2014 14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing

Iteration Based Collective I/O Strategy for Parallel I/O Systems

Zhixiang Wang, Xuanhua Shi, Hai Jin, Song Wu
Services Computing Technology and System Lab
Cluster and Grid Computing Lab
Huazhong University of Science and Technology
Wuhan, 430074, China
{wangzhx123, xhshi, hjin, wusong}@hust.edu.cn

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

Abstract*—MPI collective I/O is a widely used I/O method that helps data-intensive scientific applications gain better I/O performance. However, it has been observed that existing collective I/O strategies do not perform well due to the access contention problem. Existing collective I/O optimization strategies mainly focus on the I/O phase efficiency and ignore the shuffle cost that may limit the potential of their performance improvement. We observe that as the size of I/O becomes larger, one I/O operation from the upper application would be separated into several iterations to complete. So, I/O requests in each file domain do not necessarily issue to the parallel file system simultaneously unless they are carried out within the same iteration step. Based on that observation, this paper proposes a new collective I/O strategy that reorganizes I/O requests within each file domain instead of coordinating requests across file domains, such that we can eliminate access contentions without introducing extra shuffle cost between aggregators and computing processes. Using benchmark workloads IOR, we evaluate our new strategy and compare with the conventional one. The proposed strategy achieves up to 47%–63% I/O bandwidth improvement compared to the existing ROMIO collective I/O strategy.

Keywords- collective I/O; parallel system; access contention; iteration

I. INTRODUCTION

Many HPC scientific applications become more and more data-intensive, and I/O performance is considered as a critical bottleneck in scientific discovery. *Data intensive science* is a concept initially defined by Jim Gray in his seminal work on the *Fourth Paradigm* of scientific discovery [1]. In scientific applications such as earthquake simulation (SCEC [2]) and combustion (S3D [3]), the analysis data is commonly represented with a multi-dimensional array-based data model. They normally only need to access a sub-array of it during an analysis. For example, for a combustion study with S3D [2, 4], the computation is conducted upon a 1408×1408×1100 array variable, but the majority of analysis is performed on the orthogonal planes of it (either 1408×1408 or 1408×1100 points). As the multidimensional computing volume is usually stored in a canonical order in files, such access may request many noncontiguous data blocks in an interleaving fashion when accessing the array volume along some dimensions from multidimensional arrays.

To handle this access pattern efficiently, the MPI-IO that includes collective I/O optimization [5, 6] is commonly used to combine those noncontiguous requests into a large contiguous one to improve the overall parallel I/O performance. At the same time, many scientific applications currently run on parallel file systems like Lustre [7], GPFS [8], and PVFS [9] in order to have highly concurrent access to its underlying data. However, as the I/O size becomes larger, it may lead to new challenges when performing MPI-IO optimization in parallel file systems [10, 11] as discussed below.

First, when processes carry out large contiguous I/O requests simultaneously, they need to communicate with all the I/O servers, which causes significant communication overhead and resource contentions among those processes. Second, most of the current parallel file systems use a lock mechanism [12] to maintain POSIX data consistency, which may cause lock contentions and serialize the I/O requests at the I/O servers [13]. Third, when the multiple processes simultaneously send their requests to the I/O servers, it may cause them to be served on the disk in an interleaving fashion, which makes the disk head have to move back and forth to retrieve the data. These issues can significantly degrade the collective I/O performance.

In order to alleviate the access conflicts, numerous different strategies have been studied, such as redistributing data among aggregators to make the way file domains organize match the data layout on the parallel file system [14-16]. However, most existing research did not consider the issue comprehensively. As we know, the most widely used implementation of collective I/O is ROMIO's two-phase I/O [5], which consists of both I/O phase and shuffle phase. Most of these existing strategies focus on promoting efficiency of the I/O phase with little attentions paid on the shuffle cost. These one-sided designs may have hidden flaws that may impair the potential of their effectiveness for improving the overall collective I/O performance [17, 18]. In section III, we will show an example to illustrate it.

In this paper, we provide a new perspective of optimization that can solve the access contention problem without increasing shuffle complexity. We observe that, when the size of file domain is larger than the buffer size set by the program, collective I/O should partition all of the file domain data requests into several iterations to complete [5]. This means that a finer-grained parallelism in a collective

978-1-4799-2784-5/14 $31.00 © 2014 IEEE
DOI 10.1109/CCGrid.2014.61

I/O operation is the iteration instead of the whole file domain. Our idea is based on the fact that I/O requests in each file domain do not necessarily issue to the parallel file system simultaneously unless they are carried out within the same iteration step. So, we do not have to reorganize data between aggregator processes; instead, we can perform the data reorganization within the file domain of each aggregator. With our analysis in the paper, we also infer that this new collective I/O strategy will not introduce new shuffle cost while other optimization methods may do.

Our proposed solution is based on iterations, which provide a finer-grained parallelism and more flexibility for the reorganization method to solve access contention problems mentioned above. The contribution of our design is three-fold. First, by looking into the collective I/O operation process, we find that I/O data from the same collective operation is not issued to the file system as a whole. They may be separated into several iterations. Second, we propose a new data reorganizing design which builds on the iteration process and decouples the optimization of the I/O phase and shuffle phase so that we can address the contention problem without introducing extra shuffle cost. Third, we have carried out experimental tests to verify the effect of the proposed strategy and the results have confirmed the advantage and potential of the proposed new approach.

The rest of the paper is organized as follows. Section II first reviews the background about collective I/O and some concurrent issue challenges. Section III provides a motivation example and shows how other optimization strategies may introduce extra communication cost; then we introduce our IBCS design and implementation method. Experimental results are discussed in section IV and related work is discussed in section V. Finally, section VI summarizes this study and discusses future work.

II. BACKGROUND

In this section, we first introduce some concepts about collective I/O and some specific details about its two-phase implementation. Then, we discuss some concurrent access problems.

A. *Collective I/O and Two-Phase Implementation*

MPI-2/MPI-3 specification [6] extends the parallel programming interfaces by including file I/O interfaces. MPI-IO supports many parallel I/O operations and collective I/O is one of them. Instead of performing many non-contiguous and small I/O requests, collective I/O requires multiple processes to cooperate with each other to form large and contiguous ones. It is one of the most important I/O access optimizations expected to help bridge the gap between the I/O throughput and computing performance of parallel applications. The most well-known and widely used implementation of collective I/O is ROMIO [5], which is developed and maintained by Argonne National Laboratory. ROMIO uses a technique termed two-phase I/O strategy [5] to implement collective I/O operations, which as the name suggests, consists of two main phases: shuffle phase and I/O phase.

Take the collective write operation for example. Before these two phases, a subset of processes is chosen as I/O aggregator which should interact with the I/O servers directly. The whole file region accessed by processes is also divided into equal contiguous regions, which are referred as *File Domains* (FDs), and then assigned to those aggregators. Each aggregator should carry out I/O requests for the data in its own file domain during the I/O phase and communicate with those processes that have data residing in its file domain during the shuffle phase. For the collective write operation, the shuffle phase is carried out before the I/O phase. During the shuffle phase, aggregators need to communicate with those computing processes in order to gather their access information and requesting data. After that, during the I/O phase, all aggregators make write calls to the underlying file system based on its file domain.

In ROMIO, there are two main user-controllable parameters for the collective I/O: the number of aggregators and the size of the temporary buffer each aggregator needs for the two-phase I/O. For the nodes that contain multiple CPU processes, ROMIO picks one process from each node as an aggregator for the others by default, and the collective I/O buffer size is set to 4MB. The user can change these parameters via MPI-IO's hint mechanism in the applications.

If the I/O size is too large to fit in a single buffer size, ROMIO will perform the two-phase process in several iterations to complete the whole I/O operation. Each aggregator first calculates the offset length of the first and last bytes of its file domain. It then divides the length by the maximum size allowed for the temporary buffer. The result is the number of times (denoted as *ntimes* in ROMIO) it needs to perform I/O. If all aggregators do not have the same *ntimes* value, then a global *max_ntimes* value is calculated. Each process must therefore be ready to participate in the communication phase *max_ntimes* number of times (but not necessary for the I/O phase). At the same time, synchronization is used between iterations to guarantee that all aggregators have finished before moving on to the next iteration step [5].

B. *Concurrent Access Problem*

Many HPC applications run on the parallel file systems like Lustre and PVFS. These file systems provide significant parallelism by distributing data across I/O servers so that I/O requests can be served at the different I/O servers at the same time.

However, in real applications, concurrent access problems can occur. First, distributed parallel file systems such as Lustre and GPFS are originally designed to support the POSIX file consistency semantics for concurrent shared-file access, and they depend on some locking systems to enforce that constraint. For example, in Lustre, the locking protocol requires that a lock must be obtained before any file stripe can be modified or written into the client-side cache. So, when two processes are writing data to the same file stripe, even if it is not overlapped, the two writes have to be serialized due to the lock contentions which results a performance degrade.

Apart from that, there exist some other contention problems arising from multiple aggregator processes accessing data on the same I/O servers concurrently [19]. An I/O server may receive I/O requests randomly from any aggregator. When many aggregator processes access data residing on the same I/O servers, this will cause the contentions of the file system and network resources, which bring in significant communication overhead. As file size and the number of processes become larger, such degradation of performance will become more significant [19]. Besides, when each I/O server receives and serves concurrently multiple requests from different processes without coordinating the order for them, the request arrival order at the I/O servers would be uncertain or essentially random which may also increase the disk head's seek time when accessing such interleaving I/O requests [20].

III. Iteration-Based Collective I/O Strategy (IBCS) Design and Implementation

In this section, we present the design of the proposed *iteration-based collective I/O* strategy. First we present a motivation example and then introduce the new collective I/O strategy. We present the implementation methodology and analysis thereafter.

A. A Motivation Example

Fig. 1 demonstrates a representative example of a typical access pattern and data layout on I/O servers in current HPC systems. To keep the example simple, we assume to have six processes (*P1~6*) running at three different nodes with one process from each node acting as the aggregator, i.e. *P1*, *P3* and *P5* are aggregators. We also assume that there are three I/O servers (denoted as OST using Lustre terms in the Fig. 1). The file data is assumed to be distributed across these servers in the most common round-robin fashion.

By the default file domain partitioning method, the whole access region is evenly partitioned into three file domains and assigned to the three aggregators. Then each aggregator takes several iterations to issue requests of its file domain to the file system depending on the buffer size. For example, in Fig. 1 *P1* carries out I/O requests for the file domain of file stripe #1-6, while *P3* and *P5* are responsible for file stripe #7-12 and #13-18 respectively. To be specific, we assume that the buffer size is twice that of the stripe size, thus I/O requests in a file domain should be separated into three iteration steps to complete, as Fig. 2 shows. As we can see, due to the existing file domain partitioning method and aggregator assignment, every aggregator needs to carry out the I/O requests and communicate with more than one OST within each iteration, which would result in access contention problems. For instance, all aggregators compete to access the *OST1* and *OST2* in the first iteration, while *OST3* stays idle.

There exists a solution to address this problem by taking the data layout information into consideration when deciding the partition of file domains. As shown in Fig. 3, by exchanging the file domain data between aggregators, each aggregator takes responsibility for the data that resides in the same OST, making a "one-to-one" matching pattern between aggregators and OSTs which avoids the contention problem. However, this design may introduce inter-node communications during the shuffle phase. As shown in Fig. 3, aggregator#1 (*P1*) only needs to communicate with *P2* before the optimization, because its file domain only contains data from *P2* and itself (i.e. file stripes #1-6). However, after reorganizing the file domain, *P1* will have to communicate with processes #2-6 (which is inter-node communication), because its file domain holds their data requests (i.e. file stripes #1, 4, 7, 10, 13, 16). A similar situation happens for aggregator#2 (*P3*) and aggregator#3 (*P5*).

This trade-off between I/O cost and shuffle cost seems inevitable, because once we change the organization of the file domain data issued to the underlying file system in terms of data layout or data locality, we also influence the communication pattern between the aggregator and its corresponding processes based on the file domain when we distribute data during the shuffle phase. Unlike them, our proposed new IBCS solution is exactly designed to avoid such situations from happening.

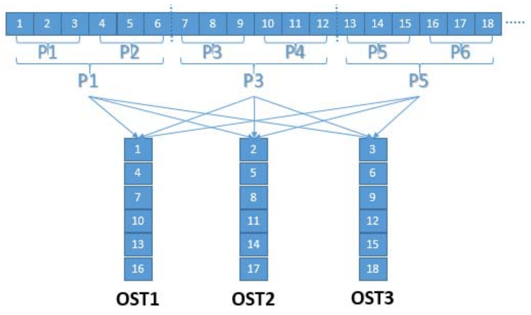

Figure 1. A representative example of a typical access pattern and the data layout on I/O servers in HPC applications.

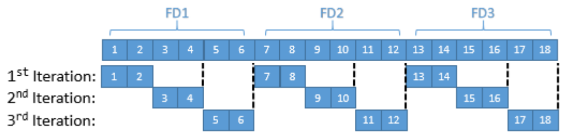

Figure 2. Each aggregator needs three iteration steps to complete I/O requests and the access contention problem occurs.

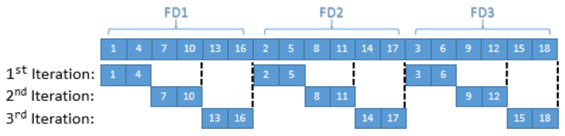

Figure 3. A common way to exchange file domain data in order to reduce access contentions. After the reorganization, each file domain contains the data that resides in one OST. However, it may introduce extra inter-node communication cost.

B. IBCS Design and Implementation

1) IBCS Design

As we have explained in the previous section, our strategy tries to reduce the contentions based on the fact that the iteration is a finer-grained parallelism during a collective

I/O operation. It means that we do not have to make sure that no contention arises when aggregators carry out the whole file domain data. Instead, we only need to make sure that no conflicts happen when carrying out I/O requests within an iteration. The basic idea of IBCS is to reorganize the I/O requests within a file domain so that each OST is only accessed by one aggregator process within every iteration step to avoid contentions.

We still use the previous example to explain our design. As shown in Fig. 1 and Fig. 2, we have assumed that the collective buffer size used here is twice that of a file stripe so that I/O requests of the aggregator's file domain data need three iteration steps to be completed and contention exists as described. Fig. 4 illustrates what the request order becomes after we reorganize I/O requests following our strategy within every file domain. It shows that IBCS rearranges I/O requests to make every aggregator access only one OST and vice versa within the same iteration step to avoid access contentions. Apart from that, our strategy also packs I/O requests whose data resides on the same OST together and then issues them to the I/O servers as a whole so that we gain the physical disk contiguity when accessing data in the OST. For example, after the rearrangement, file stripe #1 and #4 are accessed as a whole within the first iteration and their disk positions are contiguous physically on the *OST1*, which reduces the disk seek time.

Notice that in our strategy design, we do not change the file domain of each aggregator, but only reorganize the issue order of data within the file domain. This design does not complicate process communications during the shuffle phase as mentioned earlier. Once the relationship between aggregators and computing processes are decided by the file domain partitioning method, according to the application's access pattern, no more complexity is introduced for the shuffle phase because of our optimization for the I/O phase.

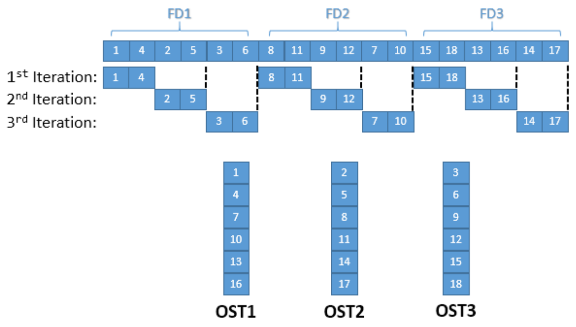

Figure 4. The request order after reorganizing data within the file domain by our strategy. After reorganizing, every aggregator accesses only one OST respectively within the same iteration step to avoid access contentions and has better physical contiguity to reduce the disk seek time.

2) *IBCS Implementation Methodology*

The implementation of the proposed IBCS design is not complicated. We decide the issuing order of requests with two rules: one is to make every OST serve at most one aggregator during one iteration to avoid contentions; the second is to make every aggregator issue I/O requests during every iteration to maximize concurrency, so that we will not increase the number of iteration steps that one file domain originally needs.

First, each aggregator collects data that resides in the same OST in its file domain and groups them together so that each group contains data that resides in the same OST. In order to gain a better arrival order for the disk, we also sort the data in the group by its offset in the file. As described, in every iteration step, we transfer data with the specified buffer size that resides in one OST. If the size of the group data cannot be divided by the buffer size exactly, we will get the remaining data. Apparently, the size of these remaining data is smaller than the buffer size. There are two methods to deal with this. If we simply choose to transmit them exclusively in an iteration step in order to avoid contentions even though it would not fulfill the whole buffer, we may end up with increasing the total number of iterations that may impact the I/O throughput. On the other hand, if we mix all the remaining data from different groups we may still cause contentions. Thus, we provide a parameter *Threshold*, so that we can dynamically decide which method to use.

If $SIZE_{remaining\ data} > Threshold * SIZE_{buffer}$, we continue to transmit them with an iteration step even though we know it would not fulfill the whole buffer. This way we may increase the number of iterations to keep from contentions, but the increment is well limited as with the well-tuned control condition here (usually no more than one extra iteration step), so that the tradeoff will not be bothered that much. On the other hand, if $SIZE_{remaining\ data} < Threshold * SIZE_{buffer}$, in our implementation, we use an "extended buffer" to handle this situation instead of mixing all the remaining data and transmitting them all in one single iteration. That is, we apply for a little more buffer memory so that we can make the extra data fit. Fig. 5 shows the extended buffer used here, the block colored darker is the extra data whose size does not fulfill the buffer, but we extend the buffer size a little more to hold it.

After calculating the buffer size for each group, we partition the group into several items that consist of data blocks with buffer-size so that each item can be transferred within an iteration step. For example, in Fig. 6, each file domain has been partitioned into three groups that contain a certain OST's data (i.e. the group1 of *FD1* contains data in *OST1* etc.). We assume that the buffer size for each iteration step is the same as the stripe size; so group1 contains 3 items, and group2&3 have 2 items, respectively. The item here is the smallest reorganizing unit in our algorithm, and we should note that the number of items in each group is also the number of iteration steps needed to carry out requests of that OST's data.

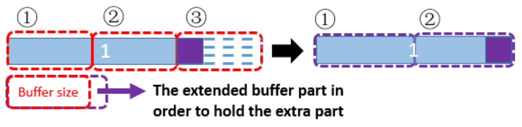

Figure 5. Extended buffer used to hold extra data.

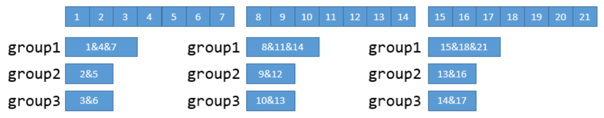

Figure 6. Each file domain is partitioned into groups. Each group contains data that resides in the same OST and sorted by the offset.

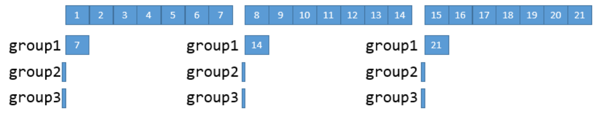

Figure 7. The situation of remaining items in each group after the first couple of iterations, the number of some groups' items is zero.

We first find out the minimum number of items for each group in the global file domain. In our example, it is two because each group has at least two items. Then we start with those first two iterations. In order to make each OST only serve one aggregator, a possible schedule order is the following: when aggregator#1 transfers two items from group2 with two iterations, then aggregator#2 transfers two items from group3 and aggregator#3 transfers two items from group1. Since every group has at least two items, we can make sure that each OST's service time is saturated and no OST is serving more than two aggregators at the same time. Then, after 6 (2*3) iteration steps, the number of items in some of the groups is zero, because they have finished their data transferring during the previous iteration steps. In our example, the situation of remaining items in each group is shown in Fig. 7. Then we deal with the remaining items in a similar way. We still choose to schedule the group that has the least number of items in it first (greater than zero). The reason why we continue with those groups is that we can always find items in other file domain's groups, and we will not make the aggregator idle during a certain iteration step. At the same time, the sooner we finish one group's item transfer, the less OST this aggregator would occupy, which provides more flexibility for other aggregators' item schedule.

C. *Strategy Analysis*

This algorithm can find a proper issue order of I/O requests that avoids contentions without causing extra iteration steps for those access patterns that have the following two characteristics: 1) each file domain has the same *ntimes* value, i.e. the same number of iteration steps that it needs in order to complete all of its I/O requests; and 2) the amount of data in each OST is the same. If we make an M*N size table (N is the number of aggregators and M is the number of OSTs) with the number of items in each file domain that belong to each OST, then as long as we add up the values in each row, we can get the same number for each column and vice versa. Take the access pattern in Fig.6 for example, when we make such a table as in Table I, we can see that each file domain's *ntimes* value is 7 (i.e. each aggregator needs 7 iteration times to complete I/O requests in its file domain) and the total amount of data for each OST is also 7 (i.e. each OST has 7 buffer-size data required by this collective I/O operation). Actually many scientific applications with non-contiguous I/O pattern have similar access patterns, if not exactly the same. This is due to the fact that when applying collective I/O optimization, the non-contiguous access is gathered into a contiguous data block which is usually striped across all the I/O servers evenly. At the same time, the contiguous data block is partitioned into balanced file domains between aggregators by default [5] which our design does not change. These two factors guarantee that our strategy is generally applicable by many scientific application I/O patterns such as big array access.

TABLE I. ACCESS PATTERN DESCRIPTION

Number Of items	Aggregator 1	Aggregator 2	Aggregator 3
OST1	3	2	2
OST2	2	3	2
OST3	2	2	3

The reason why our design is effective is three-fold. First, by reorganizing the requests within a file domain, we make sure that each OST simultaneously serves only one aggregator process, i.e. we adopt a "one-to-one" matching pattern between aggregators and I/O servers to avoid contentions. Second, in our implementation, the item we schedule consists of requesting data that is already sorted by the offset in an ascending order, so when we issue those requests as a whole to the file system within an iteration process, we can increase the amount of data read from or write to the disk with each disk head seek so as to amortize the seek overhead. Third, without interfering with the file domain data organization, we will not increase the number of processes each aggregator needs to communicate with during the shuffle phase, i.e. we will not introduce extra shuffle cost.

IV. Experimental Results and Analyses

In this section, we show the results and analyses of the experiments by using our proposed iteration based collective I/O strategy compared to ROMIO's existing one.

A. *Experimental Setup*

Our experiments are conducted on a cluster with 64 computing nodes and 12 file I/O servers. Each computing node is equipped with 16 Intel Xeon E5-2670 CPU processors and 64GB RAM. Each I/O server installs Lustre-2.1.3 file system and the file stripe size is set to 1MB. All the nodes are connected via a Gigabit Ethernet. The experiments are tested with MPICH3.0.4 release, and we implement our strategy in its ADIO driver for Lustre. The operating system is Red Hat Enterprise Linux 6.2 with kernel 2.6.32.

B. *IOR Benchmark*

The benchmark we use is IOR-2.10.3 which is a parallel file system benchmark developed at Lawrence Livermore National Laboratory [21]. It provides different access types to run the benchmark such as MPI-IO, POSIX, and HDF5 [22]. It can mimic many I/O access patterns of real applications by choosing different parameters.

We conduct three groups of experiments. First, we compare I/O performance of our IBCS with ROMIO's existing collective I/O strategy. Our test performs both write and read operations to a shared file, configured with 6 processes performing collective I/O operations acting as aggregators. Each process transfers 576MB data, each I/O call writes or reads 48MB data, and the collective I/O buffer size is set to 4MB. We run IOR several times to calculate the average value of I/O bandwidth. We can see from Fig. 8 that the I/O throughput using IBCS outperforms ROMIO's existing strategy for both write and read operations. Compared to the collective I/O strategy ROMIO currently uses, IBCS achieves more than 63% higher throughput for write operation and 47% for read operation.

We then measure and compare the write performance of both strategies with respect of different buffer sizes and different numbers of I/O servers. Shown in Fig. 9, we change the collective buffer size to 512KB, 1MB, 4MB, and 8MB. We can see from Fig. 9 that, when collective I/O buffer size is small, the I/O performance becomes worse. This is because it needs more iteration steps to carry out the I/O requests within a collective I/O operation. However, the I/O bandwidth of the IBCS still outperforms that of ROMIO strategy even when the buffer size is small, because our design can still adopt our strategy to reorganize data under such memory-limited situations. We can also see that, as the buffer size increases, the improvement of IBCS is still faster than ROMIO strategy, which is mainly because IBCS can reduce disk seek time due to the better data locality it provides.

We also evaluate the impact of the number of I/O servers and present the results in Fig. 10. In these tests, we set the number of aggregators the same as the number of the I/O servers. The number of I/O servers is varied as 4, 6, 8, and 12. We observe that our strategy achieves better scalability compared to ROMIO's strategy with the increasing number of I/O servers. As the number of I/O servers grows, the access contentions and the communication cost between aggregators and processes both increase. The advantage of our strategy here is due to the fact that it can reduce the cost of both the I/O phase and the shuffle phase.

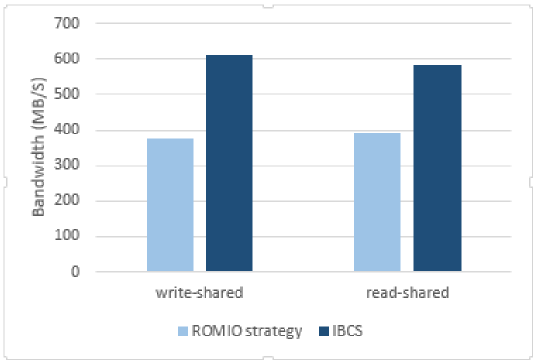

Figure 8. Performance of IBCS and ROMIO's existing collective I/O strategy. We run IOR several times then calculate the average value of I/O bandwidth.

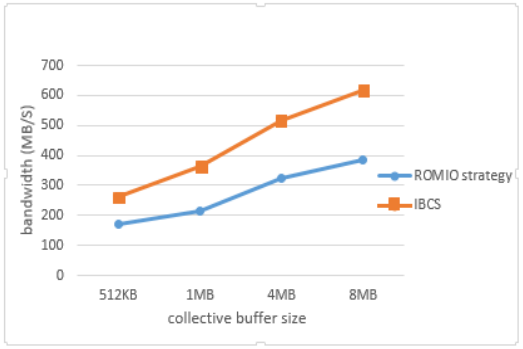

Figure 9. Performance of IBCS and ROMIO's existing collective I/O strategy with different collective I/O buffer sizes.

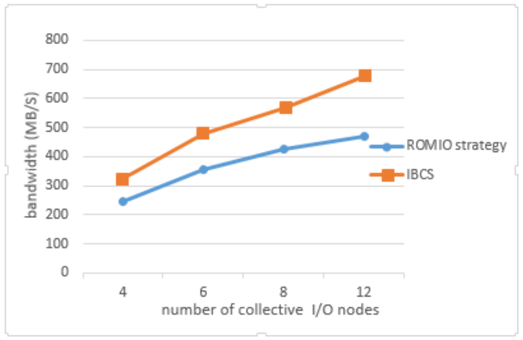

Figure 10. Performance of IBCS and ROMIO's existing collective I/O strategy with different number of I/O servers.

All these experiment results have well confirmed that the proposed IBCS in this study is a promising optimization method designed to explore data locality and access parallelism. It can help data-intensive scientific applications provide high performance parallel I/O in order to meet the growing demands of HPC applications.

C. *BTIO Benchmark*

Apart from the synthetic benchmark above, our evaluation also uses the BTIO benchmark [23] from the *NAS Parallel Benchmark* (NPB3.3.1) suite to represent a typical scientific application with interleaved intensive computation and I/O phases. BTIO uses a *Block-Tridiagonal* (BT) partitioning pattern on a three-dimensional array across a square number of compute processes. Each process is responsible for multiple Cartesian subsets of the entire data set and performs large writes and reads of a nested strided data to a shared file.

We consider the Class C and full subtype BTIO workload in our experiment. That is, we write and read a total size of 6.64GB data with collective buffering. We use 9, 16, 25, and 36 compute processes since BTIO requires a square number of processes. Collective buffer size is set to 4MB, output file is striped across 4 OSTs on the Luster file system, and 4 processes are picked as collective I/O aggregators.

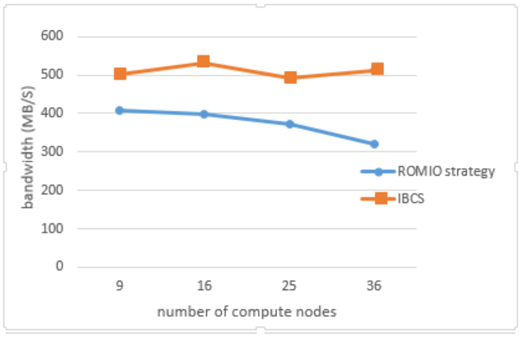

Figure 11. I/O bandwidth results for BTIO benchmark.

As shown in the Fig. 11, compared to ROMIO strategy, our IBCS design achieves better throughput and scalability. If the number of compute processes increases, the percentage improvement of our design in bandwidth also tends to increase. This is mainly because with the increase in compute processes, an aggregator needs to communicate with more computer processes, such that data shuffle cost between these processes and corresponding aggregators increase. However, as discussed, our strategy can help to reduce the shuffle phase cost as well as the I/O phase cost in the two-phase collective I/O implementation. The experiment result confirms that IBCS design helps to improve the I/O performance significantly.

V. Related Work

There has been significant amount of research studies about techniques to improve the parallel I/O performance of data-intensive scientific applications [19, 24-27]. A lot of researches aim to design and evaluate efficient and scalable I/O strategies to support I/O middleware concepts such as the file data description and collective I/O.

Chen et al. [25] propose a new collective I/O strategy called *Layout Aware Collective I/O* (LACIO), which makes the optimization decision based on the physical data layout information. The proposed strategy rearranges the partitions of file domains and the requests of aggregators in a fashion that match the physical data layout on storage servers of a system in order to reduce access contentions and exploit better concurrency and locality.

Dickens et al. [19] report poor MPI-IO performance on Lustre when the aggregators perform large, contiguous I/O operations in parallel. They design a user level library named Y-Lib which redistributes data in a way that conforms much more closely to the storage layout to increase the performance of MPI-IO. The basic idea of Y-Lib is to minimize the number of OSTs with which a given aggregator communicates and shows good results.

In order to improve the disk access time, Zhang et al. [26] design a new collective I/O implementation, named resonant I/O, which rearrange I/O requests by the presumed on the disk data layout. The design makes sure that those data from the same file to arrive at each I/O node in ascending order of file offsets to allow the disk to serve the requests in its preferred order to achieve high disk throughput.

Liao et al. [27] propose file partition methods that allow an aggregator's file domain to be aligned to the lock boundary of the underlying file system in order to avoid the lock contentions. This is done by aligning every two-neighbor file domain to the nearest stripe boundary so that no file stripe will be accessed by more than two aggregators. Although this method may cause unbalanced I/O load among aggregators, this side effect will not interfere much as I/O requests are large enough. Actually, an ADIO driver for Lustre has been added to the recent release of ROMIO which adopts the similar approach and proves good performance result. In order to limit the number of aggregators an I/O server may communicate with, the static-cyclic and group-cyclic partitioning methods fix the association of file stripes to aggregators.

These studies mostly focus on the optimization of the I/O phase, and by changing the organizing way of file domains, they may increase latent communication complexity of processes during the shuffle phase, such as more inter-node process communications. However, based on the iteration process, our design is intended to address the contention problems by taking both I/O phase and shuffle phase into consideration.

VI. Conclusion and Future Work

Collective I/O has been proven a critical technique in optimizing many scientific applications running on HPC systems. However, our observation also confirms that when running collective I/O applications on the parallel I/O systems, there exists a significant I/O performance problem due to access contention. Based on the fact that an I/O operation would be separated into several iteration steps to complete, we propose, implement, and evaluate an iteration based collective I/O strategy to address this problem.

In the near future, we plan to further explore a cooperative file domain partitioning method to cooperate with our design in order to make it fit more access patterns.

Acknowledgment

Experiments presented in this paper were carried out using the High Performance Computing Center experimental testbed in Services Computing Technology and System Lab/ Cluster and Grid Computing Lab. This work is supported by the NSFC under grants No. 61370104 and No. 61133008, National Science and Technology Pillar Program under grant 2012BAH14F02, MOE-Intel Special Research Fund of Information Technology under grant MOE-INTEL-2012-01, National 863 Hi-Tech Research and Development Program under grant 2012AA01A306, and the National Science Foundation under grant CNS-1162488.

References

[1] T. Hey, K. Tolle, and S. Tansley, "Jim Gray on eScience: a transformed scientific method", The Fourth Paradigm: Data-Intensive Scientific Discovery. 2009.

[2] Y. Cui, K. B. Olsen, T. H. Jordan, K. Lee, J. Zhou, P. Small, D. Roten et al., "Scalable earthquake simulation on petascale supercomputers",

in Proceedings of 2010 International Conference for High Performance Computing, Networking, Storage and Analysis (SC'10), pp. 1-20. IEEE, 2010.

[3] J. H. Chen, A. Choudhary, B. De Supinski, M. DeVries, E. R. Hawkes, S. Klasky, W. K. Liao et al., "Terascale direct numerical simulations of turbulent combustion using S3D", Computational Science & Discovery, vol. 2, no. 1, pp. 015001. IOP Publishing, 2009.

[4] T. Yuan, S. Klasky, H. Abbasi, J. Lofstead, R. Grout, N. Podhorszki, Q. Liu, Y. Wang, and W. Yu, "EDO: improving read performance for scientific applications through elastic data organization", in Proceedings of 2011 IEEE International Conference on Cluster Computing (CLUSTER'11), pp. 93-102. IEEE, 2011.

[5] R. Thakur, W. Gropp, and E. Lusk, "Data sieving and collective I/O in ROMIO", in Proceedings of The 7th Symposium on the Frontiers of Massively Parallel Computation (Frontiers'99), pp. 182-189. IEEE, 1999.

[6] W. Gropp, E. Lusk, N. Doss, and A. Skjellum, "Using MPI: portable parallel programming with the message-passing interface", vol. 1. MIT Press, 1999.

[7] P. J. Braam, "The lustre storage architecture". 2004.

[8] F. B. Schmuck, and R. L. Haskin, "GPFS: a shared-disk file system for large computing clusters", in FAST, vol. 2, no. 19. 2002.

[9] R. B. Ross, and R. Thakur, "PVFS: a parallel file system for linux clusters", in Proceedings of the 4th Annual Linux Showcase and Conference, pp. 391-430. 2000.

[10] H. Jin, T. Ke, Y. Chen, and X.H. Sun, "Checkpointing orchestration: toward a scalable hpc fault-tolerant environment", in Proceedings of 2012 IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid'12), pp. 276-283. IEEE, 2012.

[11] J. Gray, D. T. Liu, M. Nieto-Santisteban, A. Szalay, D. J. DeWitt, and G. Heber, "Scientific data management in the coming decade", in ACM SIGMOD Record, vol. 34, no. 4, pp. 34-41. ACM, 2005.

[12] S. Philip, "Lustre: Building a file system for 1000-node clusters", in Proceedings of 2003 Linux Symposium, vol. 2003. 2003.

[13] W. K. Liao, "Design and evaluation of MPI file domain partitioning methods under extent-based file locking protocol", IEEE Transactions on Parallel and Distributed Systems (TPDS), vol. 22, no. 2, pp. 260-272. IEEE, 2011.

[14] N. Arifa, W. K. Liao, and A. Choudhary, "Delegation-based I/O mechanism for high performance computing systems", IEEE Transactions on Parallel and Distributed Systems (TPDS), vol. 23, no. 2, pp. 271-279. IEEE, 2012.

[15] H. Song, Y. Yin, Y. Chen, and X. H. Sun, "Cost-intelligent application-specific data layout optimization for parallel file systems", Cluster computing, vol. 16, no. 2, pp. 285-298. Springer, 2013.

[16] B. Dong, X. Li, L. Xiao, and L. Ruan, "A new file-specific stripe size selection method for highly concurrent data access", in proceedings of 2012 ACM/IEEE International Conference on Grid Computing (GRID'12), pp. 22-30. IEEE, 2012.

[17] J. Liu, Y. Chen, and Y. Zhuang, "Hierarchical I/O scheduling for collective I/O", in Proceedings of 2013 IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid'13), pp. 211-218. IEEE, 2013.

[18] K. Cha, and S. Maeng, "Reducing communication costs in collective I/O in multi-core cluster systems with non-exclusive scheduling", The Journal of Supercomputing, vol. 61, no. 3, pp. 966-996. Springer, 2012.

[19] P. M. Dickens, and J. Logan, "Y-lib: a user level library to increase the performance of MPI-IO in a lustre file system environment", in Proceedings of 2009 ACM International Symposium on High Performance Distributed Computing (HPDC'09), pp. 32-38. ACM, 2009.

[20] X. Zhang and S. Jiang, "InterferenceRemoval: removing interference of disk access for MPI programs through data replication", in Proceedings of 2010 ACM International Conference on Supercomputing (ICS'10), pp. 223-232. ACM, 2010.

[21] H. Shan, and J. Shalf, "Using IOR to analyze the I/O performance for HPC platforms". 2007.

[22] M. Folk, G. Heber, Q. Koziol, E. Pourmal, and D. Robinson, "An overview of the HDF5 technology suite and its applications", in Proceedings of 2011 EDBT/ICDT Workshop on Array Databases, pp. 36-47. ACM, 2011.

[23] P. Wong and R. Der. Wijngaart, "NAS parallel benchmarks I/O version 2.4", Technical Report NAS-03-002, NASA Ames Research Center, Moffet Field, CA. 2003.

[24] K. Gao, W. K. Liao, A. Nisar, A. Choudhary, R. Ross, and R. Latham, "Using subfiling to improve programming flexibility and performance of parallel shared-file I/O", in Proceedings of 2009 International Conference on Parallel Processing (ICPP'09), pp. 470-477. IEEE, 2009.

[25] Y. Chen, X. H. Sun, R. Thakur, P. C. Roth, and W. D. Gropp, "LACIO: a new collective I/O strategy for parallel I/O systems", in Proceedings of 2011 IEEE International Parallel & Distributed Processing Symposium (IPDPS'11), pp. 794-804. IEEE, 2011.

[26] X. Zhang, S. Jiang, and K. Davis, "Making resonance a common case: a high-performance implementation of collective I/O on parallel file systems", in Proceedings of 2009 IEEE International Parallel & Distributed Processing Symposium (IPDPS'09), pp. 1-12. IEEE, 2009.

[27] W. K. Liao, and A. Choudhary, "Dynamically adapting file domain partitioning methods for collective I/O based on underlying parallel file system locking protocols", in Proceedings of 2008 International Conference for High Performance Computing, Networking, Storage and Analysis (SC'08), pp. 1-12. IEEE, 2008.