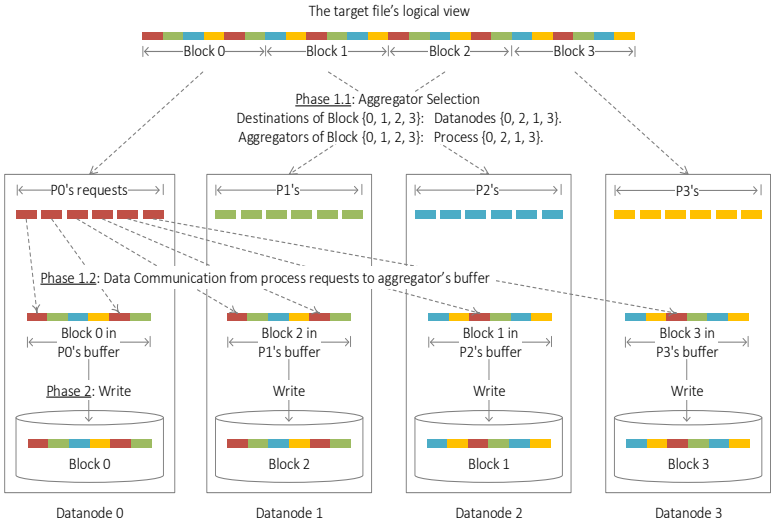

Fig. 2. Synchronous parallel file write.

II. Design and Implementation

Subsection II-A introduces the essential design considerations; then Subsection II-B introduces the redesign of the file lock control that enables lock sharing among multiple client processes. The synchronous and asynchronous parallel file write are introduced in Subsection II-C. Subsection II-D describes the metadata management and Subsection II-E discusses some important notes on the implementation.

A. Design Considerations

Before presenting the details of the design and implementations, we discuss some important considerations that led to this design. MapReduce is known as an effective batch data processing system. It transparently supports embarrassingly parallel processing, where there is no dependency among peer subtasks. The parallel file write functionalities provided by popular distributed file systems are also "embarrassingly" simplistic. In HDFS, parallel write to a shared "HDFS file" is simply not allowed. For parallel write, multiple tasks of the same MapReduce job must write into multiple subfiles of a directory. A task needs to obtain the write lock of a file in order to write onto it and only one task can hold the lock at any given time. While a file is being written, other attempts to write or to update the file will be rejected immediately. As a result, to support parallel write on a shared file, including inter-block and intra-block writes, the default HDFS must be modified. As mentioned briefly in Section I, inter-block parallelism requires the modified lock control and metadata management. Besides these modifications, the intra-block writes also require extra block aggregation techniques.

In HDFS deployments, HDDs are the dominant storage devices. HDDs perform well for contiguous data accesses but poorly for non-contiguous ones. To avoid non-contiguous data writes, HDFS utilizes streaming write for all data writes and disables the "offset seek" function for files opened in WRITE mode. With a streaming write method, the bytes are written in sequence. Because multiple streaming writes on the same file are not allowed in Java or HDFS, SCALER must transparently aggregate a data block into to an assigned aggregator, which will then save the block onto HDFS.

Furthermore, the SCALER design adopts a client-side collective I/O approach that is similar to the Collective I/O scheme in MPI-IO. Since the underlying I/O modules do not allow parallel write to HDFS blocks, the aggregation methods need to be aware of an HDFS file's block partition scheme. In other words, the data partitioning among aggregators should be aligned with the target file's block partition on HDFS.

HDFS provides a block location retrieval service. This is because MapReduce's task scheduling is data locality aware and always tries to schedule a task on the Datanode where the target data reside. However, in HPC systems, parallel file systems provide a simple file interface to the applications and hide all the details of underlying nodes and disks management. As a result, the default Collective I/O implementation does not pay attention to the locality of the aggregator processes because the storage server nodes are considered remote. Our previous work [16] has recognized this problem. To further efficiently support parallel write in HDFS, SCALER's I/O aggregation algorithm takes the block location into consideration and selects the aggregation destination properly to achieve system load balance and minimize the amount of cross-node data migration.

B. The Lock Sharing Mechanism Redesign

To allow multiple client processes to open and write a shared file, the system needs to allow multiple processes to acquire the write lock to a single file at the same time. Thus, the first step of our work is to add a new write lock management mechanism for MPI based applications.

When multiple MPI processes open a shared HDFS file in WRITE mode, HDFS receives multiple requests to acquire the write lock of that file. For the first request that arrives,

the Namenode initializes the write lock token, performs the open operations, and returns the lock to the process. After that, requests from consequent processes receive a response which contains the same write lock that it sent to the first process. For each file that is being opened for write, Namenode also keeps a counter. Every time a new process requests to open the file in WRITE mode, the counter will increase by one. On the other hand, whenever a process closes a file, Namenode decreases the corresponding counter by one and checks the counter's value. If it is zero, which means, no process is still using the file, Namenode closes it; otherwise, Namenode keeps the file open.

C. Synchronous and Asynchronous Write Functions

Typical I/O systems like POSIX, MPI-IO, etc., support two types of I/O operations: synchronous and asynchronous I/O operations, which are also called blocking and non-blocking I/O. Under synchronous I/O, the requesting process will be blocked until the completion of the I/O function call that means the completion of the actual I/O operation. Asynchronous I/O allows the requesting process to continue after submitting the request, and after that the actual I/O operation can be conducted at a later time and possibly overlap with the computation.

These two different I/O functions can satisfy different user needs. When the synchronous write function returns, the data are in the right place in HDFS and can be safely read out by other processes. Thus, synchronous I/O is well-suited to scenarios that require real-time data consistency where the consequence reads will arrive immediately or very soon after the writes on the same data finish. When there is no requirement of real-time data read back, programmers could choose to use asynchronous I/O to get optimal response time for their data writes. The actual data writes and reorganization can be done later. To fully satisfy HPC application usage, we implemented both of these two types of I/O operations to bridge MPI-IO and HDFS.

1) Synchronous Data Write: For synchronous I/O, a two-phase I/O technique is adopted and illustrated in Fig. 2. The basic idea is to reorganize data among processes so that each data block only has one process writing it and the block-level contention is eliminated. For each conflict block to be written, the system selects a process as its aggregator (Phase 1.1) before any data movement. Then, all the small data pieces written to that block will be sent to an aggregator (shown as Phase 1.2). After the aggregator finishes gathering all the data for that block, it enters Phase 2 where it will save the block onto HDFS. Each of writers in Phase 2 conducts independent block writing. Therefore, the key design element is Phase 1 that decides the target aggregator for each written data pieces and requires a carefully designed algorithm.

Aggregator Selection Algorithm. The aggregator selection algorithm balances block distribution among nodes and minimizes the amount of data movement. Due to the nature of parallel task execution, the slowest task usually determines the completion time of the whole job. Algorithm 1 prioritizes workload balance as its first priority.

Algorithm 1 Block Aggregation Selection

Terminology

$A(i,j)$ is size of inter-node data exchanged if node i is selected as the aggregator for block j.
Threshold q is the maximum number of allocated blocks on each node.
B is number of blocks.
N is number of nodes.
$M(i)$ is number of allocated blocks on node i.
$S(j)$ is the assigned node id for block j.

Algorithm

Calculate $A(i,j)$ based on access pattern, partition matrix A by rows: $\sum V(i) = \sum\sum A(i,j)$
$q \leftarrow \lceil B/N \rceil$
$M(i) \leftarrow 0, 1 \leq i \leq N$
$S(j) \leftarrow -1, 1 \leq j \leq B$
for $i = 1 \rightarrow B$ **do**
  $(n,b) \leftarrow \{(x,y)|minA(x,y)\}$
  $S(b) \leftarrow n$
  $M(n) \leftarrow M(n) + 1$
  if $M(n) = q$ **then**
    $V(n) \leftarrow +\infty$
  end if
end for

This algorithm first initializes a matrix A where $A(i,j)$ records the amount of transferred data if node i is selected as host for block j. It also calculates the maximum number of blocks that each node can accept, which is the total number of blocks divided by the total number of Datanodes. The algorithm also maintains a list of the numbers of allocated blocks for each node. Then, the algorithm starts to select the destination Datanode for each block. It always selects the cell with the minimum value from the entire matrix $A(x,y)$. Then, node x will be selected as the host of block y. When a node is selected, its allocated number increases by one. Once the node reaches a threshold of allocated blocks, it sets its associated row of matrix A to $+\infty$, which indicates that such node will not participate in further aggregator selection. The algorithm repeats this procedure for all target blocks and the result is saved into a list indicating the destination Datanode of each block.

Time complexity of initialization is $B \times N + B + N$, where B is total number of blocks and N is total number of Datanodes. Finding the minimum cell of matrix A takes at most $B \times N$ of time. Assigning $+\infty$ to a vector takes constant B units of time. This procedure inside the for loop runs in time $O(NB)$. Hence, the time complexity of this algorithm is $O(NB^2)$.

2) Asynchronous Data Write: To make the best use of the Hadoop architecture, an asynchronous file write mechanism is implemented. Synchronous write has to perform the online block aggregation before saving blocks onto HDFS. Comparing with that, asynchronous write can further improve the

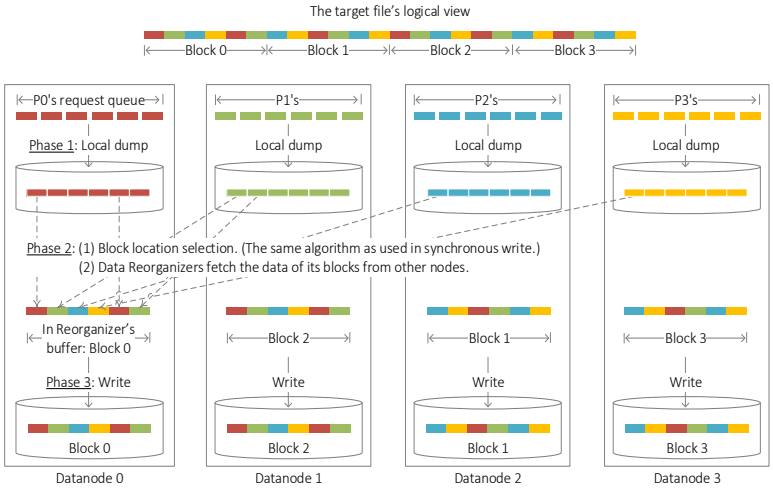

Fig. 3. Asynchronous parallel file write.

I/O performance by making the block aggregation operations offline. Therefore, it can fully exploit the benefits of data locality; as confirmed by the results in Subsection III-D.

As illustrated in Fig 3, we adopt a three-phase design to support asynchronous data write. In Phase 1, all the write requests dump their data onto their hosting Datanodes. These data are saved contiguously and the writing logs are kept in metadata for future data reorganization. The log metadata keeps the mapping between the local data pieces and their location in the destination HDFS files. Once the local dumping is done, the asynchronous function call returns. To maintain the same level of fault tolerance as the default HDFS, the local dumped data will be erased only after the reorganization has been successfully performed.

As mentioned above, for asynchronous operations, the data are just dumped locally with metadata recorded (as shown in Fig. 3's Phase 1), after the write requests return.

Once the data reorganization is started, on each Datanode, a data reorganizer process runs as a daemon, which is responsible for the block aggregation (Phase 2 in Fig. 3). Similar to the case of synchronous write and using the same block aggregation selection algorithm, the customized HDFS data placement policy selects a destination Datanode for each target block. The algorithm attempts to balance the I/O workload and minimizes the inter-Datanode data movement. Once the block locations are decided, each data reorganizer will fetch all the blocks of its charge (Phase 3 in Fig. 3).

D. Metadata Management

A large HDFS file is partitioned into multiple HDFS blocks with a fixed size (default 64MB), and the Namenode manages the blocks' locations and the order of blocks in that file. The order of blocks stands for the logical sequence indices of blocks. For example, the first block includes the bytes that offset from 0 to $(64 \times 220 - 1)$ and the second block's data bytes offset from (64×220) to $(2 \times 64 \times 220 - 1)$. Since the default HDFS does not allow parallel data writes or updates, the block sequence is the same as the blocks' write completion order. In other words, after Namenode receives a metadata-committing message saying that nth block is completed, the next write request naturally means that the requesting process wants to write data to the $(n+1)$th block.

With parallel file write enabled, multiple processes and threads can request to write different blocks in a shared file. Their requests may arrive at Namenode at different times in a random order. Hence, there can be a mismatch between the correct logical block order and the allocation request order. To solve this problem, we let each client process or thread pass the block number as a hint along with the block requests to Namenode. After all the blocks get to their final destinations, Namenode reorders the metadata entries according to these hints and commits the correct information to persistent storage.

E. SCALER Implementation

A prototype SCALER system is implemented under MPICH2 (version 1.4.1p1) and Hadoop (version 1.0.0).

1) Client-Side Implementation in MPICH2: The client-side implementation is in the form of an ADIO driver [17] added into the ROMIO module of MPICH2. Listed below are the tasks and related descriptions of SCALER's client-side implementation.

- **MPI programs in C communicating with HDFS Clients in Java.** To allow MPI programs written in C language to access HDFS-Client's Java I/O interfaces the libhdfs [18], a JNI (Java Native Interface) based C API for HDFS, is utilized. The Hadoop library libhdfs provides a simple subset of C APIs to manipulate Hadoop files.
- **Online block aggregation.** In MPICH2's original collective I/O and ADIO implementations, there are two key functions used to specify the aggregation policy: file domain calculator (`ADIOI_Calc_file_domains`) and aggregator calculator (`ADIOI_Calc_aggragator`). We customize these two functions in our ADIO driver for HDFS using SCALER's aggregation policy shown in

Algorithm 1.

- **Local buffering and the metadata.** In the asynchronous write mode, SCALER buffers data locally in the requests' hosting Datanodes, as shown in Fig. 3's Phase 1. This mechanism creates an effective burst write buffer for HDFS. To ensure the data integrity, metadata is needed to keep records on which files and what offsets the locally buffered data belong. We utilize a B+ tree data structure to keep the metadata information. Each request (file domain for Collective I/O) is represented by one leaf node via marking it with the start and end offsets. Each leaf node further points to a specific file name and its corresponding local offset that match the corresponding request. Such design guarantees the simplicity of metadata management and ensures high efficiency when locating a request.

2) Server-Side Implementation in HDFS: In HDFS Namenode, we add one set of RPC calls that conducts the lock control and metadata management for parallel write. While performing parallel writes, the SCALER's client-side MPI-IO driver will invoke these new calls. The original RPC calls in HDFS stay unchanged, so the SCALER implementation will not affect the execution of MapReduce applications.

III. Performance Evaluation

SCALER's prototype implementation is evaluated on a 65-node Sun Fire Linux based cluster, including one head node and 64 computing nodes. The head node served as Master and Namenode, and the remaining nodes served as Slaves and Datanodes. The Namenode is a Sun Fire X4240, equipped with dual 2.7GHz Opteron quad-core processors, 8GB memory, and 12 500GB 7200RPM SATA-II drives configured in RAID5 array. Each Datanode has two Opteron quad-core processors, 8GB memory and a 250GB 7200RPM SATA-II disk (HDD). All the 64 nodes are connected by gigabit Ethernet. The N-1 file writes performed in all the following evaluations are intra-block parallel writes and each HDFS block is shared by all client processes. We also did experiments on the Longhorn cluster [19] as described in Subsection III-G.

A. The Scalability of the Lock Control and Metadata Management

In our implementation, two simple mechanisms are added to the Namenode: the lock sharing mechanism, and the metadata reordering and committing mechanism for concurrent write. Our first experiment is to show the scalability of these add-ons. A synthetic parallel write benchmark is created which creates a task on each Datanode. All these tasks write to one shared HDFS file but different blocks. In other words, there is no sharing of HDFS blocks and collective data exchange among processes is not possible (as shown in Fig. 2 and Fig. 3). All processes write the blocks to their hosting Datanodes, which means there would be no remote data access and the workloads on all Datanodes are balanced. The number of Datanodes, number of tasks, and workload amount for each process for different numbers of metadata entries are varied. Fig. 4 shows

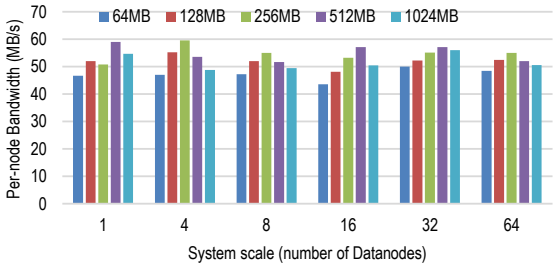

Fig. 4. Scalability of file lock and metadata management.

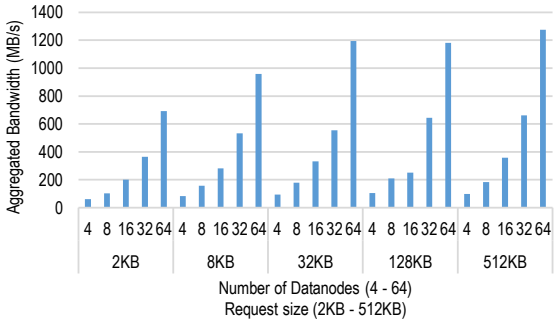

Fig. 5. Scalability of synchronous write with IOR.

the experimental results. The different legends represent the data write amounts per task. This confirms that the per-node average bandwidth does not show any degradation while the system scale grows. Therefore, the added lock and metadata management does not affect HDFS's scalability.

B. Scalability of Synchronous Write

To further validate the system scalability, the IOR parallel I/O benchmark [20] was used to evaluate synchronous write performance at various system scales. The request size was varied from 2KB to 512KB, the numbers of nodes was varied from 4 to 64, and each task writes 128MB of data. Fig. 5 shows that the synchronous write mechanism gained scalability. It can be observed by the bandwidth increases linearly as system scale increases. Compared with the default HDFS, our synchronous file write improves bandwidth noticeably. Secondly, the bandwidth is stable among different request sizes for a given number of Datanodes. This shows the ability of SCALER to handle small requests efficiently. This is due to the block-level Collective I/O approach that aggregates small requests to larger ones before writing them onto the file system.

C. Scalability of Asynchronous Write

The time required for an asynchronous write function is from the beginning of the call to the time that the data have been dumped onto the hosting Datanodes. Fig. 6 shows that asynchronous write mechanism also achieves excellent system scalability. Doubling the number of Datanodes simply doubles the aggregated bandwidth. Additionally within the same system scale, different request sizes result in approximately the

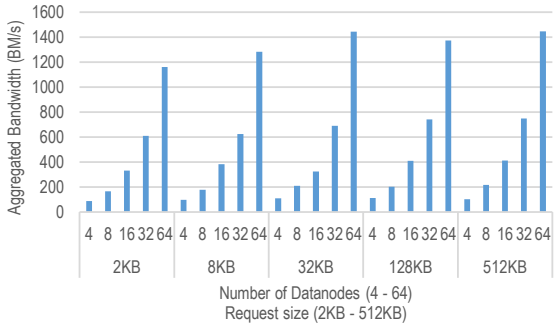

Fig. 6. Scalability of asynchronous write with IOR.

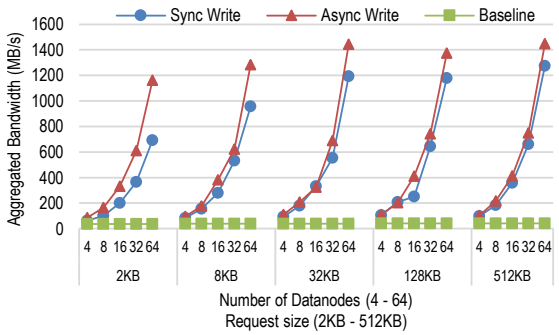

Fig. 7. Performance comparison with HDFS's default N-1 parallel file write.

same bandwidth. This shows that the system is able to handle both large and small requests efficiently.

D. Performance Comparison with the Baseline

In this test, SCALER's N-1 parallel write performance is compared with that of the default HDFS. Fig. 7 shows that HDFS's default N-1 write performance is fixed and equal to the write performance of a single Datanode independent of scale. SCALER performance, in contrast, scales linearly with the number of Datanodes.

E. Performance Comparison with the Upper Limit

For comparison, N-N file write performance is assumed to be the upper limit of HDFS's performance. Then, the performance of the two proposed parallel file write mechanisms are compared with this upper limit. The experimental setup is the same as the ones used in last subsection. In Fig. 8, the percentage values are calculated as SCALER's write performance divided by the upper-limit performance of HDFS. As expected, the N-1 asynchronous data write further improves the aggregated HDFS bandwidth, bringing the performance closer to that of the best scenario. The synchronous write performance ranges mostly between 70% and 85% of the upper-limit performance, although it can reach as high as 98% in some cases. For asynchronous parallel write, the performance is stable and around 95% of the HDFS's ideal N-N write performance. Through this comparison it was found that SCALER's N-1 file write is able to exploit HDFS's potentials on scalability and performance for N-1 parallel file write.

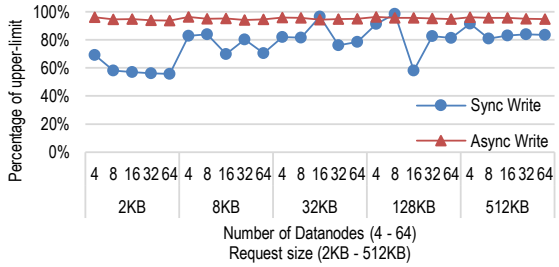

Fig. 8. Performance comparison with HDFS's upper limit.

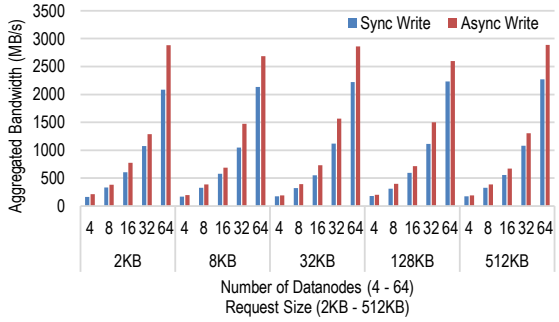

Fig. 9. Results on system scalability with HPIO.

F. Results with HPIO Benchmark

In this test, we evaluated the proposed SCALER write functions with a different parallel I/O benchmark HPIO [21] under the same system configuration. Fig. 9 shows that both synchronous and asynchronous approaches gained good scalability with HPIO: for almost all cases, doubling the system scale brings doubled overall system performance. Asynchronous writes outperformed synchronous ones in terms of aggregated bandwidths.

G. Performance Comparison with Different Number of Replicas

We also conducted our experiments on the Longhorn cluster [19] at Texas Advanced Computing Center (TACC). Each node we used is a Dell PowerEdge R610 machine that equipped with 48GB of RAM, 8 Intel Nehalem cores. A 15K RPM SAS drive is used as the local storage of HDFS.

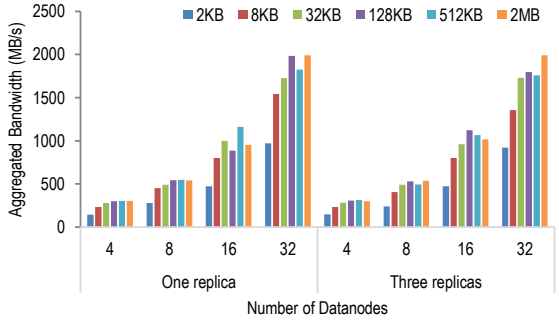

Fig. 10. Results of synchronized N-1 file writes with various request sizes on Longhorn cluster.

We run IOR benchmark to evaluate the performance. Each node writes 128MB with various request sizes, from 2KB to 2MB. In Fig. 10, SCALER shows the consistent performance scalability on Longhorn platform. The bandwidth with one replica is slightly higher than that with three replicas. Because SCALER only handles the data writes on the first replica, the other replicas are handled by HDFS's default pipelining writes, thus SCALER does not affect the default HDFS's system scalability.

IV. Related Work

Yarn [13], the new generation of Hadoop system, is featured with the capability of resource sharing, which supports various workloads including both MPI and MapReduce. Mesos [14] is a resource-sharing platform that efficiently schedules MPI and Hadoop jobs and guarantees performance isolation for these two parallel execution frameworks. Yarn and Mesos allow MPI and MapReduce to share computing resources in datacenters. SCALER allows them to share HDFS storage system.

The Chunk Aware I/O (CHAIO) [16] in our previous work proposed to reorganize I/O requests and eliminate chunk-level contentions in order to efficiently improve N-1 distributed file access on data-intensive file systems. It selects the aggregator using a greedy algorithm to assign destination Datanodes for the blocks accessed by more than one processes. CHAIO is a preliminary study of SCALER and the algorithm is designed to minimize data exchange and to achieve workload balance.

MRAP [22] is a MapReduce based framework that provides customized data access patterns to assist MapReduce applications to migrate data from HPC storage, and efficiently reduces preprocessing overhead of MapReduce applications. It tries to bridge the semantic gap between HPC storage and data-intensive systems. Alternatively, SCALER provides a copy-free approach that enables MPI applications to perform N-1 parallel file writes directly onto a data-intensive file system.

Parallel Log-based File System (PLFS) [23] is a log-based middle-ware, which accelerates checkpoint applications by rearranging N-1 file writes to achieve a bandwidth as good as that of N-N write on an underlying parallel file system; however during read back, the online reconstruction of the log-formatted data to original data may cause performance degradation. This work is extended by [24] to enable HPC applications to use HDFS as storage system, and to provide semantic translation that supports concurrent writes to HDFS. While reading files, the system launches the processes at the same locations as their subfiles. Our approach strives to provide a general interface for concurrent write to HDFS via MPI-IO. SCALER synchronous write function and the reorganized data need no additional metadata operation involved for future read. The data are organized in the same format as that of HDFS.

BlobSeer [25] is a scalable distributed storage system that supports high throughput concurrent file access from MapReduce applications using advanced versioning control. SCALER is able to serve concurrent file access for both MPI based HPC applications and MapReduce applications. Both of them can benefit from our N-1 parallel file write mechanism.

VisIO [26] is an I/O library that utilizes HDFS as the storage for large-scale interactive visualization applications. The applications that VisIO targets perform N-N file read, which is inherently supported by data-intensive distributed file systems. SCALER focuses on providing the N-1 file write feature in HDFS.

In [27], researchers evaluated HPC and Internet service workloads on top of two representative file systems, PVFS2 and CloudStore file system. They revealed the same problem and confirmed performance degradation of CloudStore file system for N-1 workload. SCALER's synchronous and asynchronous I/O mechanisms improve HDFS's poor N-1 file write performance.

Some researchers extended existing parallel file systems, including PVFS2, GPFS, and Lustre, to allow them to support MapReduce applications, such as in [28], [29], and [30]. Researchers compared PVFS and HDFS in [28], proposed an enhanced PVFS with customized data layout, and relaxed consistency in order to provide competitive I/O performance to unmodified Hadoop applications. IBM GPFS-SNC [29] delivered an enterprise alternative of HDFS implementation to achieve better performance and security, with full POSIX compliance. To let HDFS support data access from MPI applications, SCALER adds some important missing features to HDFS. This allows SCALER to successfully exploit the I/O potential of HDFS in terms of N-1 file write.

V. Conclusion

While cloud computing is increasingly becoming a fundamental computing infrastructure, merging the computing power of high performance computers with the data processing capabilities of datacenters becomes more and more in demand. To satisfy the storage requirements of this demand, we have designed and implemented SCALER — a parallel I/O mechanism to allow MPI applications to perform N-1 concurrent file write onto HDFS. The contributions of this study are twofold: the improvement of I/O functionality and I/O optimization.

On I/O functionality: We successfully enabled the missing N-1 parallel file write functionality in HDFS, which allows MPI applications and HDFS to share the same copy of data. This I/O functionality provides the fundamental building block required to merge MPI based HPC computing power with MapReduce data processing power.

On I/O optimization: SCALER supports both synchronous and asynchronous parallel write. A local data buffer mechanism is introduced for asynchronous file writes. This allows better I/O scalability and better utilization of the distributed architecture while maintaining the fault tolerance nature of HDFS. This can significantly benefit HPC applications which have heavy data writing phases.

We have developed the prototype SCALER system. Experimental results prove that the SCALER approach is able to handle various HPC data requirements efficiently. We plan to continue improving the system in the future; through adopting

more optimization techniques at the intra-file level and at the intra-block level.

REFERENCES

[1] IBM Blue Gene team staff, "Overview of the IBM Blue Gene/P Project," *IBM Journal of Research and Development*, vol. 52, no. 1/2, pp. 199–220, Jan. 2008.

[2] J. Dean and S. Ghemawat, "MapReduce: Simplied Data Processing on Large Clusters," in *Proceedings of USENIX Symposium on Operating Systems Design and Implementation*, 2004.

[3] W. Gropp, E. Lusk, and A. Skjellum, *Using MPI: Portable Parallel Programming with the Message Passing Interface*. MIT press, 1999, vol. 1.

[4] L. Dagum and R. Menon, "OpenMP: An Industry Standard API for Shared-memory Programming," *Computational Science & Engineering, IEEE*, vol. 5, no. 1, pp. 46–55, 1998.

[5] 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.

[6] 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*, 2002.

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

[8] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The Google File System," in *Proceedings of ACM Symposium on Operating Systems Principles*, 2003.

[9] 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.

[10] H. Lin, X. Ma, P. Chandramohan, A. Geist, and N. Samatova, "Efficient Data Access for Parallel BLAST," in *Proceedings of IEEE International Parallel and Distributed Processing Symposium*, 2005.

[11] B. Xie, J. Chase, D. Dillow, O. Drokin, S. Klasky, S. Oral, and N. Podhorszki, "Characterizing Output Bottlenecks in a Supercomputer," in *Proceedings of the International Conference for High Performance Computing, Networks, Storage and Analysis (Supercomputing)*, 2012.

[12] Y. Chen, C. Chen, X.-H. Sun, W. D. Gropp, and R. Thakur, "A Decoupled Execution Paradigm for Data-intensive High-end Computing," in *Proceedings of IEEE International Conference on Cluster Computing*, 2012.

[13] 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.

[14] 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.

[15] R. Thakur, W. Gropp, and E. Lusk, "Data Sieving and Collective I/O in ROMIO," in *Proceedings of the Symposium on the Frontiers of Massively Parallel Computation*, 1999.

[16] H. Jin, J. Ji, X.-H. Sun, Y. Chen, and R. Thakur, "CHAIO: Enabling HPC Applications on Data-intensive File Systems," in *Proceedings of the International Conference on Parallel Processing*, 2012.

[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] (2014, Mar.) LibHDFS - Hadoop Wiki. Apache Software Foundation. [Online]. Available: http://wiki.apache.org/hadoop/LibHDFS

[19] Texas Advanced Computing Center. (2014, Mar.) Longhorn User Guide. [Online]. Available: https://www.tacc.utexas.edu/user-services/user-guides/longhorn-user-guide

[20] H. Shan, K. Antypas, and J. Shalf, "Characterizing and Predicting the I/O Performance of HPC Applications Using a Parameterized Synthetic Benchmark," in *Proceedings of the International Conference for High Performance Computing, Networks, Storage and Analysis (Supercomputing)*, 2008.

[21] A. Ching, A. Choudhary, W.-k. Liao, L. Ward, and N. Pundit, "Evaluating I/O Characteristics and Methods for Storing Structured Scientific Data," in *Proceedings of IEEE International Parallel and Distributed Processing Symposium*, 2006.

[22] 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.

[23] J. Bent, G. Gibson, G. Grider, B. McClelland, P. Nowoczynski, J. Nunez, M. Polte, and M. Wingate, "PLFS: A Checkpoint Filesystem for Parallel Applications," in *Proceedings of the International Conference for High Performance Computing, Networks, Storage and Analysis (Supercomputing)*, 2009.

[24] C. Cranor, M. Polte, and G. Gibson, "HPC Computation on Hadoop Storage with PLFS," *Parallel Data Laboratory at Carnegie Mellon University*, pp. 1–9, 2012.

[25] B. Nicolae, G. Antoniu, L. Bougé, D. Moise, and A. Carpen-Amarie, "BlobSeer: Next-generation Data Management for Large Scale Infrastructures," *Journal of Parallel and Distributed Computing*, vol. 71, no. 2, pp. 169–184, 2011.

[26] C. Mitchell, J. Ahrens, and J. Wang, "VisIO: Enabling Interactive Visualization of Ultra-Scale, Time Series Data via High-bandwidth Distributed I/O Systems," in *Proceedings of IEEE International Parallel and Distributed Processing Symposium*, 2011.

[27] E. Molina-Estolano, M. Gokhale, C. Maltzahn, J. May, J. Bent, and S. Brandt, "Mixing Hadoop and HPC Workloads on Parallel Filesystems," in *Proceedings of the 4th Annual Workshop on Petascale Data Storage*, 2009.

[28] 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.

[29] K. Gupta, R. Jain, I. Koltsidas, H. Pucha, P. Sarkar, M. Seaman, and D. Subhraveti, "GPFS-SNC: An Enterprise Storage Framework for Virtual-machine Clouds," *IBM Journal of Research and Development*, vol. 55, no. 6, pp. 2–1, 2011.

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