Contents lists available at ScienceDirect
Big Data Research
www.elsevier.com/locate/bdr

Hierarchical Collective I/O Scheduling for High-Performance Computing ☆

Jialin Liu, Yu Zhuang, Yong Chen

Department of Computer Science, Texas Tech University, Lubbock, TX, USA

A R T I C L E I N F O

Article history:
Received 7 December 2014
Accepted 15 January 2015
Available online xxxx

Keywords:
Collective I/O
Scheduling
High-performance computing
Big data
Data intensive computing

A B S T R A C T

The non-contiguous access pattern of many scientific applications results in a large number of I/O requests, which can seriously limit the data-access performance. Collective I/O has been widely used to address this issue. However, the performance of collective I/O could be dramatically degraded in today's high-performance computing systems due to the increasing shuffle cost caused by highly concurrent data accesses. This situation tends to be even worse as many applications become more and more data intensive. Previous research has primarily focused on optimizing I/O access cost in collective I/O but largely ignored the shuffle cost involved. Previous works assume that the lowest average response time leads to the best QoS and performance, while that is not always true for collective requests when considering the additional shuffle cost. In this study, we propose a new hierarchical I/O scheduling (HIO) algorithm to address the increasing shuffle cost in collective I/O. The fundamental idea is to schedule applications' I/O requests based on a shuffle cost analysis to achieve the optimal overall performance, instead of achieving optimal I/O accesses only. The algorithm is currently evaluated with the MPICH3 and PVFS2. Both theoretical analysis and experimental tests show that the proposed hierarchical I/O scheduling has a potential in addressing the degraded performance issue of collective I/O with highly concurrent accesses.

1. Introduction

The volume of data collected from instruments and simulations for scientific discovery and innovations keeps increasing rapidly. For example, the Global Cloud Resolving Model (GCRM) project [1], part of DOE's Scientific Discovery through Advanced Computing (SciDAC) program, is built on a geodesic grid that consists of more than 100 million hexagonal columns with 128 levels per column. These 128 levels will cover a layer of 50 kilometers of atmosphere up from the surface of the earth. For each of these grid cells, scientists need to store, analyze, and predict parameters like the wind speed, temperature, pressure, etc. Most of these global atmospheric models process data in a 100-kilometer scale (the distance on the ground); however, scientists desire higher resolution and finer granularity, which can lead to significant larger sizes of datasets. Table 1 shows the data requirements of representative scientific applications run at Argonne Leadership Computing Facility (ALCF) through the DOE's INCITE program [34]. The data volume processed online by many applications has exceeded TBs or even tens of TBs; the off-line data is near PBs of scale.

Table 1
Data requirements of representative INCITE applications at ALCF [34].

Project	On-line data	Off-line data
FLASH: Buoyancy-Driven Turbulent Nuclear Burning	75 TB	300 TB
Reactor Core Hydrodynamics	2 TB	5 TB
Computational Nuclear Structure	4 TB	40 TB
Computational Protein Structure	1 TB	2 TB
Performance Evaluation and Analysis	1 TB	1 TB
Climate Science	10 TB	345 TB
Parkinson's Disease	2.5 TB	50 TB
Plasma Microturbulence	2 TB	10 TB
Lattice QCD	1 TB	44 TB
Thermal Striping in Sodium Cooled Reactors	4 TB	8 TB

During the retrieval and analysis of the large volume of datasets on high-performance computing (HPC) systems, scientific applications generate huge amounts of non-contiguous requests [27,38], e.g., accessing the 2-D planes in a 4-D climate dataset. Those non-contiguous requests can be considerably optimized by performing a two-phase collective I/O [11]. However, the performance of the

☆ This article belongs to BDA-HPC.

E-mail addresses: jaln.liu@ttu.edu (J. Liu), yu.zhuang@ttu.edu (Y. Zhuang), yong.chen@ttu.edu (Y. Chen).

http://dx.doi.org/10.1016/j.bdr.2015.01.007
2214-5796/

collective I/O could be dramatically degraded when solving big data problems on a highly-concurrent HPC system [12,30]. A critical reason is that the increasing shuffle cost of collective requests can dominate the performance. This increasing shuffle cost is due to the high concurrency caused by intensive data movement and concurrent applications in today's HPC system. The shuffle phase is the second phase of a two-phase collective I/O. A collective I/O will not finish until the shuffle phase is done. Previous research has primarily focused on the optimization of the other phase, the I/O phase, of a collective I/O for data-intensive applications. In this study, instead of only considering the service time during the I/O phase, we argue that a better scheduling algorithm in collective I/O should also consider the requests' shuffle costs on compute nodes. An aggregator who has the longest shuffle time can dominate an application's overall performance, due to the reason that the slowest aggregator actually determines the overall performance of a collective I/O. In this research, we propose a new *hierarchical I/O (HIO)* scheduling to address this issue. The basic idea is, by saturating the aggregators' 'acceptable delay', the algorithm schedules each application's slowest aggregator earlier. The proposed algorithm is named as *hierarchical* I/O scheduling, because the predicted shuffle cost is considered at the MPI-IO layer on compute nodes and the server-side file system layer. Both layers leverage the shuffle cost analysis to perform an improved scheduling for collective I/O. The current analyses and experimental tests have confirmed the improvements over existing approaches. The proposed hierarchical I/O scheduling has a potential in addressing the degraded performance issue of collective I/O with highly concurrent accesses.

The contribution of this research is three-fold. First, we propose an idea of scheduling collective I/O requests with considering the shuffle cost. Second, we have derived functions to calculate and predict the shuffle cost. Third, we have carried out theoretical analyses and experimental tests to verify the efficiency of the proposed hierarchical I/O (HIO) scheduling. The results have confirmed that the HIO approach is promising in improving data accesses for high-performance computing. This work is an extension of our previous work [25]. The major difference is we generalize the HIO idea to a broader view, in which not only collective read but also write operation scheduling is designed, analyzed, and evaluated. The second difference is we add more evaluation results to demonstrate the potential of HIO. The third improvement is the Time Window concept. In the previous work, we only discussed how to use HIO to perform the scheduling on the queuing I/O requests, but we did not consider the starvation and interruption of aggregators within the same application, in other words, the aggregators from same application's different instance could also mess up with each other. We address the problem in this paper by applying a flexible time window concept. Besides, the shuffle cost prediction and HIO implementation are also extended a lot.

The rest of this paper is organized as follows. Section 2 reviews collective I/O and motivates this study by analyzing a typical example of interrupted collective read. Section 3 introduces the HIO scheduling algorithm. Section 4 presents the theoretical analysis of the HIO scheduling. Section 5 discusses the implementation. The experimental results are discussed in Section 6. Section 7 discusses related work and compares them with this study. Section 8 summarizes this study and discusses future work.

2. Background

2.1. Collective I/O

MPI is the dominant parallel programming model on all large-scale parallel machines, such as Cray XT5/XK6/XK7, IBM Blue Gene/P, IBM Blue Gene/Q supercomputers. We briefly review its I/O interface, MPI-IO, in this subsection. We will also discuss the MPI-IO's common implementation, the most important optimization, collective I/O, and its nonblocking version.

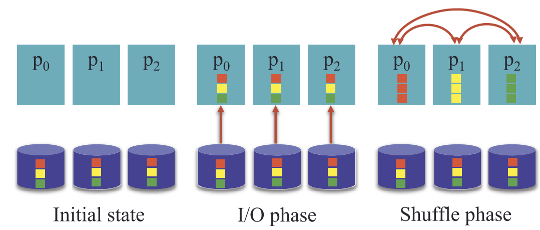

Fig. 1. Two phase collective I/O.

MPI-IO is a subset of the MPI-2/MPI-3 specification [13]. It defines an I/O access interface for parallel I/O. The primary motivation for MPI-IO specification came from the observation that parallel I/O optimizations require two basic abstractions: the ability to define a set of processes, i.e., MPI communicators, and the ability to define complex data access patterns, i.e., MPI data types. By equipping the two abilities, the MPI-IO is designed as an interface that supports many parallel I/O operations and optimizations. The implementation of MPI-IO is usually a middleware connecting parallel applications and underlying various parallel file systems, providing the code-level portability across many different machine architectures and operating systems. ROMIO is a popular MPI-IO implementation [37]. It provides an abstract-device interface called ADIO for implementing the portable parallel I/O API. It performs various optimizations, including collective I/O and data sieving, for common access pattern of parallel applications.

Collective I/O is one of the most important I/O access optimizations. In collective I/O, multiple processes cooperate with each other to carry out large aggregated I/O requests, instead of performing many non-contiguous and small I/Os independently. The motivation of collective I/O is several-fold. First, collective I/O can filter overlapping and redundant requests from multiple processes. Second, for many parallel applications, even though each process may access several noncontiguous portions of a file, the requests of multiple processes are often interleaved and may instead result in the access of one large contiguous portion of a file. Third, the collective I/O can reduce the number of system calls by combining small and noncontiguous requests into large and contiguous ones.

A widely-used implementation of collective I/O is the two-phase I/O protocol [37]. This strategy serves the I/O requests using an I/O phase and a data exchange phase. As shown in Fig. 1, in the case of two phase collective read, the first phase consists of a certain number of processes that are assigned as aggregators to access large contiguous data. In the second phase, those aggregators shuffle the data among all processes to the desired destination.

Collective I/O is a technique to optimize one application's I/O, such optimization does not consider the interruption from other application, or other processes. While in current and future extreme scale HPC system, the highly concurrency is not neglectable. As the interruption increasing, the service order of the aggregator is random and one application's aggregators can have different waiting time (they are supposed to be served at the same time). To improve the average execution time and improve the performance, there should be an optimized scheduling methods.

Our scheduling idea for addressing the interruption issue originates from the nature of the two phase collective I/O itself. By scheduling the aggregators from multiple concurrent applications, we achieved lower average execution time.

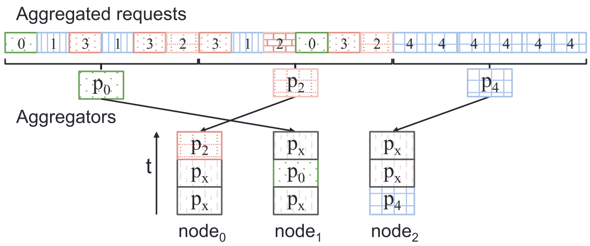

Fig. 2. Interrupted collective read. (p_x is from other applications.)

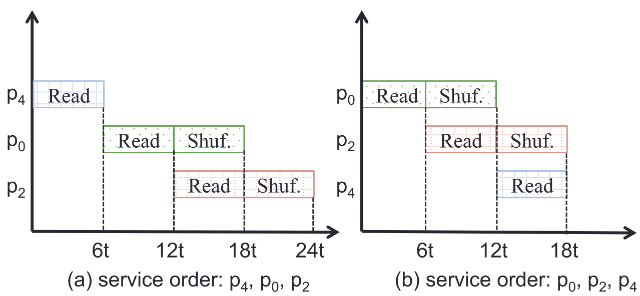

Fig. 3. Two different service orders.

3. Motivation

Collective I/O plays a critical role in cooperating processes to generate aggregated I/O requests, instead of performing non-contiguous small I/Os independently [9,37]. As we discussed in the background section, a widely-used implementation of collective I/O is the two-phase I/O protocol [37]. For collective reads, in the first phase, a certain number of processes are assigned as aggregators to access large contiguous data; in the second phase, those aggregators shuffle the data among all processes to the desired destination. There is no synchronization during the shuffle phase, which means, as long as one aggregator gets its data, it will redistribute the data among processes immediately without waiting for other aggregators.

An observation is that, on today's HPC system with highly concurrent accesses, the service order of the aggregators on storage nodes can have an impact on the application's overall performance. The example in Fig. 2 shows a two-phase collective read operation, which is interrupted by processes from other concurrent applications due to highly concurrent accesses. In Fig. 2, five processes (p_0–p_4) (on the same compute node for simplicity) from one MPI application are accessing the data striped across three storage nodes. During the first phase, the I/O aggregators, p_0, p_2 and p_4, are assigned with evenly partitioned file domains. In this case, we can predict that only two aggregators (i.e., p_0 and p_2) will have to redistribute the data among other processes (i.e., p_1 and p_3) in the shuffle phase. The reason why p_4 does not need to participate in the shuffle phase is that p_4's requests are only accessed by p_4 itself. From Fig. 2, we can also find that the service order of each aggregator is different on the storage nodes, which means three aggregators of the same application are not serviced at the same time. For example, assuming other processes have the same service time, then a possible service order for these three I/O aggregators is p_4, p_0 and p_2. Such a service order can have variants and can have an impact. We compare two of them in Fig. 3.

In Fig. 3, we analyze the cost for two different service orders. We assume that the service time and the shuffle cost is equal for the same amount of data movement and each process has service time $6t$, while the total cost is calculated as the sum of the read cost and the shuffle cost. In Fig. 3(a), the aggregator p_4 is serviced first. After $6t$, p_0 receives the service, and then p_2. During the shuffle phase, only p_0 and p_2 need to redistribute the data with other processes. Therefore, this application's total execution time is $3 \times 6t + 6t = 24t$. In Fig. 3(b), p_0 is serviced first. We find that the execution time is reduced to $18t$. The performance gain comes from scheduling the 'slowest aggregator' first. The 'slowest aggregator' in this study refers the aggregator who takes the longest time to redistribute the data in the shuffle phase. Another observation from Fig. 3(b) is that the service order of p_4 will not have impact on the total cost, which means even if p_4 comes first on node 2 in some case, we can still service it later. In other words, we can delay p_4 at most $12t$, this delay time is acceptable (no performance degradation will be caused). This example only shows that scheduling aggregators properly can improve the performance for one application, whereas for multiple concurrent applications, how to achieve the average lowest execution time is a challenge. Besides, the shuffle cost of different aggregators varies. How to predict the shuffle cost and pass it to the server is a challenge too. In the following sections, we introduce a hierarchical I/O scheduling (HIO) algorithm to address these issues. To the best of our knowledge, the hierarchical I/O scheduling is the first method that considers the increasing shuffle cost in collective I/O due to highly concurrency accesses.

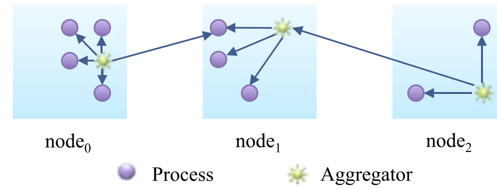

Fig. 4. Example of process assignment in three node multi-core system. The arrows show the inter- and intra-communication in the shuffle phase.

4. Hierarchical I/O scheduling

From the previous analysis, we can see that by scheduling slower aggregators earlier, the execution time (access cost and shuffle cost) can be reduced. In the case of with concurrent applications, however, the goal of the optimal scheduling should be achieving the lowest 'average' execution time. From the observation in Fig. 3(b), we know that application's aggregators may have 'acceptable delay' time. If such time is well utilized to service other applications, we can potentially achieve a win–win situation. In the following subsections, we first discuss how to predict the shuffle cost, and then formally introduce the concept of 'acceptable delay'. We also apply the time window concept [36] to divide the long I/O queue on each node into sub-sequences. Finally, we present the proposed hierarchical I/O scheduling algorithm.

4.1. Analysis and prediction of shuffle cost

In order to analyze the shuffle cost, we need to know how aggregators are assigned and how will they communicate with other desired processes. Most previous works assumed that only one aggregator is assigned in one node [4], which is also a default configuration in MPICH. The communication cost thus includes inter-node and intra-node cost, as shown in Fig. 4. The amount of data redistributed impacts the cost too. Therefore, the shuffle costs are mainly determined by exchanging data and the number and position of desired processes.

In Fig. 4, we illustrate an example of different communication patterns in the shuffle phase. In this example, there are totally three compute nodes, where each node is assigned with different number of processes, with only one process acting as the aggregator. During the shuffle phase, the aggregator either sends the data to processes on other nodes, which results in inter-node communication, or redistributes the data within the same node, which

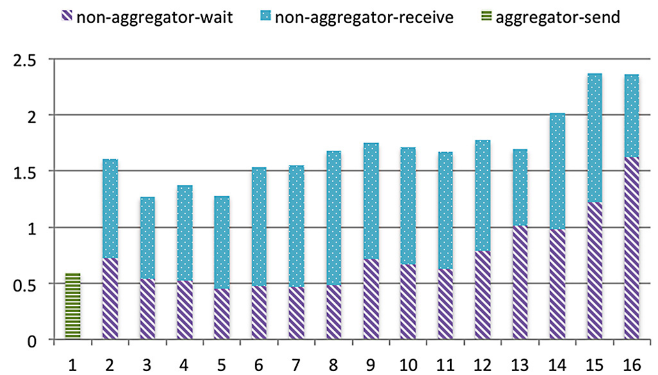

Fig. 5. Timing aggregator and non-aggregator's cost in the shuffle phase.

results in intra-node communication. As a consequence, each aggregator will have different shuffle cost.

To predict the shuffle cost of each aggregator, we then conducted an initial experiment by timing one aggregator and its corresponding processes, as shown in Fig. 5. This initial test is performed in two nodes, each node has 12 cores. We did a collective I/O using 16 processes, in which 12 processes including one aggregator are from the same node, while the other four processes are from the second node (this does not need to manually specify, just by simply launching 16 processes). The data amount that the aggregator will send to each non-aggregator is same. We modify the MPI-IO to timing the sending/receiving and other steps. The first bar in Fig. 5 shows the aggregator's sending time, which is the time this aggregator takes to send the data to the collective buffer. The lower dashed portion in other bars (2–16) shows the waiting time of non-aggregators and the higher portion refers to their receiving time. We can see that the data exchange time between aggregator and non-aggregators is various, due to which, we can argue that the shuffle cost is determined by the maximum data exchange time (in this case, i.e., 2.3 ms). Based on the above theoretical analysis and the initial experiment, we can derive the following equation to predict the shuffle cost:

$$T = max(max(\frac{m_{ai}}{B_a}), max(\frac{m_{ej}}{B_e})) + \gamma$$
$$= max(\frac{m_{ai}}{B_a}, \frac{m_{ej}}{B_e}) + \gamma \tag{1}$$

where T is the total shuffle cost of one aggregator; m_{ai} is the ith intra-message size (MByte) and $0 < i < A$, where A is the total number of intra-communication; m_{ej} is the jth inter-communication message size (MByte), $0 < j < E$, where E is the total number of inter-node communication; B_a is the saturated throughput of intra-node communication (MB/s); B_e is the saturated throughput of inter-node communication of a given cluster system (MB/s); and γ is the latency. The difference with ours is that their approach calculates the shuffle cost of each node, while ours calculates the cost in terms of each aggregator.

In order to distinguish the intra-communication and inter-communication at the runtime, we need to know the Hydra's process-core binding strategy. Hydra is a process management system compiled into the MPICH2 as a default process manager. Without any user-defined mapping, we assume the basic default allocation strategy is used, i.e., round-robin mechanism, using the OS specified processor IDs. Whether the communication will be intra or inter can be determined with the following equation:

$$Comm \text{ is } \begin{cases} intra & \text{if } a\_id\%n\_c = p\_id\%n\_c \\ inter & \text{else} \end{cases} \tag{2}$$

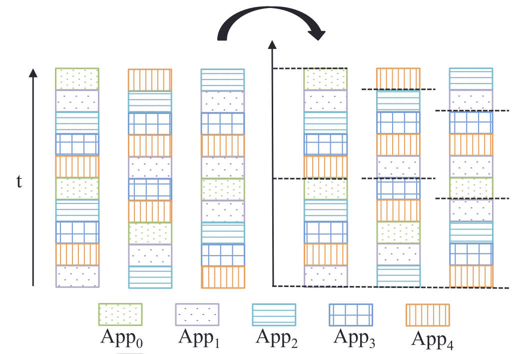

Fig. 6. Divided requests queues with time window.

where Comm is short for communication, a_id is the rank of the aggregator, p_id is the rank of non-aggregator processes, and n_c is the number of cores per node.

4.2. Acceptable delay

As we have analyzed in the previous section, aggregators with lower shuffle costs can be scheduled later, whereas slow aggregators who have higher shuffle costs are better to be serviced first. We introduce the *Acceptable Delay* (AD) in this study to support the hierarchical I/O scheduling. An aggregator's AD refers to the maximum acceptable time it can be delayed. The AD is defined as follows:

$$AD_i = max\{T_0, T_1, T_2, \ldots, T_n\} - T_i \tag{3}$$

where AD_i is ith aggregator's acceptable delay, and T_i is ith aggregator's shuffle cost. Usually, I/O requests from the same application are better to be serviced at the same time in order to achieve lower average response time. However, due to aggregators' various ADs, it is not necessary to do that, which means we can utilize every aggregator's AD to better schedule I/O requests, by saturating one aggregator's AD and servicing other applications first. We also define a Relative Acceptable Delay (RAD) as the rank in the ascending order of AD.

4.3. Time window

In previous work, a time window is used to avoid starvation and to maintain fairness [36]. With the time window concept, all I/O requests waiting in a queue are divided equally by a predefined time interval. Each divided window consists of a sequence of I/O requests. In the same window, I/O requests are ordered by the value of 'Application ID'; whereas in different windows, requests in an earlier 'Time Window' will be serviced prior to those in a later one, to avoid starvation.

As shown in Fig. 6, we utilize the time window to organize the I/O queue too but applied for the new hierarchical I/O scheduling algorithm. The motivation here is different compared to the existing work. In our design, we argue that the earlier request from one application should always be serviced earlier than the later one from the same application. The earlier I/O is from an aggregator of the earlier collective access. After this earlier collective access finishes, the application needs a synchronization before issuing another collective read/write. Scheduling the later aggregator earlier will cause delay. Therefore, instead of a fixed-width time window, we set a flexible time window on each server. On each storage node, the original queue will be divided into several windows. In each window, there are no more than two aggregators from the same application. In other words, the aggregators within a window are all from different applications.

Please cite this article in press as: J. Liu et al., Hierarchical Collective I/O Scheduling for High-Performance Computing, Big Data Research (2015), http://dx.doi.org/10.1016/j.bdr.2015.01.007

4.4. HIO algorithm

The main idea of the hierarchical I/O scheduling algorithm is to utilize the aggregator's 'acceptable delay' to minimize the shuffle cost. Because aggregators from different applications have various ADs, we cannot directly compare the AD from different applications. The algorithm first generates an initial order, and then tunes the order by comparing the aggregator's AD and read cost. If one aggregator's AD is larger than its successor's read cost, then the order of the two requests can be exchanged. The algorithm is described in Algorithm 1 – HIO scheduling algorithm. Before the HIO scheduling algorithm performed, the I/O queue is divided into fixed windows (e.g., 10) on each node. If there are more than one aggregator from the same application, the window width is reduced as discussed in the previous subsection. The outer loop (i to $n-1$) in the algorithm is carried out in parallel because the scheduling is performed on each node separately. The actual scheduling starts from the second loop (j to $m-2$). Each request's AD is compared with its successor's read cost. If $agg_j.ad > agg_{j+1}.read$, then exchange the order of agg_j and agg_{j+1}. At the same time, the AD is updated as: $agg_j.ad = agg_j.ad - agg_{j+1}.read$, $agg_{j+1}.ad = agg_{j+1}.ad + agg_j.read$.

```
input :
        n: number of storage nodes;
        m: number of applications;
        threshold: 0.2;
        agg[i][j].ad:the acceptable delay of jth
                aggregator on ith node;
        agg[i][j].read:the read cost of jth
                aggregator on ith node
        agg[i][j].rad:the relative ad of jth
                aggregator on ith node
output: Optimal service order on each node
for i ← 0 to n − 1 do
    ratio=sum(agg[i].shuffle)/sum(agg[i].read);
    if ratio > threshold then
        qsort agg[i] by rad;
    end
    else
        qsort agg[i] by app_id;
    end
    for j ← 0 to m − 2 do
        for k ← j + 1 to m − 1 do
            if agg[i][j].ad > agg[i][k].read then
                temp=ag[i][j];
                agg[i][j]=agg[i][k];
                agg[i][k]=temp;
                agg[i][j].ad+=agg[i][k].read;
                agg[i][k].ad-=agg[i][j].read;
            end
            else
                j++;
                break;
            end
        end
    end
end
```

Algorithm 1: HIO scheduling algorithm.

The initial order is generated by sorting the RAD (if *shuffle/read > threshold*), through which each application' slowest aggregator is scheduled earlier. The reason why a threshold is set that if the shuffle cost is too small compare to the read cost, the algorithm just sorts I/Os by application ID. The detailed reason is discussed in Section 4. The algorithm then tunes the initial order by saturating each aggregator's AD. These two steps make sure that the slower aggregator moves ahead, and the faster aggregator moves back. The tuning counts the read cost in order to balance the service order among all applications.

5. Theoretical analysis

The HIO scheduling can reduce the service time and shuffle cost. In this section, we analyze the cost reduction through an analytical model for collective I/O and compare against one latest Server I/O scheduling [36].

Assuming the number of concurrent applications is m, and the number of storage nodes is n. On each node, assuming every application has a request, then there will be m aggregators on each node. Suppose each request needs time t to finish the service on the server side and s_{ij} to finish the shuffle phase (s_{ij} is the jth aggregator's shuffle cost of the ith application). The longest finish time of requests on all nodes determines the application's completion time on the server side, which could be $\{t, 2t, 3t, \ldots, mt\}$. The application's total cost is the sum of the completion time on the server side and the maximum shuffle cost on the client side. Without any scheduling optimization, applications' server-side completion time has the same distribution, i.e., the density is $g(x)$, while the probability distribution function is $G(x) = (x/m)^n$. Therefore, the completion time of each application can be derided as shown in Eq. (4):

$$\begin{aligned} T_i &= Service\ time + Shuffle\ cost \\ &= E(max(Tc_i)) + max(s_{ij}) \\ &= (\sum_{x=1}^{m} xg(x))t + max(s_{ij}) \\ &= (\sum_{x=1}^{m} x(G(x) - G(x-1)))t + max(s_{ij}) \\ &= (\sum_{x=1}^{m} x((\frac{x}{m})^n - (\frac{x-1}{m})^n))t + max(s_{ij}) \\ &= mt + max(s_{ij}) - \frac{t}{m^n}\sum_{x=1}^{m-1} x^n \end{aligned} \tag{4}$$

in which Tc_i is the ith application's completion time on one node.

With the Server I/O scheduling, in which the same applications' requests are serviced at the same time on all nodes, the service time for those applications are fixed: $t, 2t, 3t, \ldots, mt$. The average execution time is:

$$\begin{aligned} T_i &= \frac{1}{m}(\sum_{x=1}^{m} xt + m(max(s_{ij}))) \\ &= \frac{m+1}{2}t + max(s_{ij}) \end{aligned} \tag{5}$$

For the potential of the HIO scheduling, we analyze the best case and the worst case separately. The best case requires two conditions: first, each application's slowest aggregator comes to different node; second, the slowest aggregator dominates the application's execution time, which can be described as $max(s_{ij}) - min(s_{ij}) > (m-1)t$. With the HIO scheduling, the slowest aggregator is serviced first on each node and determines each application's execution time. Therefore, we have the average execution time:

$$\begin{aligned} T_i &= \frac{1}{m}(mt + m(max(s_{ij}))) \\ &= t + max(s_{ij}) \end{aligned} \tag{6}$$

For the worst case, either the first condition or the second condition is not satisfied. If the first condition is not met, it indicates that applications' slowest aggregators arrive at the same node. Thus, the average execution time is:

$$T_i = \frac{1}{m}((t + 2t + 3t + \ldots + mt) + m(max(s_{ij})))$$
$$= \frac{m+1}{2}t + max(s_{ij}) \qquad (7)$$

If the second condition is not met, the slowest aggregator's shuffle cost is close to zero. With the HIO scheduling, the initial order will be sorted by application id, which means that the same application will be serviced at the same time. Then we have the average execution time same as Eq. (5). In another word, the worst case of the HIO scheduling at least has the same performance as that of the Server I/O scheduling.

Comparing Eq. (4) and Eq. (5), the Server I/O scheduling can achieve an average execution time reduction as the following:

$$T_{reduction} = mt + max(s_{ij}) - \frac{t}{m^n}\sum_{x=1}^{m-1} x^n$$
$$- \frac{m+1}{2}t - max(s_{ij})$$
$$= \frac{m-1}{2}t - \frac{t}{m^n}\sum_{x=1}^{m-1} x^n$$
$$= \frac{m-1}{2}t \ (n \to \infty) \qquad (8)$$

The best case of the HIO scheduling can further reduce the execution time of Eq. (5) by:

$$T^b_{reduction} = \frac{m+1}{2}t + max(s_{ij})$$
$$- t - max(s_{ij})$$
$$= \frac{m-1}{2}t \qquad (9)$$

The theoretical analysis and this comparison show that the HIO scheduling achieves better scheduling performance, especially when the shuffle cost keeps increasing due to highly concurrent accesses from large-scale HPC systems and/or big data retrieval and analysis problems.

6. Discussion of HIO for write operation

The HIO was originally designed for the collective read operation. Since the read operation's two phase procedure is different with the write operation. Our scheduling only works for the read operation, which is I/O phase first and shuffle phase second. The reason is that the I/O phase of read on server side is not the last step for the application, and the later shuffle phase on client side, if well utilized by the HIO, the overall performance can be improved. However, for collective write operation, the data are first shuffled on the client side, then after each aggregator getting its file view and all the data within that view, the aggregator will be sent to the storage nodes to do the I/O. This procedure is totally opposite of the read operation. The HIO idea seems to fail in this case. Fortunately, by rethinking the HIO idea, we found that it is not difficult to apply the HIO for write operations without modification. Same with collective read, the aggregator's shuffle phase for collective write also has various costs. Such various costs will lead to different arriving order on the storage server.

Suppose we have a 'third' phase for collective write, which means when the aggregators are returned to the compute nodes, they will also do the "shuffle" similar to the collective read's second phase. But assuming the cost of the third phase equals to zero, then it is not difficult to find that the collective write is just one 'worst' case of collective read for HIO. In Section 5, we have discussed how our HIO addresses the worst cases, in which the shuffle cost is close to zero and we have Eq. (5).

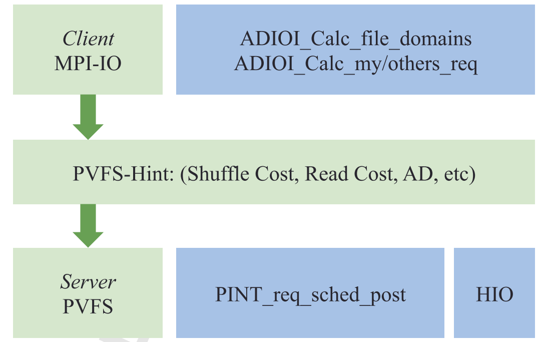

Fig. 7. Implementation.

7. Implementation

The aggregators' shuffle cost and AD are calculated at the MPI-IO layer. Our evaluation was carried out on the ROMIO that is included in MPICH2-1.4. It provides a high-performance and portable implementation of MPI-IO including collective I/O. The MPICH2-1.4 and ROMIO provide a PVFS2 ADIO device. We modified this driver to integrate the shuffle cost analysis and pass it to the PVFS server side scheduler as a hint. When an application calls the collective read function ADIOI_Read_and_exch in ad_read_coll.c under the src/mpi/romio/adio/common, the shuffle cost is calculated after the aggregators are allocated, i.e., ADIOI_Calc_file_domains. The message size m is calculated with ADIOI_Calc_my_req and ADIOI_Calc_others_req. The calculated shuffle cost is stored into a variable of PVFS-hint type. The hint is passed to file servers along with I/O requests (Fig. 7).

On the PVFS server side, in the request scheduling function PINT_req_ sched_post(), we implemented the HIO algorithm. The original function only enqueues the coming requests into the tail of the queue, while the HIO algorithm first divides the waiting queue into several sequences, and performs the scheduling within each sub-queue following the scheduling algorithm discussed in Section 3.

8. Experiments and analyses

8.1. Experimental setup

We have conducted tests on a 16-node Linux testbed. This cluster is composed of one PowerEdge R515 rack server node and 15 PowerEdge R415 nodes, with a total of 32 processors and 128 cores. Nodes are fully connected via a PowerConnect 2848 network switch. The PowerEdge R515 server node has dual quad-core 2.6 GHz AMD Opteron 4130 processors, 8 GB memory, and a RAID-5 disk array with 3 TB storage capacity composed of 7200 RPM Near-Line SAS drives. Each PowerEdge R415 node has dual quad-core 2.6 GHz AMD Opteron 4130 processors, 4 GB memory and a 500 GB 7200 RPM Near-Line SAS hard drive. We conducted experiments with the MPI-IO-Test parallel I/O benchmark [2]. The proposed hierarchical I/O scheduling algorithm was compared with other scheduling strategies through tests. We have also evaluated the HIO scheduling algorithm with a real climate science application. The HIO scheduling algorithm is evaluated and compared with two other scheduling algorithms, Server I/O scheduling (denoted as SIO) [36] and the normal collective I/O (denoted as NIO).

8.2. Results and analyses

In the first test, we run multiple instances of MPI-IO-Test simultaneously. We conducted the experiments by specifying the number of aggregator as 6 and the number of processes as 50

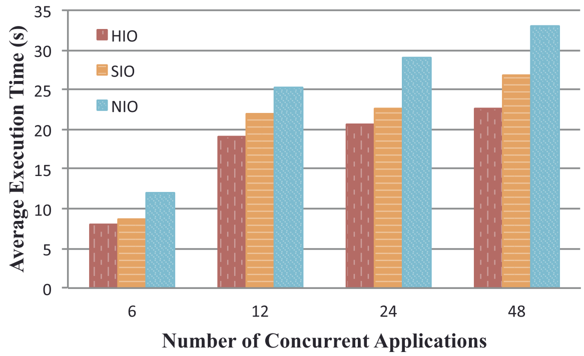

Fig. 8. Average execution time with concurrent applications.

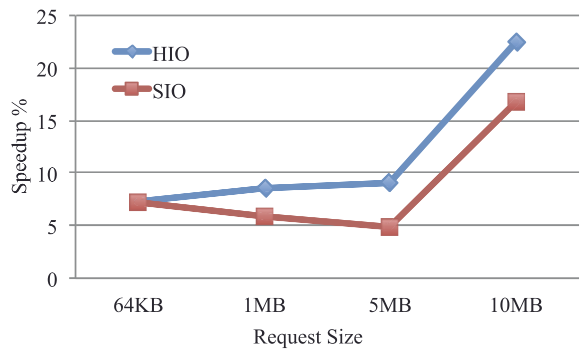

Fig. 9. Speedup of HIO and SIO with different request size.

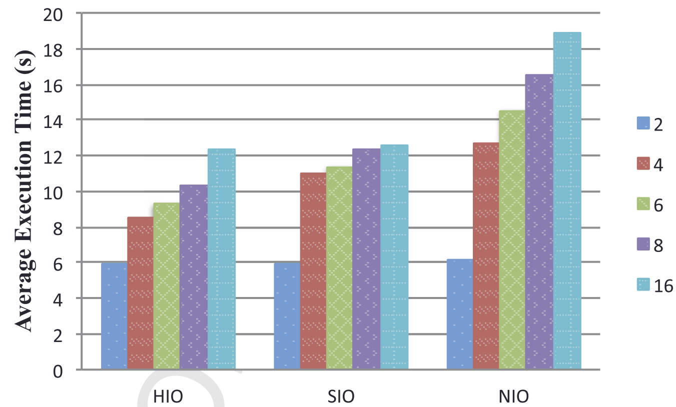

Fig. 10. Average execution time with different number of storage nodes.

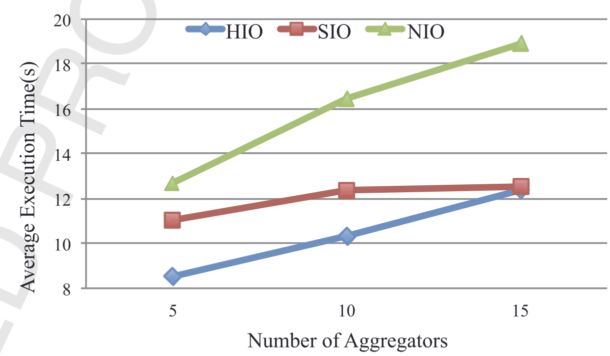

Fig. 11. Average execution time with different number of aggregators.

for each application. The I/O request size was set to a fixed value, 16 MB. We run with 6, 12, 24, and 48 processes simultaneously. Six storage nodes were deployed. The results are plotted in Fig. 8. From the figure, we can observe that the HIO scheduling outperformed other scheduling. The total execution time was decreased by up to 34.1% compared with NIO and by up to 15.2% compared with SIO. Furthermore, when the number of concurrent applications increased, the performance gain was even better.

We have conducted experiments with varying the request size too. As reported in Fig. 9, the I/O request size was set as 64 KB, 1 MB, 5 MB, and 10 MB respectively. The number of concurrent applications was set as six, and the number of aggregators was configured as six too. During this test, we compared the ratio of shuffle cost against the total cost. It was found that the ratio increased from 0.7% to 5.6%, as the request size increased. This fact matches with our observation that the shuffle cost considerably increases when applications become more and more data intensive.

When the requests size increased, the performance gain of using HIO scheduling was increased too, from 6.8% to 18.3% in terms of the execution time reduction rate. This result also matches with our theoretical analysis discussed in Section 4.

We have also evaluated the impact of the number of storage nodes and report the results in Fig. 10. In this test, there are 6 applications running simultaneously, and the number of aggregators in each application was set the same as the number of storage nodes, in order to have each application access all storage nodes. The request size was set as 15 MB, and the aggregator's request size is equal. The number of storage nodes was varied as 2, 4, 6, 8, and 16.

We observe that, from Fig. 10, the normal collective I/O did not scale well with the increasing size of the system. While both HIO and SIO achieved better scalability, we also find that the HIO performed and scaled better than SIO. The advantage of the HIO is due to the reduced shuffle cost. As the number of storage nodes increased, the inter-communication between aggregators and processes on different nodes also increased, which has been confirmed in a prior study too [4]. It can be projected that, as the system scale keeps increasing in the big data computing era, the shuffle cost in the two-phase collective I/O will become a critical issue. The proposed HIO scheduling in this study is essentially for addressing this issue and is likely to be promising at the exascale/extreme scale of HPC system.

We also evaluated the HIO with various numbers of aggregators. In Fig. 11, we test the HIO, SIO and NIO with 5, 10 and 15 aggregators separately. As the number of aggregator increasing, the interruption and the shuffle cost will also increase. We can find that the NIO shows a linear increasing trend. Both the HIO and SIO reduce the average execution time. The performance gain of SIO comes from the reduction of interruption of aggregators on storage nodes. While the HIO achieves more by reducing the shuffle cost.

As we have discussed in Section 6, the HIO was originally designed for collective read. For collective write, since the shuffle phase is already done before any scheduling, so the HIO cannot utilize the various acceptable delays any more, therefore, we can see from Fig. 12, the HIO and SIO do not distinguish a lot. Both of them achieve an average speedup about 12%.

We have evaluated the HIO scheduling with a real climate science application and datasets from the Bjerknes Center for Climate Research as well [24]. This set of tests was specifically for understanding the benefits of the HIO scheduling for a special access pattern, accessing 2D planes in scientific datasets. In scientific computing, scientists are interested in understanding the phenomenon behind the data by performing subsets queries [24]. Those subsets queries usually happen in the 2D planes, e.g., parameters along time dimension and level dimension in a climate science data. The datasets can range between GBs and TBs. Previous studies have shown the poor scalability of collective I/O due to high concurrency and I/O interruption. The proposed HIO al-

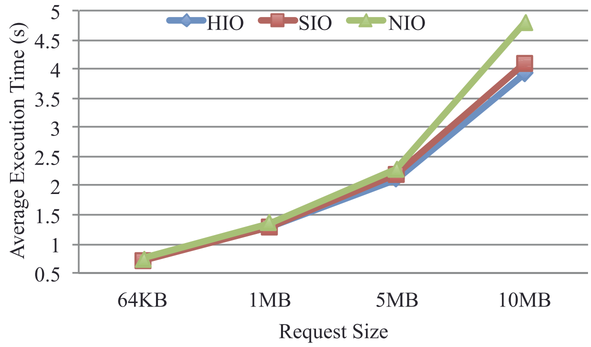

Fig. 12. Collective write with HIO.

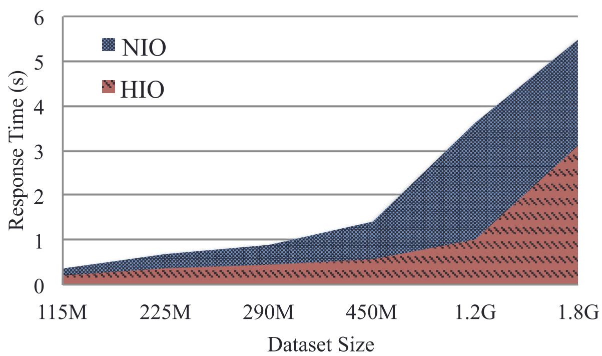

Fig. 13. FASM with HIO.

gorithm addresses this issue by better scheduling concurrent I/Os. The total dataset size evaluated in this series of tests is more than 12 GB. We run multiple 2D subsets queries concurrently using the FASM system [24] and performed the HIO scheduling. We run 20 queries for each dataset. A sample query statement is like "select temperature from dataset where 10 < temperature < 31". These queries were generated randomly and followed a global normal distribution. The performance gain with the HIO scheduling compared to the conventional collective I/O is shown in Fig. 13. It can be observed that the HIO scheduling improved the average query response time clearly and by up to 59.8%.

All these tests have well confirmed that the proposed hierarchical I/O scheduling in this study can improve the performance of collective I/O given highly concurrent and interrupted accesses. It holds a promise for big data problems and scientific applications on large-scale HPC systems.

9. Related work

Nowadays, the big data problem has attracted interests from different research areas from industry and academia [3,40]. We compare our work with the most related recent works.

9.1. I/O scheduling

Parallel I/O scheduling has been widely studied by many researchers at a hope of obtaining the peak sustained I/O performance. Few of them, however, meets the current demand of data-intensive applications and big data analysis yet. Disk-directed I/O [20] and server-directed I/O [35] have been proposed to improve the bandwidth of disks and network servers respectively. There are also numerous scheduling algorithms targeting the quality of service (QoS) [14,15,17,33,41]. The proposed hierarchical I/O scheduling in this study takes one step further to optimize the scheduling of collective I/O while considering the highly concurrent I/O requests from data-intensive applications.

In [36], a server-side I/O coordination method is proposed for parallel file systems. Their idea is to coordinate file servers to serve one application at a time in order to reduce the average completion time, and in the meantime maintain the server utilization and fairness. By re-arranging I/O requests on the file servers, the requests are serviced in the same order in terms of applications on all involved nodes. However, without considering the shuffle cost in the collective I/O, it is unlikely to achieve the optimal performance. In [43], the authors proposed a scheme namely IOrchestrator to improve the spatial locality and program reuse distance by calculating the access distances and grouping the requests with small distance together. These two works seem similar but differ in the motivation. The first one is based on the observation that the requests with synchronization needs will be optimized if they are scheduled at the same time, whereas the latter one is motivated by exploring the program's spatial locality.

In [16], the authors proposed three scheduling algorithms, with considering the number of processes per file stripe and the number of accesses per process, to minimize the average response time in collective I/O. In servicing one aggregator, instead of scheduling one stripe at a time in the increasing file offset order, they propose to prioritize the file stripes based on their access degree, the number of accessing processes. Their work optimized the scheduling of stripes within an aggregator, whereas our work focuses on the scheduling of aggregators. Besides, their work only considers the average I/O response time. The reduced I/O response time, however, does not always lead to the reduced total cost that includes the I/O response time and the shuffle cost.

9.2. Two phase collective I/O optimization

Collective I/O has been proposed about 15 years [26,28,37], optimization of collective I/O has never been stopped. We have studied most of the work related to two phase collective I/O, from our classification, the research efforts have been focused on the following points:

1) Implementation [8,37,39], in which the collective I/O are designed implemented and advanced feature are supported;
2) File view partition [19,42], which are related to the two phase collective I/O's global file view partition to optimize the aggregator's I/O access;
3) Aggregator selection [5]. Interesting ideas are about how to selection the processes as aggregators and how to define the number of aggregators;
4) Cache and buffering [22,29]. Researchers find the traditional cache and buffering idea can be utilized to optimize the collective I/O, which is done on the client side;
5) Data compression [10] is another example that other well-studied ideas are used in collective I/O.

All these works have successfully improved the performance of two phase collective I/O, and proved that the collective I/O is promising in the parallel computing. But we can also find that the new challenges in today's large scale, big data, and power constrained era, drive the further development of collective I/O, and our HIO is proposed under this circumstance. Among the previous works in collective I/O, there are two which are most related.

The first one is in [4,5], the authors discussed the increasing shuffle cost in today's HEC system too. Their discussions are for motivating the importance of node re-ordering for reducing the collective I/O's shuffle cost. Their work provides a method to evaluate the shuffle cost and designed algorithms to automatically assign the aggregators at the node level, whereas our work focuses on the scheduling of aggregators considering highly concurrent accesses to achieve the optimal collective I/O performance.

The second one is the LACIO idea [6], in which the author discuss the gap between logical access and physical storage. When the aggregators are assigned with different file view, the view itself is just logical space, therefore, the aggregator may not know the physical distribution of data on the parallel storage nodes. In other words, the aggregator from the same application will interrupt with each other on the storage nodes and the LACIO idea well addressed the issue. Our HIO idea targets the interruption among concurrent applications, which in another side, to improve the performance (we did our evaluation by first removing the interruption from the same application, such that we can tell the HIO really reduces the interruption among different applications). Previous work, like LACIO, focuses on one application's collective I/O, while our work goes one more step to optimize the performance of concurrent case. Both of the direction, when combined, is indeed a trend for extreme scale systems in the future.

9.3. Data organization and file systems

Our work focuses on I/O scheduling, while the data is not movable. In fact, there are bunch of work targeting the data organization and file systems. We concluded some related work in this last subsection. For example, there exist other works that address the scientific data retrieval issues by optimizing the data organization [18,23]. These works provide efficient mechanisms from the data level and fit the access pattern of scientific applications. Our work also improves the application's accesses, most of which are non-contiguous, but through a hierarchical scheduling. There are also works utilizing existing database techniques and compression algorithms to boost the big data analysis. For example, Fastbit implemented bitmap index in the large datasets [7]. ISABELA improved the big data query by compressing the datasets [21]. Our work focuses on collective I/O scheduling, which is beneficial and critical to big data retrieval and analysis too. In the future, we would also apply machine learning algorithms [31,32] to further refine the scheduling.

10. Conclusion

Collective I/O has been proven a critical technique in optimizing the non-contiguous access pattern in many scientific applications run on high-performance computing systems. It can be critical for big data retrieval and analysis too as non-contiguous access pattern also commonly exists in big data problems. The performance of collective I/O, however, could be dramatically degraded due to the increasing shuffle cost caused by highly concurrent accesses and interruptions. This problem tends to be more and more critical as many applications become highly data intensive. In this study, we propose a new hierarchical I/O scheduling for collective I/O to address these issues. This approach is the first considering the increasing shuffle cost involved in collective I/O. Through theoretical analyses and experiments, it has been confirmed that the hierarchical I/O scheduling can improve the performance of collective I/O. In the future, we will apply a similar approach for write operations. We will analyze the feasibility of implementing hierarchical I/O scheduling only at the MPI-IO layer as well. More experiments will be conducted to analyze how the shuffle cost can affect the big data analysis and further refine our algorithm. We will also try to apply similar approaches for write operations and develop different scheduling methods for different parallel file systems.

Acknowledgements

This research is sponsored in part by the National Science Foundation under grant CNS-1162488 and the Texas Tech University startup grant. The authors are thankful to Yanlong Yin of Illinois Institute of Technology and Wei-Keng Liao of Northwestern University for their constructive and thoughtful suggestions toward this study. We also acknowledge the High Performance Computing Center (HPCC) at Texas Tech University for providing resources that have contributed to the research results reported within this paper.

References

[1] The global cloud resolving model (GCRM) project, http://kiwi.atmos.colostate.edu/gcrm/.
[2] MPI-IO test, http://public.lanl.gov/jnunez/benchmarks/mpiiotest.htm.
[3] C.P. Aibek Musaev, De Wang, Litmus: landslide detection by integrating multiple sources, in: The 11th International Conference on Information Systems for Crisis Response and Management, 2014.
[4] K. Cha, S. Maeng, Reducing communication costs in collective I/O in multi-core cluster systems with non-exclusive scheduling, J. Supercomput. 61 (3) (2012) 966–996.
[5] M. Chaarawi, E. Gabriel, Automatically selecting the number of aggregators for collective I/O operations, in: CLUSTER, IEEE, 2011, pp. 428–437.
[6] Y. Chen, X.-H. Sun, R. Thakur, P.C. Roth, W.D. Gropp, LACIO: a new collective I/O strategy for parallel I/O systems, in: IPDPS, IEEE, 2011, pp. 794–804.
[7] J. Chou, M. Howison, B. Austin, K. Wu, J. Qiang, E.W. Bethel, A. Shoshani, O. Rübel, Prabhat, R.D. Ryne, Parallel index and query for large scale data analysis, in: Conference on High Performance Computing Networking, Storage and Analysis, SC 2011, Seattle, WA, USA, November 12–18, 2011, ACM, 2011.
[8] P. Dickens, R. Thakur, Improving collective I/O performance using threads, in: Proceedings of the Joint International Parallel Processing Symposium and IEEE Symposium on Parallel and Distributed Processing, Apr. 1999, pp. 38–45.
[9] P.M. Dickens, R. Thakur, Evaluation of collective I/O implementations on parallel architectures, J. Parallel Distrib. Comput. 61 (8) (Aug. 2001) 1052–1076.
[10] R. Filgueira, D.E. Singh, J.C. Pichel, J. Carretero, Exploiting data compression in collective I/O techniques, in: Proceedings of the IEEE International Conference on Cluster Computing, 10th CLUSTER'08, Tsukuba, Japan, Sept.-Oct. 2008, IEEE, 2008, pp. 479–485.
[11] K. Gao, W. Keng Liao, A.N. Choudhary, R.B. Ross, R. Latham, Combining I/O operations for multiple array variables in parallel netCDF, in: CLUSTER, IEEE, 2009, pp. 1–10.
[12] J. Gray, D.T. Liu, M.A. Nieto-Santisteban, A.S. Szalay, D.J. DeWitt, G. Heber, Scientific data management in the coming decade, SIGMOD Rec. 34 (4) (2005) 34–41.
[13] W. Gropp, E. Lusk, R. Thakur, Using MPI-2: Advanced Features of the Message Passing Interface, Scientific and Engineering Computation, MIT Press, 2000, pub-MIT:adr.
[14] A. Gulati, I. Ahmad, C.A. Waldspurger, PARDA: proportional allocation of resources for distributed storage access, in: FAST, USENIX, 2009, pp. 85–98.
[15] L. Huang, G. Peng, T. cker Chiueh, Multi-dimensional storage virtualization, ACM SIGMETRICS Perform. Eval. Rev. 32 (1) (June 2004) 14–24.
[16] C. Jin, S. Sehrish, W. Keng Liao, A.N. Choudhary, K. Schuchardt, Improving the average response time in collective I/O, in: EuroMPI, in: Lect. Notes Comput. Sci., vol. 6960, Springer, 2011, pp. 71–80.
[17] M. Karlsson, C. Karamanolis, X. Zhu, Triage: performance isolation and differentiation for storage systems, Technical Report HPL-2004-40, Hewlett Packard Laboratories, Sept. 2, 2004.
[18] W. Kendall, M. Glatter, J. Huang, T. Peterka, R. Latham, R.B. Ross, Terascale data organization for discovering multivariate climatic trends, in: SC, ACM, 2009.
[19] W. Keng Liao, A. Choudhary, Dynamically adapting file domain partitioning methods for collective I/O based on underlying parallel file system locking protocols, in: SC'08, ACM/IEEE, Austin, TX, Nov. 2008.
[20] D. Kotz, Disk-directed I/O for MIMD multiprocessors, Technical Report PCS-TR94-226, Dartmouth College, July 1994.
[21] S. Lakshminarasimhan, J. Jenkins, I. Arkatkar, Z. Gong, H. Kolla, S.-H. Ku, S. Ethier, J. Chen, C.-S. Chang, S. Klasky, R. Latham, R.B. Ross, N.F. Samatova, ISABELA-QA: query-driven analytics with ISABELA-compressed extreme-scale scientific data, in: Conference on High Performance Computing Networking, Storage and Analysis, SC 2011, Seattle, WA, USA, November 12–18, 2011, ACM, 2011.
[22] J. Liu, S. Byna, Y. Chen, Segmented analysis for reducing data movement, in: 2013 IEEE International Conference on Big Data, Oct. 2013, pp. 344–349.
[23] J. Liu, S. Byna, B. Dong, K. Wu, Y. Chen, Model-driven data layout selection for improving read performance, in: 2014 IEEE International Parallel Distributed Processing Symposium Workshops (IPDPSW), May 2014, pp. 1708–1716.
[24] J. Liu, Y. Chen, Improving data analysis performance for high-performance computing with integrating statistical metadata in scientific datasets, in: HPCDB, SC'12, 2012.
[25] J. Liu, Y. Chen, Y. Zhuang, Hierarchical I/O scheduling for collective I/O, in: Proc. of the 13th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, CCGrid'13, 2013.

[26] J. Liu, B. Crysler, Y. Lu, Y. Chen, Locality-driven high-level i/o aggregation for processing scientific datasets, in: 2013 IEEE International Conference on Big Data, Oct. 2013, pp. 103–111.
[27] J.F. Lofstead, M. Polte, G.A. Gibson, S. Klasky, K. Schwan, R. Oldfield, M. Wolf, Q. Liu, Six degrees of scientific data: reading patterns for extreme scale science IO, in: HPDC, ACM, 2011, pp. 49–60.
[28] Y. Lu, Y. Chen, Y. Zhuang, J. Liu, R. Thakur, Collective input/output under memory constraints, Int. J. High Perform. Comput. Appl. (2014).
[29] X. Ma, M. Winslett, J. Lee, S. Yu, Improving MPI IO output performance with active buffering plus threads, in: Proceedings of the International Parallel and Distributed Processing Symposium, IEEE Computer Society Press, Apr. 2003.
[30] J. No, R. Thakur, A. Choudhary, Integrating parallel file I/O and database support for high-performance scientific data management, in: Proceedings of SC2000: High Performance Networking and Computing, Dallas, TX, Nov. 2000, IEEE Computer Society Press, 2000.
[31] Y. Pang, F. Xiao, H. Wang, X. Xue, A clustering-based grouping model for enhancing collaborative learning, in: 2014 13th International Conference on Machine Learning and Applications, IEEE, 2014.
[32] Y. Pang, X. Xue, A.S. Namin, Identifying effective test cases through k-means clustering for enhancing regression testing, in: 2013 12th International Conference on Machine Learning and Applications (ICMLA), vol. 2, IEEE, 2013, pp. 78–83.
[33] A. Povzner, D. Sawyer, S.A. Brandt, Horizon: efficient deadline-driven disk I/O management for distributed storage systems, in: HPDC, ACM, 2010, pp. 1–12.
[34] R. Ross, R. Latham, M. Unangst, B. Welch, Parallel I/O in practice, tutorial notes, in: SC'08, ACM/IEEE, Austin, TX, Nov. 2008.
[35] K.E. Seamons, Y. Chen, P. Jones, J. Jozwiak, M. Winslett, Server-directed collective I/O in Panda, in: Proceedings of SC'95, San Diego, CA, Dec. 1995, IEEE Computer Society Press, 1995.
[36] H. Song, Y. Yin, X.-H. Sun, R. Thakur, S. Lang, Server-side I/O coordination for parallel file systems, in: SC, ACM, 2011.
[37] R. Thakur, W. Gropp, E. Lusk, Data sieving and collective I/O in ROMIO, in: Proc. of the 7th Symposium on the Frontiers of Massively Parallel Computation, IEEE, Feb. 1999, pp. 182–189.
[38] Y. Tian, S. Klasky, H. Abbasi, J.F. Lofstead, R.W. Grout, N. Podhorszki, Q. Liu, Y. Wang, W. Yu, EDO: improving read performance for scientific applications through elastic data organization, in: CLUSTER, IEEE, 2011, pp. 93–102.
[39] V. Venkatesan, M. Chaarawi, E. Gabriel, T. Hoefler, Design and evaluation of nonblocking collective I/O operations, in: 18th European MPI Users' Group Meeting, EuroMPI 2011, Santorini, Greece, September 18–21, vol. 6960, Springer, 2011, pp. 90–98.
[40] D. Wang, D. Irani, C. Pu, Evolutionary study of web spam: webb spam corpus 2011 versus webb spam corpus 2006, in: 2012 8th International Conference on Collaborative Computing: Networking, Applications and Worksharing (Collaboratecom), 2012, pp. 40–49.
[41] Y. Wang, A. Merchant, Proportional service allocation in distributed storage systems, Technical Report HPL-2006-184, Hewlett Packard Laboratories, Feb. 18, 2007.
[42] W. Yu, J.S. Vetter, Parcoll: partitioned collective I/O on the cray XT, in: ICPP, IEEE Computer Society, 2008, pp. 562–569.
[43] X. Zhang, K. Davis, S. Jiang, IOrchestrator: improving the performance of multinode I/O systems via inter-server coordination, in: SC'10, ACM/IEEE, New Orleans, LA, USA, Nov. 2010.

Please cite this article in press as: J. Liu et al., Hierarchical Collective I/O Scheduling for High-Performance Computing, Big Data Research (2015), http://dx.doi.org/10.1016/j.bdr.2015.01.007