CHAIO: Enabling HPC Applications on Data-Intensive File Systems

Hui Jin, Jiayu Ji, Xian-He Sun
Department of Computer Science
Illinois Institute of Technology
{hjin6, jji3, sun}@iit.edu

Yong Chen
Department of Computer Science
Texas Tech University
yong.chen@ttu.edu

Rajeev Thakur
MCS Division
Argonne National Laboratory
thakur@mcs.anl.gov

***ABSTRACT*—The computing paradigm of "HPC in the Cloud" has gained a surging interest in recent years, due to its merits of cost-efficiency, flexibility, and scalability. Cloud is designed on top of distributed file systems such as Google file system (GFS). The capability of running HPC applications on top of data-intensive file systems is a critical catalyst in promoting Clouds for HPC. However, the semantic gap between data-intensive file systems and HPC imposes numerous challenges. For example, N-1 (N to 1) is a widely used data access pattern for HPC applications such as checkpointing, but cannot perform well on data-intensive file systems.**

In this study, we propose the CHunk-Aware I/O (CHAIO) strategy to enable efficient N-1 data access on data-intensive distributed file systems. CHAIO reorganizes I/O requests to favor data-intensive file systems and avoid possible access contention. It balances the workload distribution and promotes data locality. We have tested the CHAIO design over the Kosmos file system (KFS). Experimental results show that CHAIO achieves a more than two-fold improvement in I/O bandwidth for both write and read operations. Experiments in large-scale environment confirm the potential of CHAIO for small and irregular requests. The aggregator selection algorithm works well to balance the workload distribution. CHAIO is a critical and necessary step to enable HPC in the Cloud.

Keywords-high-perfomrance computing, MapReduce, data-intensive, distributed file system

I. INTRODUCTION

"HPC in the Cloud" is an emerging computing paradigm that advocates traditional high performance computing (HPC) applications in Cloud environments. Such a computing paradigm enables HPC scientists to run their applications with the flexible pay-as-you-go pricing model. Numerous efforts have been elaborated to investigate the potential of "HPC in the Cloud" computing paradigm [1] [2] [3] [4].

As HPC applications become more and more data intensive, data and its management is recognized as one of the most critical components for scientific computing. Parallel file systems (PFS) are currently the state-of-art storage architecture for typical HPC environments. However, PFS assume dedicated, highly reliable hardware with fast network connectivity, which makes it unrealistic to be deployed in the Cloud that is built on top of commodity hardware.

Data-intensive distributed file systems are storage systems specially designed to support MapReduce frameworks that share the same assumptions with Clouds [5] [6] [7] [8]. As a consequence, data-intensive file systems are a natural choice for the data manipulation of HPC applications in the Cloud.

Unfortunately, data-intensive file systems are not designed with HPC semantics in mind, and few HPC applications can benefit from them directly even if they are not consistency constrained. A large body of HPC applications is either not supported or cannot perform well on data-intensive distributed file systems.

N-1 (N to 1) is a widely used data access pattern for parallel applications such as checkpointing and logging [9]. The N processes usually issue requests to different regions of one shared file, which leads to non-sequential data access, unbalanced data distribution and violates the data locality. All these factors make N-1 based HPC applications not usable on data-intensive file systems.

We have set up an experimental environment to compare the write performance of N-1 and N-N (N to N) data access patterns on two data-intensive file systems, Hadoop distributed file system (HDFS) [7] and Kosmos file system (KFS) [8]. We added components to the IOR benchmark to access data-intensive file systems. We utilize the API provided by libHDFS to access HDFS. However, the N-1 write is not supported by HDFS since libHDFS currently only allows hdfsseek in read only mode [10]. On the other hand, KFS, a C++ based data-intensive file system, supports the N-1 data access by allowing concurrent non-sequential writes to one chunk [8].

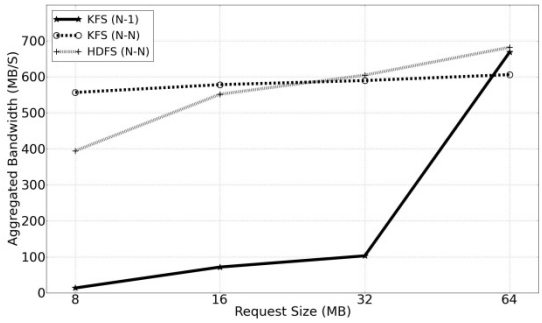


Fig. 1. Performance Comparison of N-1 and N-N (Write)

Fig. 1 compares the performance of N-1 and N-N performance on 16 I/O nodes (chunk servers). The chunk (block) size is 64MB for both file systems. We have 16 processes in each run to issue strided I/O requests. The N-1 curve presents unstable performance with different request sizes. Smaller request sizes lead to more contention in the shared chunk and more performance degradation. The

problem is common for HPC applications as often the request size is much smaller than the chunk size of data-intensive file systems (64MB or higher) [11].

In recognition of the semantic gap between HPC applications and data-intensive file systems, the objective of this research is to bridge the gap and facilitate efficient shared data access of HPC applications to data-intensive file systems.

The contribution of this study is three-fold,

- CHAIO, a chunk-aware I/O strategy to enable efficient N-1 data access patterns on data-intensive distributed file systems, is introduced. CHAIO reorganizes data from different processes to avoid contention and achieve sequential data access.
- An aggregator selection algorithm is proposed to decide a process that issues the I/O requests on behalf of the conflicting processes to balance the I/O workload distribution and regain the data locality.
- CHAIO is prototyped over the Kosmos file system. Extensive experiments have been carried out to verify the benefit of CHAIO and its potential in fostering scalability.

The rest of this paper is organized as follows. Section II introduces the background and related work. We present the design idea and methodology of CHAIO in Section III. The experimental results are presented in Section IV. We conclude this study in Section V.

II. BACKGROUND AND RELATED WORK

HPC applications are parallel scientific applications that rely on low-latency networks for message passing and use parallel programming paradigms such as MPI to enable parallelism [12].

A. *Parallel File System v.s. Data-Intensive File System: A Comparison*

Parallel file systems currently serve as the de-facto file systems for the data manipulation of HPC applications. Representative examples of parallel file systems include IBM GPFS [13], Oracle Lustre [14] and PVFS [15]. HPC applications access PFS via either POSIX interface or MPI-IO, a subset of the MPI-2 specification [16] that enables performance optimizations such as collective I/O [17].

Data-intensive distributed file systems are specialized file systems for data-intensive computing frameworks such as MapReduce [5]. Leading data-intensive file systems include Google file system (GFS) [6], Hadoop file system (HDFS) [7], and Kosmos file system (KFS) [8]. Data-intensive file systems usually come with interfaces to interact with general HPC applications. For Java_based HDFS, libHDFS can be used as the programming interface to support MPI applications [10]. The Kosmos file system offers a native interface to support HPC applications [8]. POSIX imposes many hard consistency requirements that are not needed for MapReduce applications and are not natively supported for data-intensive file systems. MPI-IO [17] was designed on general-purpose file systems and its access to data-intensive file system is currently not supported as well.

Parallel file systems and data-intensive file systems share similar high-level designs. They are both cluster file systems that are deployed on a bunch of nodes. Both of them divide a file into multiple pieces (stripes or blocks/chunks), which are distributed onto multiple I/O servers. However, because these two file systems assume different targeting applications and computing environments, there are several distinguish differences in their design:

File Caching. Client-side cache is an effective approach to improving the bandwidth, especially for small I/O requests. However, the adoption of cache also threatens the data consistency. Data-intensive file systems employ cache for better performance since consistency is not the top design goal. The client accumulates the write requests in memory until its size reaches the chunk size (usually 64MB) or the file is closed, which triggers the write operation to I/O servers. To guarantee consistency and durability, PVFS drops client side cache. GPFS and Lustre support file caching but depends on sophisticated distributed locking mechanism to assure the consistency [18].

Concurrency and Locking. One data chunk is exclusively used by one process/task in MapReduce applications. As such, concurrency is not supported well by data-intensive file systems. Concurrent write operation to one shared file is not supported by HDFS. KFS supports shared file write by placing an exclusive lock on each chunk. All the processes accessing the same chunk compete for the lock to perform I/O operations. Parallel file systems are designed to support POSIX interface and concurrency is inherently supported. GPFS and Lustre leverage more complex distributed locking mechanism to mitigate the impact of caching. For example, GPFS employs a distributed token-based locking mechanism to maintain coherent caches across compute nodes [19]. However, PVFS does not support POSIX semantics for concurrent writes and relies on applications to handle concurrency.

Data Locality. Parallel file systems are designed for typical HPC architecture that separates I/O nodes from compute nodes. File system server processes are deployed on I/O nodes and client processes are deployed on compute nodes. Client processes see server processes as symmetric and data locality is not considered by PFS. On the other hand, the deployment of data-intensive file systems calls for the existence of local disk on each compute node. The client processes of data-intensive file systems should be co-located with server processes to gain high data locality and better I/O performance.

Fault Tolerance. Parallel file systems do not have native fault tolerance support inherently and usually rely on hardware level mechanism like RAID for fault tolerance. Failures could occur frequently for data-intensive file systems that assume commodity hardware at scale. As such,

chunk-level replication is adopted to support the fault tolerance of data-intensive file systems.

B. HPC on Data-Intensive File Systems

VisIO is an I/O library that strives to utilize HDFS as the storage for large-scale interactive visualization applications [20]. VisIO provides a mechanism for using non-POSIX distributed file system to provide linear scaling of I/O bandwidth. The application targeted by VisIO is N-N read, which is naturally supported by data-intensive distributed file systems.

In [11], the authors targeted the scenario of migrating data from HPC storage system to data-intensive frameworks such as MapReduce, and proposed MRAP to bridge semantic gaps. As an extension of MapReduce, MRAP eliminates multiple scans and reduces the number of pre-processing MapReduce programs.

In [21], the authors examined both HPC and Hadoop workloads on PVFS and KFS, and confirmed the performance degradation of N-1 data access pattern on the Kosmos file system.

In [22], the authors enhanced PVFS to match the HDFS-specific optimizations. A non-intrusive shim layer was proposed such that unmodified Hadoop applications can store and access data in PVFS. Several optimizations, including prefetching data, emulating replication and relaxed consistency, were also implemented to make PVFS performance comparable to HDFS.

Nevertheless, all these existing works acknowledged the concurrency issue on data-intensive file systems but did little to overcome it. This research is motivated by the observation that some HPC applications with concurrent I/O access cannot perform well even they are not consistency constrained. This work extends the scope of HPC applications supported by data-intensive file systems, and improves the overall I/O performance of HPC systems as a consequence.

C. N-1 Data Access and its Handling

Modern PFS either leverages distributed locking protocols to achieve consistency for N-1 shared data access (GPFS and Lustre), or does not support POSIX semantics for concurrent writes and relies applications to solve conflicting operations (PVFS). The lock-based solution imposes considerable overhead and several works have been conducted to address this concern.

Collective I/O merges the requests of different processes with interleaved data access patterns and forms a contiguous file region, which is further divided evenly into non-overlapping, contiguous sub-regions denoted as file domains [17]. Each file domain is assigned an aggregator process that issues the I/O requests on behalf of the rest of the processes in that file domain. Collective I/O does not take underlying file system into consideration when deciding file domains and cannot eliminate the conflict chunks. It is still possible that two aggregator processes concurrently access one shared chunk in collective I/O. Users can customize the collective buffer size on each aggregator process by setting parameter cb_buffer_size but that does not solve the problem.

In [18], the authors proposed to partition files based on the underlying locking protocols such that the file domains are aligned to locking boundaries. Data shipping was introduced by GPFS to bind each file block to a single I/O agent that acts as the delegator [23].

PLFS is a virtual parallel log structured file system that sits between parallel file systems and applications and transforms the N-1 data access into N-N pattern [9]. PLFS currently supports parallel file systems such as GPFS and PanFS. Extra efforts need to be taken to adapt PLFS to support data-intensive file systems because the underlying N-N data access potentially imposes more overhead to the metadata management, which is unwanted for data-intensive file systems due to the centralized metadata server.

In [24], the authors presented Blobseer, a storage system that supports efficient, fine-grain access under heavy concurrency. They also demonstrated the potential of BlobSeer in substituting HDFS to support efficient MapReduce applications. BlobSeer adopts versioning instead of locking protocols to handle the concurrency issue.

While demonstrating their success on the N-1 data access of PFS, the ideas of these works can also be applied to data-intensive file systems to alleviate the problem. However, unique features of data-intensive file systems require additional efforts for a complete solution. The selection of the aggregator process is actually a key factor in determining the overall performance of N-1 access on data-intensive file system, especially when the requests from different processes are irregular with varied sizes. However, the selection of aggregator process is not covered by existing PFS optimization techniques since the client processes are usually independent of server processes.

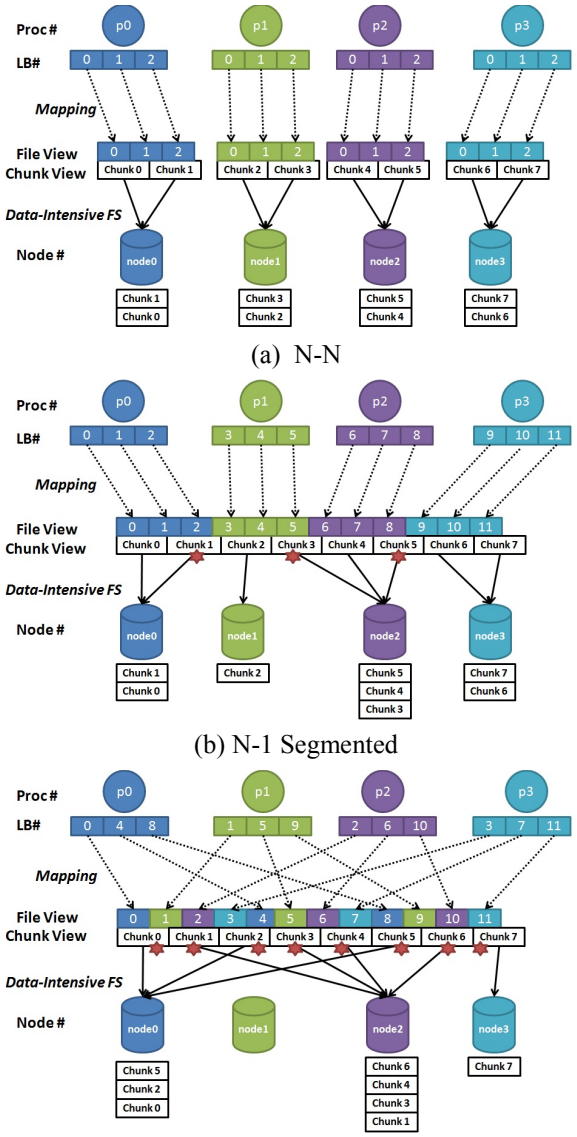
III. CHUNK-AWARE I/O DESIGN AND METHODOLOGIES

A. Data Access Patterns

The data access patterns in HPC applications like checkpointing can be classified as either N-N or N-1 [9]. In N-N data access pattern, each process accesses an independent file with no interference with other processes. Fig. 2(a) demonstrates N-N data access pattern and how it is handled by data-intensive file systems. We assume the chunk size and request size as 64MB and 40MB, respectively, which means a chunk is composed of 1.6 requests. Each compute node has one process, and we have four nodes host the data-intensive file system.

Each process issues three I/O requests, which are marked by logical block number (LB#) to reflect its position in the file. The file view layer in the figure shows the mapping between the requests and their positions in the file. Based on the data access information and chunk size, the requests are translated into chunks by the data-intensive file

system, which are distributed onto the nodes with the consideration of data locality.



(a) N-N

(b) N-1 Segmented

(c) N-1 Strided

Fig. 2. Data Access Patterns and the Handling of Data-Intensive File Systems

In the N-N data access case of Fig. 2(a), each process accesses an individual file and does not incur contention. The I/O workload is evenly distributed such that each node holds two chunks. The downside of the N-N data access pattern, however, is that it involves more files and requires extra cost in metadata management, which is unwanted for data-intensive file systems because of the centralized metadata management.

N-N data access pattern is the ideal case to avoid contention. However, most HPC applications have the processes cooperate with each other and adopt N-1 data access pattern in practice. The processes access different regions of one shared file in N-1 data access. Depending on the layout of regions, N-1 data access can be further classified into two categories: N-1 segmented and N-1 strided [9].

In N-1 segmented data access pattern, each process accesses a contiguous region of the shared file. Fig. 2(b) illustrates N-1 segmented data access pattern and how it is handled by the data-intensive file system. The request size is determined by HPC applications and does not match the chunk size well. The requests from multiple processes could be allocated to one chunk and lead to contention. We term a chunk as conflict chunk if it is accessed by multiple requests. In Fig. 2(b) we have three conflict chunks with id 1, 3 and 5.

Conflict chunks degrade the I/O performance because of the following reasons:

- First, the file system alternates among different requests on the conflict chunk, which violates the sequential data access assumption of data-intensive file systems.
- Furthermore, the conflict chunk is composed of requests from multiple compute nodes and only one node is selected to host the chunk. Data locality is not achieved for the requests from other compute nodes. For example, for chunk 3 of Fig. 2(b), the request from p1 (LB# 5) is not a local data access.
- The chunk placement is decided by the first request with the consideration of data locality. This mechanism results in unbalanced data distribution. In Fig. 2(b) we can observe that three chunks (3, 4, and 5) are allocated onto node 2 while node 1 only has one chunk. It is more critical for data-intensive file systems to balance the chunk distribution since the chunk size is normally sized 64MB or higher, which is magnitudes higher than the strip size (usually 64KB) of PFS.

In the N-1 strided data access pattern, each process issues I/O requests to the file system in an interleaved manner. As illustrated in Fig. 2(c), strided data access has a higher probability to incur conflict chunks and has greater impact in degrading the performance. Actually, all the 8 chunks have contention in the case shown in Fig. 2(c). The data locality and balanced data distribution will be further deteriorated as well. Fig. 2(c) demonstrates the worst case that node 2 has 4 chunks, while no chunk is allocated to node 1.

In practice, N-1 strided is a more common data access pattern than N-1 segmented for HPC applications such as checkpointing [9].

This study is motivated by the performance issues with the N-1 data access on data-intensive file systems. As demonstrated in the following subsections, the proposed new CHAIO strategy rearranges the I/O requests to eliminate conflict chunks, achieve data locality and balance data distribution.

B. CHAIO Design, Methodology and Analysis

1) CHAIO Design

The basic idea of CHAIO is to reorganize the I/O requests such that each chunk is accessed by one process to eliminate contention. Fig. 3 shows how CHAIO handles the scenario shown in Fig. 2(c).

We add a communication phase to exchange data among processes. One process is selected as an aggregator process for each conflict chunk. The aggregator collects data from the non-aggregator processes accessing the same conflict chunk, and issues the I/O requests to the file system. From the perspective of data-intensive file systems, each chunk is accessed by the aggregator process only. Even though CHAIO introduces slight message passing overhead, it improves the performance significantly by removing the contention and marshaling the I/O requests.

With CHAIO, each chunk has only one aggregator process that acts as the file system client to issue the I/O request, as shown in the I/O phase of Fig. 3. The data locality is assured since the file system by default allocates the chunk to the node where the aggregator process resides.

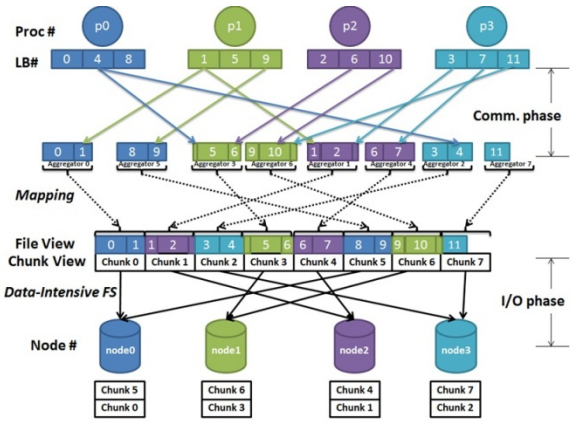


Fig. 3. N-1 Strided write with CHAIO

The N-1 read of CHAIO is performed in the reverse order. The aggregator first gets data from the file system and distributes the data to the corresponding processes.

2) Aggregator Selection Algorithm

In this subsection we introduce the aggregator selection algorithm that balances the chunk distribution among the nodes.

The aggregator selection algorithm takes the data access pattern, chunk size and process distribution as input. The output of the algorithm is the decision of the aggregator process for each chunk.

The pseudo code of the aggregator selection algorithm is listed in Algorithm 1. The node with less conflict chunks has higher priority to be selected to host the aggregator process. If multiple chunks are serviced by the selected node, the chunk with the least service nodes is selected. The algorithm is a greedy algorithm that is biased toward the node or chunk with least matching options.

The algorithm sets a threshold to limit the aggregator processes on each node and guarantee balanced I/O workload distribution. A node will not be selected to run more aggregator processes if it is already fully loaded with the threshold number of aggregator processes. The threshold value will be increased if there is no eligible node but not all the chunks have been allocated yet.

Multiple processes from the selected node may access the same conflict chunk in a multicore architecture. The process with the largest I/O request size to minimize the message passing overhead will be selected in this case. Fig. 3 shows an example where each node has two chunks with the assistance of the aggregator selection algorithm.

Definition:

A chunk is *allocated* if its aggregator process has been decided.

Chunk *c* is *serviced* by node *s* if there is at least one I/O request from *s* to *c*.

Terminology:

C is the collection of the unallocated chunks.

a(s) is the number of chunks that have been allocated to node *s*.

n(s) is the number of unallocated chunks that are served by node *s*.

g(c) is the number of nodes that service chunk *c*.

p is the number of conflict chunks.

q is the number of nodes.

f(c)=s means we select a process on node *s* as the aggregator process for chunk *c*.

Algorithm:

Initialize *C, n(s), g(c), p* and *q* based on the data access info, chunk size and the process distribution.

a(s)=0

threshold= $\lceil p / q \rceil$

while (*size(C)>0*)

find the node *s* with *min(a(s)+n(s))* and satisfies *a(s)≤threshold.*

if *s=null*

increase *threshold* by 1

continue

end if

for each chunk serviced by node *s*, find the chunk *c* with m*in(g(c))*.

f(c) =s

a(s)=a(s)+1

for each node that services *c*,

remove *c* from its chunk list

set *n(s)=n(s)-1*

end for

remove *c* from *C*.

end while

Algorithm 1. Aggregator Selection Algorithm

3) CHAIO Implementation

CHAIO can be implemented either inside the application code or in the I/O middle-ware layer such as MPI-IO. CHAIO takes the data access information, chunk size and the process distribution information as input. The data access information can be obtained from the application or from MPI primitives, i.e., MPI_File_get_view. The data-intensive file system needs to expose the chunk size information to CHAIO, which is trivial to implement. We also need to know the process distribution information that indicates the mapping between processes and nodes, usually in a round-robin or interleaved manner. We can obtain this information easily from the job scheduler.

Each process first captures the aforementioned input and carries out the aggregator selection algorithm. The output of the algorithm is organized in a hash table data structure which stores the chunk id and the corresponding rank id for the aggregator process.

When a process carries out an I/O request, it first calculates the chunk id of the I/O request and checks the hash table derived from the aggregator selection algorithm. If the chunk id of the I/O request matches one entry in the hash table, it means the I/O request is involved in a conflict chunk and we need to take action. If the rank id of the process matches the aggregator process id from the hash table, the process will receive data from other processes and then issue the I/O request of the entire chunk to the file system. If the process is not selected as the aggregator, it simply sends the data to the aggregator process.

We use nonblocking send for the non-aggregator processes so that the following I/O requests are not blocked by the message passing. Blocked receive is adopted by the aggregator to guarantee that the process is carrying out one I/O request at a time.

4) CHAIO Analysis

There are potential alternative solutions to the problem of N-1 data access besides the CHAIO approach. The straightforward solution is to adopt methodologies such that one I/O request generates one individual chunk in the data-intensive file systems. To implement the idea, we can adapt the chunk size to the I/O request size in the file system.

The primary concern with this approach is that the metadata management overhead it introduces to the file system. The number of chunks is equal to the number of I/O requests, which could be significant considering small request sizes from HPC applications [11]. On the other hand, the file system namespace and file Blockmap of the data-intensive file system is kept in the memory of the centralized metadata server (Namenode). A large number of chunks could overwhelm the centralized metadata management of the data-intensive file system and degrade I/O performance.

CHAIO aggregates multiple I/O requests of one chunk to form sequential data access and does not increase the metadata management overhead to the file system. CHAIO is implemented at either the application level or I/O middle-ware level and does not introduce complexity to the data-intensive file system.

Data-intensive applications usually adopt multiple replicas of one chunk to achieve fault tolerance. The data locality and balanced data distribution are not concerned by non-primary replicas since they select nodes randomly to store the data. Multiple replicas do not obscure the advantages of CHAIO for read operations. The I/O request returns after reading one replica from the file system and CHAIO does help to alleviate contention in this scenario. Furthermore, the performance of the primary (first) replica is improved by CHAIO for write operations. The performance of the first replica is usually more critical than others since it concerns the application elapsed time. It is a widely used optimization technique for replica_based file systems to return to the application after the first replica is completed and process the rest of the replicas in parallel with the applications [25].

IV. EXPERIMENTAL RESULTS AND ANALYSIS

A. Experiment Setup

We have carried out experiments on a cluster of 65 Sun Fire Linux-based cluster test bed. Each node is equipped with dual 2.7GHz Opteron quad-core processors, 8GB memory and 250GB SATA hard drive. All the nodes are connected with 1 Gigabit NICs in a fat tree topology. One node dedicated as the job submission node and the metadata server of the Kosmos file system. The experiments were tested with Open MPI v1.4 on Ubuntu 9.04 with kernel 2.6.28.10.

KFS is utilized as the underlying data-intensive file system in the experiments. We use IOR-2.10.2 from Lawrence Livermore National Laboratory as the benchmark to evaluate the performance [26]. We have added a KFS interface to the IOR benchmark to enable data access to the Kosmos file system. The KFS interface was implemented with the methodology similar to other interfaces of IOR such as POSIX. We implement CHAIO in IOR benchmark and compare its performance with the original IOR benchmark. We set the chunk size at 64MB in the experiments and each chunk has one replica by default.

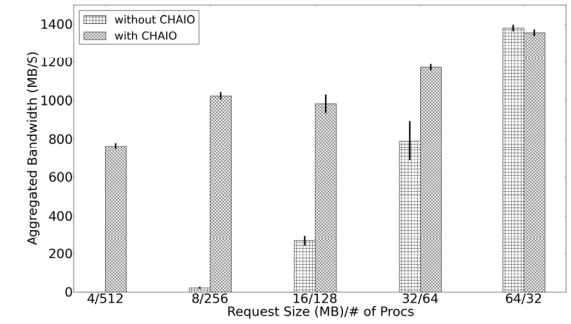
B. Performance with Different Request Sizes

We keep the number of nodes fixed at 32 in Fig. 4 and study the performance with different I/O request sizes. We fix the size of the shared file at 32GB and each process issues 16 interleaved I/O requests to implement N-1 strided data access. The number of processes is varied accordingly with different request sizes. We run each setting 10 times in the experiments, get the mean and standard deviation of the aggregated bandwidth and plot them in the figure. The standard deviation is reflected by the error bars.

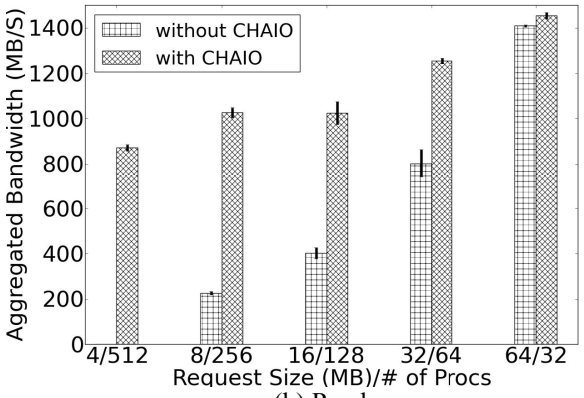
The write performance is illustrated in Fig. 4(a). A smaller size of I/O requests means more contention in conflict chunks and leaves more opportunity for performance improvement in CHAIO. Actually, when the

request size is 4MB, it is not possible to have successful N-1 data access by the existing approach due to the overwhelming contention on the conflict chunks. We were able to get successful data access for request size of 8MB but the performance was still very poor (22.92MB/s). CHAIO achieved a write bandwidth of 983.52 MB/s for 16MB request size, which is three times higher than 270.6 MB/s, the bandwidth achieved by the existing approach.

When the request size is 64MB, CHAIO does not show advantage in bandwidth performance. Since the request size is equal to the chunk size, there is no contention on the chunks and the benefit of CHAIO cannot be observed.



(a) Writes



(b) Reads

Fig. 4. Performance with Different Request Sizes

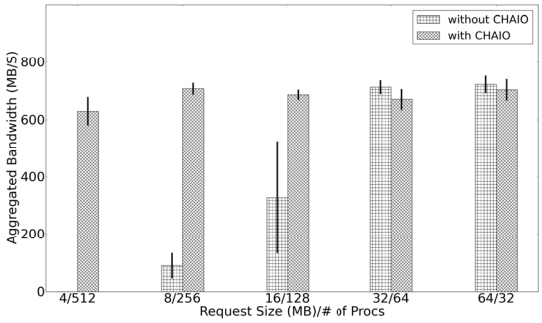
When the request size is 4MB, CHAIO shows less bandwidth than in the case with larger request sizes. There are possibly two factors leading to the performance degradation. First, a smaller request size needs more data exchange in the communication phase. Furthermore, each 8-core node is overloaded with 16 processes when the request size is 4MB and could considerably harm the overall I/O performance. Our later analysis in subsection IV- F shows that the impact of the small request size incurs little overhead and we can attribute the performance degradation to the second factor.

Fig. 4(b) compares the read bandwidth of CHAIO and the existing approach. It is easy to observe the advantage of CHAIO over the existing approach. The bandwidth of reads more than doubles the existing approach when the request size is 16MB or less.

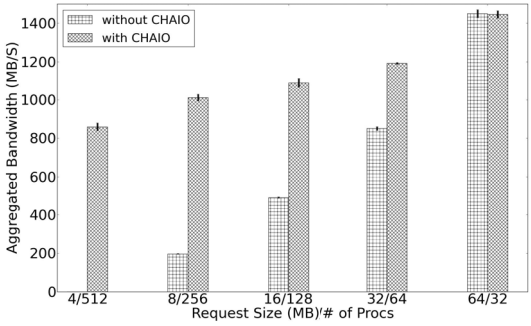
C. *Performance with Two Replicas*

We set the number of replicas as two and demonstrate its performance with different request sizes in this set of tests, and the results are shown in Fig. 5.

Though the advantage of CHAIO is less significant for more replicas, as discussed in subsection IV-B, it still presents satisfactory write performance improvement as shown in Fig. 5(a). CHAIO achieved a write bandwidth of 685.7 MB/s, which doubled the existing approach, 328.9 MB/s.



(a) Writes



(b) Reads

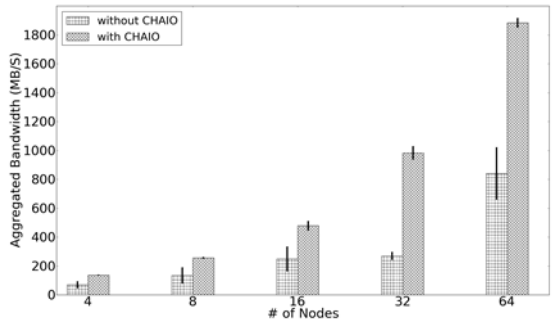
Fig. 5. Performance with Two Replicas

We also observe that both CHAIO and the existing approach have bandwidth degradation for two replicas than the one replica case of Fig. 4(a). For example, in the contention-free case with 64MB request size, the write bandwidth of two replicas is about 700MB/s, which is considerably lower than the 1300MB/s bandwidth in the case of one replica. A detailed study reveals that the performance degradation is due to node-level contention. When the number of replicas increases to two or more, each node not only services the first chunk, but the non-primary copies will also compete for the node and incur node-level contention. This study focuses on the chunk-level contention problem caused by N-1 data access. The node-level contention problem is on our roadmap for future studies.

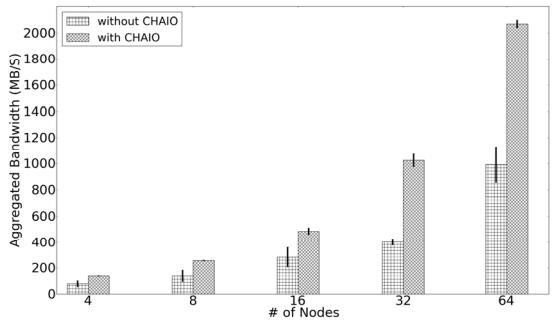
As illustrated in Fig. 5(b), read performance is not impacted by multiple replicas and CHAIO outperforms the existing approach consistently.

D. Performance with Different Number of Nodes

In Fig. 6 we vary the number of data nodes from 4 to 64 and observe its impact on the performance. For each node we spawn 4 MPI processes to carry out the I/O requests. Each process issues 16 interleaved I/O requests to one shared file to implement N-1 strided writes. The I/O request size is kept at 16MB.



(a) Writes



(b) Reads

Fig. 6. Performance with Different Number of Nodes

The write bandwidth is presented in Fig. 6(a). The bandwidth of CHAIO is two-fold higher than that of the existing approach.

The existing approach also exhibits more variance in bandwidth than CHAIO, which is caused by the unbalanced chunk distribution. In the existing approach, the conflict chunk selects the node based on the first I/O request coming into the file system, which results in uncertainties in the chunk distribution and a large variance in bandwidth.

Fig. 6(b) compares the read bandwidth of the CHAIO approach with the existing approach and confirms the advantage of CHAIO as well. CHAIO achieved a bandwidth two times higher than the existing approach for all cases.

E. Scalability Analysis

We use the performance with 4 nodes as the baseline and plot the speedup for each scenario in Fig. 7. CHAIO performs well in terms of scalability for both write and read operations, which is close to the ideal speedup case. We can observe that the existing approach achieved speedup as well; however, it is not stable. For example, there is no much improvement between 16 nodes and 32 nodes for the existing approach. A detailed study reveals that due to the uneven chunk distribution, a small set of the nodes is constantly selected as the chunk servers and hurts the scalability. The CHAIO approach reduces the access contention and regains the access locality by rearranging requests, and achieves better and stable scalability.

F. Overhead Analysis in Large-Scale Computing Environment

We have shown the performance improvement of CHAIO in terms of both bandwidth and scalability in previous subsections. We have performed tests to evaluate the potential of CHAIO in large-scale computing environment as well. To achieve that, we have measured the communication phase cost of CHAIO on the SiCortex machine at Argonne National Laboratory [27]. The SiCortex system is composed of 972 nodes, each node is equipped with 6 cores and 4GB memory. The interconnect bandwidth is 1300 MB/s with a novel network topology of "Kautz graph" [28].

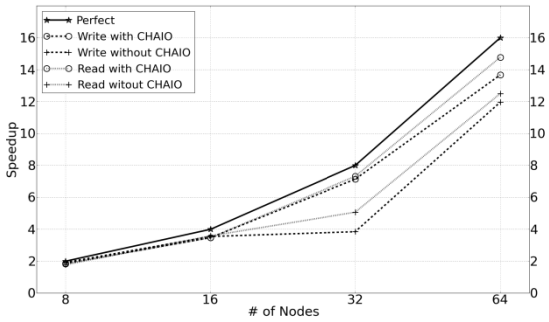


Fig. 7. Scalability Analysis

Data exchange cost is the primary, if not only, overhead introduced by CHAIO and is used as the metric to study the CHAIO overhead. 1 Fig. 8 reports the measure data exchange cost of one conflict chunk with different request sizes and number of processes. It can be observed that the data exchange cost is kept less than 1 second for all cases which is a minor overhead. The data exchange overhead is increased by 0.15 seconds when reducing the request size from 8MB to 16KB, which is still trivial compared with the I/O performance CHAIO helps to improve. The experimental tests confirm that CAHIO improves the I/O bandwidth considerably with only introducing a minor communication overhead.

We keep the number of processes at 2048, vary the request size from 1MB to 32MB, and demonstrate the data exchange overhead in Fig. 9. In this set of experiments, we

[1] We eliminated the I/O phase in the SiCortex experiments and only measured the communication phase cost for overhead analysis. The lack of local disk and the job scheduler of SiCortex make it impractical, if not impossible, to deploy Kosmos file system on SiCortex.

have irregular request sizes that cannot divide the default block size of 64MB perfectly, e.g., 3MB, 5MB, 10MB, etc. Such irregular sized requests result in irregular conflict patterns and impose challenges to the aggregator selection algorithm. As reported in Fig. 9, we do not observe much performance deviation of irregular request sizes compared to the regular ones. This infers that CHAIO is applicable to various request sizes with trivial data exchange cost.

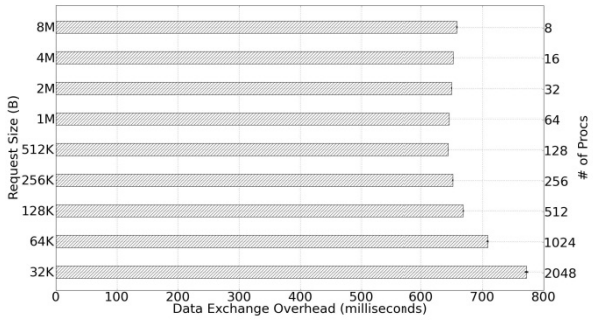


Fig. 8. Data Exchange Overhead (1 Conflict Chunk)

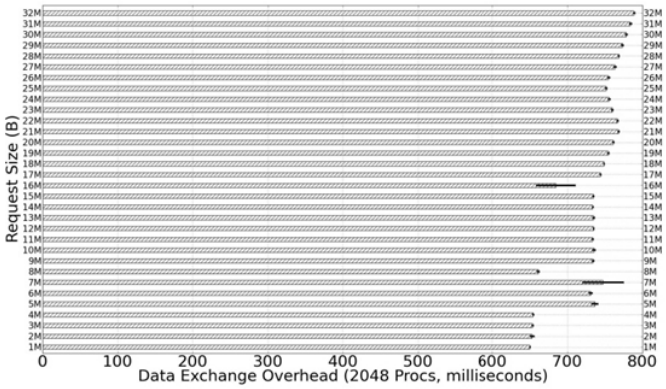


Fig. 9. Data Exchange Overhead with 2048 Procs

G. Load Balance Evaluation

The primary objective of the aggregator selection algorithm is load balance. CHAIO should balance the number of chunks allocated to each data node to gain better parallelism and performance. As such, we evaluate the load balance of CHAIO and compare it with the existing approach. We first calculate the ideal data layout that conflict chunks are evenly distributed to all the involved data nodes. The data layout of CHAIO and the existing approach are also calculated. We next compute the Manhattan distances between the ideal data layout and the two scenarios (with/without CHAIO), which is used as the metrics of unbalance [24]. Fig. 10 reports the degree of unbalance by varying request size from 1MB to 32MB with 2048 processes on SiCortex. CHAIO performs significantly better in balancing the workload among each data node than the existing approach. In particular, we observe perfect load balance (0 in y-axis) for 17 out of the 32 samples, including 11 irregular requests that cannot divide 64MB perfectly.

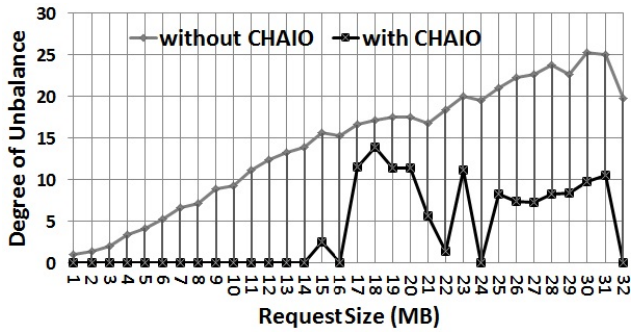


Fig. 10. Load Balance Evaluation (2048 Procs)

V. CONCLUSIONS AND FUTURE WORK

There is a surging interest in merging traditional HPC framework with MapReduce from different perspectives. Motivated by the "HPC in the Cloud" computing paradigm, this paper strives to promote the adaption of HPC applications on top of MapReduce file systems.

N-1 data access is a widely used pattern for HPC applications. Unfortunately, it is not supported well by the exiting data-intensive file systems. In this research, we propose a systematic approach to enabling efficient N-1 data access on data-intensive file systems.

We have identified three factors that degrade the performance of N-1 data access on data-intensive file systems: non-sequential data access, uneven chunk distribution, and the violation of data locality. A chunk-aware I/O (CHAIO) strategy was proposed to address these issues and overcome the challenge. The CHAIO introduces an aggregator process that collects data from multiple processes and issues the I/O requests to the file system to achieve sequential data access and data locality. An aggregator selection algorithm is proposed to balance the chunk distribution among nodes. CHAIO can be implemented at either the application level or the I/O middle-ware level and does not introduce complexity to the underlying file systems.

We have prototyped the CHAIO idea, and conducted experiments with the IOR benchmark over the Kosmos file system. Experimental results show that CHAIO improves both the write and read performance significantly. The overhead analysis shows that CHAIO introduces little overhead for small request sizes and has a real potential for large-scale computing environment. The performance gain of CHAIO is robust to different requests size. The aggregator selection algorithm works efficiently for load balance.

In the future, we plan to enable MPI-IO to interact with data-intensive file systems and provide support to general HPC I/O operations in the Cloud.

ACKNOWLEDGEMENTS

This research was supported in part by National Science Foundation under NSF grant CCF-0621435, CCF-0937877, CNS-0751200, and in part by the Office of Advanced Scientific Computing Research, Office of Science, U.S.

Department of Energy, under contract DE-AC02-06CH11357. We are thankful to Dr. Ioan Raicu of Illinois Institute of Technology and the MCS department at Argonne National Lab for the support to run large-scale simulations on the SiCortex computing system.

REFERENCES

[1] Magellan Project: A Cloud for Science. [Online]. http://magellan.alcf.anl.gov/

[2] Edward Walker, "Benchmarking Amazon EC2 for High-Performance Scientific Computing," in *Usenix Login*, 2008.

[3] Q. He, S. Zhou, B. Kobler, D. Duffy, T. McGlynn, "Case Study for Running HPC Applications in Public Clouds," in *Proc. of 1st Workshop on Scientific Cloud Computing (ScienceCloud), colocated with HPDC*, 2010.

[4] HPC in the Cloud. [Online]. http://www.hpcinthecloud.com/

[5] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," *Communications of the ACM - 50th anniversary issue: 1958-2008*, vol. 51, no. 1, 2008.

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

[7] Hadoop Distribute Filesystem Website. [Online]. http://hadoop.apache.org/hdfs/

[8] Kosmos Distributed Filesystem (CloudStore) website. [Online]. http://code.google.com/p/kosmosfs/

[9] 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 *Proc. of the International Conference for High Performance Computing, Networks, Storage and Analaysis (Supercomputing)*, 2009.

[10] libHDFS source code. [Online]. https://github.com/apache/hadoop-hdfs/blob/trunk/src/c++/libhdfs/hdfs.h

[11] S. Sehrish, G. Mackey, J. Wang and J. Bent, "MRAP: A Novel MapReduce-based Framework to Support HPC Analytics Applications with Access Patterns," in *Proc. of the 19th ACM International Symposium on High Performance Distributed Computing (HPDC)*, 2010.

[12] MPI Website. [Online]. http://www.mcs.anl.gov/research/projects/mpi/

[13] F. Schmuck and R. Haskin. , "GPFS: A Shared-Disk File System for Large Computing Clusters," in *Proc. of the 1st USENIX Conference on File and Storage Technologies (FAST)*, 2002.

[14] Lustre File System Website. [Online]. http://www.lustre.org

[15] PVFS2 Website. [Online]. http://www.pvfs.org/

[16] MPI-2: Extensions to the Message-Passing Interface. [Online]. http://www.mpi-forum.org/docs/mpi-20-html/mpi2-report.html

[17] R. Thakur, W. Gropp and E. Lusk, "Data Sieving and Collective I/O in ROMIO," in *Proc. of the 7th Symposium on the Frontiers of Massively Parallel Computation*, 1999.

[18] Wei-keng Liao, Choudhary, A., "Dynamically adapting file domain partitioning methods for collective I/O based on underlying parallel file system locking protocols," in *Proc. Supercomputing*, 2008.

[19] Jean-Pierre Prost, Richard Treumann, Richard Hedges, Alice Koniges and Alison White, "Towards a High-Performance Implementation of MPI-IO on Top of GPFS," in *Proc. of 6th International Euro-Par Conference on Parallel Processing (Euro-Par)*, 2000.

[20] C. Mitchell, J. Ahrens, and J. Wang, "VisIO: Enabling Interactive Visualization of Ultra-Scale, Time Series Data via High-Bandwidth Distirburted I/O Systems," in *Proc. of 25th IEEE International Parallel & Distributed Symposium (IPDPS)*, 2011.

[21] E. Molina-Estolano, M. Gokhale, C. Maltzahn, J. May, J. Bent and S. Brandt, "Mixing Hadoop and HPC workloads," in *Proc. of Parallel Data Storage Workshop (PDSW)*, 2009.

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

[23] Jean-Pierre Prost, Richard Treumann, Richard Hedges, Bin Jia and Alice Koniges, "MPI-IO/GPFS, an Optimized Implemetation of MPI-IO on top of GPFS," in *Proc. of Supercomputing*, 2001.

[24] Nicolae, B., Moise, D., Antoniu, G., Bouge, L., Dorier, M. , "BlobSeer: Bringing high throughput under heavy concurrency to Hadoop Map-Reduce applications," in *Proc. of IEEE International Symposium on Parallel & Distributed Processing (IPDPS)*, 2010.

[25] S. Al-Kiswany, M. Ripeanu, S. S. Vazhkudai, and A. Gharaibeh, "stdchk: A Checkpoint Storage System for Desktop Grid Computing," in *Proc. of The 28th International Conference on Distributed Computing Systems (ICDCS)*, 2008.

[26] IOR Benchmark Website. [Online]. http://sourceforge.net/projects/ior-sio/

[27] SiCortex at Argonne National Laboratory. [Online]. http://www.mcs.anl.gov/hs/hardware/sicortex.php

[28] Kautz Graph Wiki. [Online]. http://en.wikipedia.org/wiki/Kautz_graph