Hierarchical I/O Scheduling for Collective I/O

Jialin Liu
Department of Computer Science
Texas Tech University
Lubbock, Texas, USA
Email: jaln.liu@ttu.edu

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

Yu Zhuang
Department of Computer Science
Texas Tech University
Lubbock, Texas, USA
Email: yu.zhuang@ttu.edu

***Abstract*—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 system 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. 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 MPICH2 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.**

***Keywords*-collective I/O; scheduling; high-performance computing; big data; data intensive computing**

I. 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. 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 [20].

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 [18, 24], 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 [6]. However, the performance of the collective I/O could be dramatically degraded when solving big data problems on a highly-concurrent HPC system [7, 10]. A critical reason is that the increasing shuffle cost of collective requests can dominate the performance [14]. 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. 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.

The rest of this paper is organized as follows. Section II reviews collective I/O and motivates this study by analyzing a typical example of interrupted collective read. Section III introduces the HIO scheduling algorithm. Section IV presents the theoretical analysis of the HIO scheduling. Section V discusses the implementation. The experimental results are discussed in Section VI. Section VII discusses related work and compares them with this study. Section VIII summarizes this study and discusses future work.

II. 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 [23]. A widely-used implementation of collective I/O is the two-phase I/O protocol. 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.

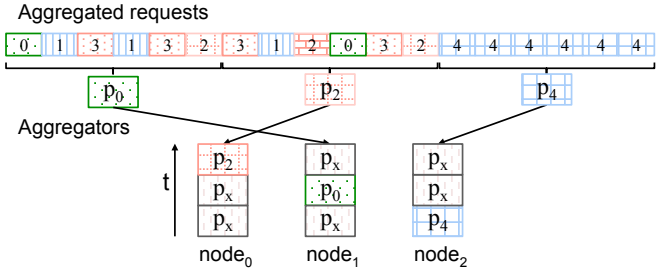

Figure 1: Interrupted Collective Read. (p_x is from other applications)

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. 1 shows a two-phase collective read operation, which is interrupted by processes from other concurrent applications due to highly concurrent accesses. In Fig. 1, 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. 1, 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. 2.

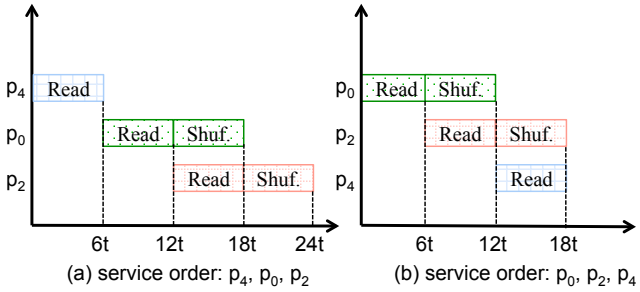

Figure 2: Two Different Service Orders.

In Fig. 2, 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 execution time is calculated as the sum of the read cost and the shuffle cost. In Fig. 2(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. 2(b), p_0 is serviced first. We find that the total 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. 2(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 node2 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 total 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.

III. Hierarchical I/O Scheduling

From the previous analysis, we can see that by scheduling slower aggregators earlier, the total 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' total execution time. From the observation in Fig. 2(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'. Then, we present the proposed hierarchical I/O scheduling algorithm.

A. Analysis and Prediction of Shuffle Cost

In order to analyze the shuffle cost, we need to know how aggregators are assigned and how they will communicate with other desired processes. Most previous works assumed that only one aggregator is assigned in one node [3], which is also a default configuration in MPICH. The communication cost thus includes inter-node and intra-node cost, as shown in Fig. 3. 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.

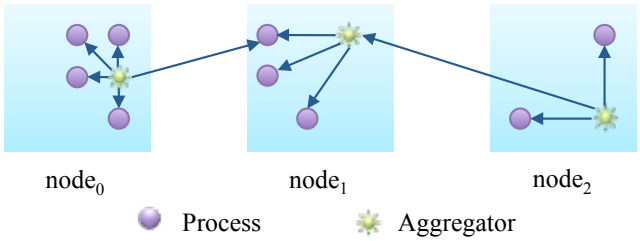

Figure 3: Example of Process Assignment in Three Node Multi-core System. The arrows show the inter and intra communication in the shuffle phase.

In Fig. 3, 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 results in intra-node communication. As a consequence, each aggregator will have different shuffle cost.

Based on the above analysis, we can derive the following equation to predict the shuffle cost:

$$\begin{aligned} T &= max(max(\frac{m_{ai}}{B_a}), max(\frac{m_{ej}}{B_e})) + \gamma \\ &= max(\frac{m_{ai}}{B_a}, \frac{m_{ej}}{B_e}) + \gamma \end{aligned} \quad (1)$$

where T is the total shuffle cost of one aggregator; m_{ai} is the i^{th} intra message size (MByte) and $0 < i < A$, where A is the total number of intra communication; m_{ej} is the j^{th} 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/sec); B_e is the saturated throughput of inter-node communication of a given cluster system (MB/sec); and γ is the latency.

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:

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

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 number of cores per node.

B. Acceptable Delay

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:

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

where AD_i is i^{th} aggregator's acceptable delay, and T_i is i^{th} 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.

C. 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 can not 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.

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 j_{th} aggregator on i_{th} node;
agg[i][j].read:the read cost of j_{th} aggregator on i_{th} node
agg[i][j].rad:the relative ad of j_{th} aggregator on i_{th} 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 is 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 IV. 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.

IV. 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-io scheduling [22].

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 j_{th} aggregator's shuffle cost of the i_{th} application). The longest finish time of requests on all nodes determines the application's completion time on the server side, which could be $\{t, 2t, 3t, ..., mt\}$. The application's total execution time 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 equation (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 i_{th} application's completion time on one node.

With the server-io scheduling, in which the same applications' requests are serviced at the same time on all nodes, the service time for those applications are fixed: $t, 2t, 3t, ..., mt$. The average total cost 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 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 total cost:

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

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 total average cost is:

$$\begin{aligned} T_i &= \frac{1}{m}((t + 2t + 3t + ... + mt) + m(max(s_{ij}))) \\ &= \frac{m+1}{2}t + max(s_{ij}) \end{aligned} \tag{7}$$

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 total execution time same with (5). In another word, the worst case of the HIO scheduling at least has the same performance as that of the server-io scheduling.

Comparing equations (4) and (5), the server-io scheduling can achieve an average cost reduction as the following:

$$\begin{aligned} T_{reduction} &= mt + max(s_{ij}) - \frac{t}{m^n}\sum_{x=1}^{m-1} x^n \\ &\quad - \frac{m+1}{2}t - max(s_{ij}) \\ &= \frac{m-1}{2}t - \frac{t}{m^n}\sum_{x=1}^{m-1} x^n \\ &= \frac{m-1}{2}t \ (n \to \infty) \end{aligned} \tag{8}$$

The best case of the HIO scheduling can further reduce the cost of (5) by:

$$\begin{aligned} T^b_{reduction} &= \frac{m+1}{2}t + max(s_{ij}) \\ &\quad - t - max(s_{ij}) \\ &= \frac{m-1}{2}t \end{aligned} \tag{9}$$

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.

V. 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.

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 III.

VI. Experiments and Analyses

A. Experimental Setup

We have conducted tests on a 16-node Linux testbed, with a total of 32 processors and 128 cores. 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-io scheduling (denoted as SIO) [22] and a normal collective I/O (denoted as NIO).

B. Results and Analyses

In the tests, we first 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 for each application. The I/O request size was set as 16 MBs, a fixed value. We run with 6, 12, 24, and 48 processes simultaneously. Six storage nodes were deployed. The results are plotted in Figure 4. 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 Figure 5, the I/O request size was set as 64KB, 1MB, 5MB, and 10MB respectively. The number of concurrent applications was set as six, and the number of aggregators was configured as six too. During these 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.

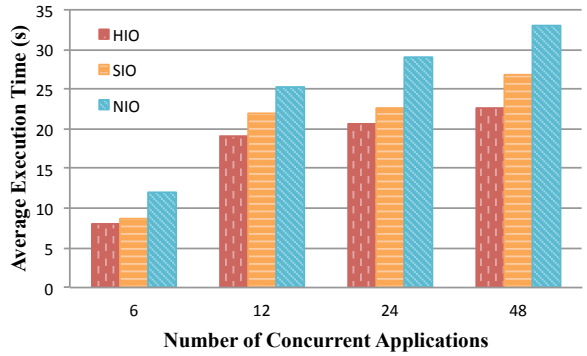

Figure 4: Average execution time with concurrent applications

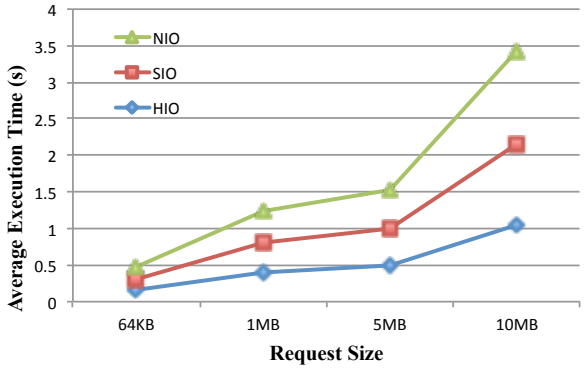

Figure 5: Average execution time with different request size

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 IV.

We have also evaluated the impact of the number of storage nodes and report the results in Figure 6. In these tests, 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 15MBs, 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 the Figure 6, 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 [3]. 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/extremescale of HPC system.

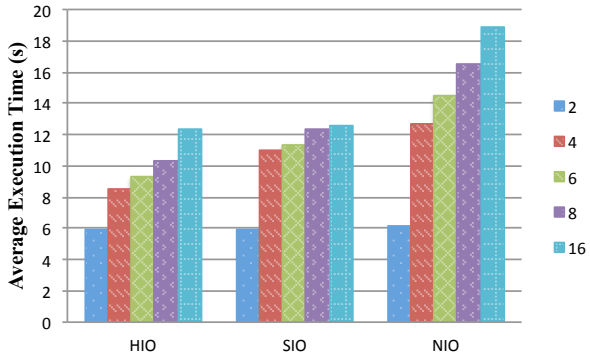

Figure 6: Average execution time with different storage nodes

We have evaluated the HIO scheduling with a real climate science application and datasets from the Bjerknes Center for Climate Research as well [17]. 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. 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 algorithm addresses this issue by better scheduling concurrent I/Os. The total dataset size evaluated in this series of tests is more than 12 GBs. We run multiple 2D subsets queries concurrently using the FASM system [17] 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 Table I. 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.

Table I: Average Response Time with HIO Scheduling

Dataset Size	NIO (ms)	HIO (ms)
115M	354.87	185.42
225M	702.71	361.09
290M	899.78	454.25
450M	1410.21	566.02
1.2G	3616.45	1027.08
1.8G	5490.84	3090.21

VII. Related Work

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[15] and server-directed I/O[21] 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) [8, 9, 19, 25]. 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 [22], 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 [26], 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 [11], 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.

In [3, 4, 13, 14], 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.

There exist other works that address the scientific data retrieval issues by optimizing the data organization [12, 18]. 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 [5]. ISABELA improved the big data query by compressing the datasets [16]. Our work focuses on collective I/O scheduling, which is beneficial and critical to big data retrieval and analysis too.

VIII. 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 the similar approach for write operations. We will analyze the feasibility of implementing hierarchical I/O scheduling only at the MPI-IO layer as well.

Acknowledgment

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] 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*, 61(3):966–996, 2012.

[4] M. Chaarawi and E. Gabriel. Automatically selecting the number of aggregators for collective I/O operations. In *CLUSTER*, pages 428–437. IEEE, 2011.

[5] J. Chou, M. Howison, B. Austin, K. Wu, J. Qiang, E. W. Bethel, A. Shoshani, O. Rübel, Prabhat, and 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.

[6] K. Gao, W. keng Liao, A. N. Choudhary, R. B. Ross, and R. Latham. Combining I/O operations for multiple array variables in parallel netCDF. In *CLUSTER*, pages 1–10. IEEE, 2009.

[7] J. Gray, D. T. Liu, M. A. Nieto-Santisteban, A. S. Szalay, D. J. DeWitt, and G. Heber. Scientific data management in the coming decade. *SIGMOD Record*, 34(4):34–41, 2005.

[8] A. Gulati, I. Ahmad, and C. A. Waldspurger. PARDA: Proportional allocation of resources for distributed storage access. In *FAST*, pages 85–98. USENIX, 2009.

[9] L. Huang, G. Peng, and T. cker Chiueh. Multi-dimensional storage virtualization. *ACM SIGMETRICS Performance Evaluation Review*, 32(1):14–24, June 2004.

[10] Y. C. X.-H. S. Hui Jin, Tao Ke. Checkpointing orchestration: Toward a scalable hpc fault-tolerant environment. In *In the Proc. 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing(CCGrid'12)*, 2012.

[11] C. Jin, S. Sehrish, W. keng Liao, A. N. Choudhary, and K. Schuchardt. Improving the average response time in collective I/O. In *EuroMPI*, volume 6960, pages 71–80. Springer, 2011.

[12] W. Kendall, M. Glatter, J. Huang, T. Peterka, R. Latham, and R. B. Ross. Terascale data organization for discovering multivariate climatic trends. In *SC*. ACM, 2009.

[13] W. keng Liao. Design and evaluation of MPI file domain partitioning methods under extent-based file locking protocol. *IEEE Trans. Parallel Distrib. Syst*, 22(2):260–272, 2011.

[14] W. keng Liao and 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.

[15] D. Kotz. Disk-directed I/O for MIMD multiprocessors. Technical Report PCS-TR94-226, Dartmouth College, July 1994.

[16] 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, and N. F. Samatova. ISABELA-QA: query-driven analytics with ISABELA-compressed extreme-scale scientific data. In *SC*. ACM, 2011.

[17] J. Liu and Y. Chen. Improving data analysis performance for high-performance computing with integrating statistical metadata in scientific datasets. Accepted to appear in the Second Annual Workshop on High-Performance Computing meets Databases (HPCDB), in conjunction with the ACM/IEEE Supercomputing Conference (SC'12), 2012.

[18] J. F. Lofstead, M. Polte, G. A. Gibson, S. Klasky, K. Schwan, R. Oldfield, M. Wolf, and Q. Liu. Six degrees of scientific data: reading patterns for extreme scale science IO. In *HPDC*, pages 49–60. ACM, 2011.

[19] A. Povzner, D. Sawyer, and S. A. Brandt. Horizon: efficient deadline-driven disk I/O management for distributed storage systems. In *HPDC*, pages 1–12. ACM, 2010.

[20] R. Ross, R. Latham, M. Unangst, and B. Welch. Parallel I/O in practice, tutorial notes. In *SC'08*. ACM/IEEE, Austin, TX, Nov. 2008.

[21] K. E. Seamons, Y. Chen, P. Jones, J. Jozwiak, and M. Winslett. Server-directed collective I/O in Panda. In *Proceedings of SC '95*, San Diego, CA, Dec. 1995.

[22] H. Song, Y. Yin, X.-H. Sun, R. Thakur, and S. Lang. Server-side I/O coordination for parallel file systems. In *SC*. ACM, 2011.

[23] R. Thakur, W. Gropp, and E. Lusk. Data sieving and collective I/O in ROMIO. In *Proc. of the 7th Symposium on the Frontiers of Massively Parallel Computation*, pages 182–189. IEEE, Feb. 1999.

[24] Y. Tian, S. Klasky, H. Abbasi, J. F. Lofstead, R. W. Grout, N. Podhorszki, Q. Liu, Y. Wang, and W. Yu. EDO: Improving read performance for scientific applications through elastic data organization. In *CLUSTER*, pages 93–102. IEEE, 2011.

[25] Y. Wang and A. Merchant. Proportional service allocation in distributed storage systems. Technical Report HPL-2006-184, Hewlett Packard Laboratories, Feb. 18 2007.

[26] X. Zhang, K. Davis, and S. Jiang. IOrchestrator: Improving the performance of multi-node I/O systems via inter-server coordination. In *SC'10*. ACM/IEEE, New Orleans, LA, USA, Nov. 2010.