Locality-driven High-level I/O Aggregation for Processing Scientific Datasets

Jialin Liu, Bradly Crysler, Yin Lu,Yong Chen
Department of Computer Science
Texas Tech University
Lubbock, Texas, USA

{jaln.liu, bradly.crysler, yin.lu, yong.chen}@ttu.edu

***Abstract*—Scientific I/O libraries, like PnetCDF, ADIOS, and HDF5, have been commonly used to facilitate the array-based scientific dataset processing. The underlying physical data layout information, however, is usually hidden from the upper layer's logical access. Such mismatching can lead to poor I/O. In this research, we have observed performance degradation in the case of concurrent sub-array accesses, where overlaps among calls that access sub-arrays led to high contention on storage servers due to the logical-physical mismatching. We propose a locality-driven high-level I/O aggregation approach to addressing these issues in this work. By designing a logical-physical mapping scheme, we try to utilize the scientific dataset's structured formats and the file systems' data distribution to resolve the mismatching issue. Therefore the I/O can be carried out in a locality-driven fashion. The proposed approach is effective and complements existing I/O strategies, such as the independent I/O or collective I/O strategy. We have also carried out experimental tests and the results confirm the performance improvement compared to existing I/O strategies. The proposed locality-driven high-level I/O aggregation approach holds a promise for efficiently processing scientific datasets, which is critical for the data intensive or big data computing era.**

***Keywords*-Big data; data intensive computing; high performance computing; collective I/O; I/O aggregation; scientific I/O library**

I. Introduction

During the pre-big data era, the main cause of I/O bottlenecks was discovered to be disk-read speeds that would inevitably hinder the performance across both the CPU and memory bandwidths. With the high performance parallel I/O interface, e.g., MPI-IO, the I/O performance has been largely improved. However, as the volume of data collected from instruments and scientific simulations keeps increasing rapidly, we still need to optimize the I/O performance by better understanding some new challenges. For example, the Global Cloud Resolving Model (GCRM) project [5], 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. Such highly structured datasets are commonly managed by scientific libraries, e.g., PnetCDF [4], HDF5 [2], and ADIOS [1]. Those libraries not only provide highly structured array formats, but are also built upon high-performance I/O interface to facilicate the data processing.

When domain scientists write programs to access the datasets, they will first specify a logical structure, e.g., start position and size for each dimension, then the library will parse the logical structure and access the physical storage. The inconsistency between the *logical access* and the *physical layout* can result in a large amount of small non-contiguous I/O to occur leading to the degradation of system performance. Currently, two-phase collective I/O is designed to optimize the non-contiguous I/O through process collaboration [10, 18]. In two-phase collective I/O, all processes will share their access information and only a small portion of processes will be assigned as I/O processes to do the actual I/O. The collaboration among processes improves the performance of each individual call while the nonblocking version further optimizes the collective I/O across multiple calls.

In our research, however, we have observed performance degradation in the case of multiple overlapping sub-array accesses, in which, each call accesses a uniquely shaped subset and different calls are overlapped to some degree. Such overlapping should be eliminated before we use the collective call. In this paper, we propose a *light-weight locality-driven high-level I/O aggregation (HiLa)* approach, in which the request will be first aggregated in a high level before it is sent to the function. In order to perform such aggregation, the proposed approach first decomposes the data request, and then aggregates the decomposed requests that are physically close. A logical-physical mapping scheme, namely *sub-correlation set*, is designed to reveal the physical layout to the high-level aggregation. After the aggregation, the concurrent I/O calls rarely overlap when they arrive at the physical storage. The proposed HiLa approach, when used to further optimize the existing I/O strategies, shows increased performance, and demonstrates potential in big data analysis. The overhead of such high-level aggregation is very small since it is performed at the logical level and does not move any actual data. The HiLa was initially motivated by our previous work, Fast Analysis with Statistical Metadata (FASM) [14], in which we demonstrated and as shown in the evaluation section, how concurrent queries cause poor performance. We are currently analyzing this issue in a broader view and generalizing it with

existing I/O strategies.

The contribution of this research is three-fold. First, we propose an idea of high-level I/O aggregation while considering the locality. Second, we have derived functions to calculate the mapping between physical layout and logical structure. Third, we have carried out theoretical analyses and experimental tests to verify the efficiency of the proposed method. The results have confirmed that the HiLa approach is promising in improving I/O performance in big data analysis.

The rest of this paper is organized as follows. Section II reviews collective I/O and scientific datasets. Section III motivates this study with an initial comparison of existing I/O strategies in the case of multiple overlapping accesses. Section IV introduces the HiLa approach and algorithm. The experimental results are discussed in Section V. Section VI discusses related works and compares them with this study. Section VII summarizes this study and discusses future work.

II. Background

A. 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.

MPI-IO is a subset of the MPI-2/MPI-3 specification [10]. 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 datatypes. By equiping 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 [18]. It provides an astract-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 [18]. This strategy serves the I/O requests using an I/O phase and a data exchange phase. For example, in the case of a collective read, the first phase constists 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 designed for an individual call from all processes involved in a communicator that is specified in the collective I/O call. The nonblocking collective I/O in MPI is an optimization for overlapping communication and computation across multiple calls [11]. The non-blocking send/receive techniques allow the programmer to leverage the CPU during the asynchronous I/O. The nonblocking collective I/O combines the advantages of collective operations with overlapped communication and computation in modern communication architecture.

B. Scientific Datasets and Scientific I/O Libraries

In today's scientific data management, scientific dataset libraries are widely used, e.g., ADIOS [1], HDF5 [2], and NetCDF/PnetCDF [3, 4]. Those libraries provide high structured array formats to store the data and are built upon MPI to facilitate data processing. Taking the PnetCDF as an example, one of the most interesting features is the various forms of data access patterns supported [3, 13]. Those patterns include accessing all elements, individual elements, array sections, subsampled array sections, etc. We illustrate the most common access pattern, accessing a sub-array, in Figure 1.

```
//specify start position and length
start[0]=1;
Start[1]=1;
start[2]=1;
start[3]=1;
length[0]=1;
length[1]=1;
length[2]=max_length;
length[3]=2;
//collective put
int ncmpi_put_vara_float_all(
int ncid,
int varid,
const MPI_Offset start[],
const MPI_Offset count[],
float *fp);
```

Fig. 1: Array Access Pattern in Scientific Datasets

In Figure 1, we demonstrate how to access a sub-array of a 4-D dataset. This dataset has four dimensions, i.e., longitude, latitude, level, and time. For this access, the code first specifies

the start position and the access length of the sub-array on each dimension. The collective put function is chosen to issue a two-phase collective I/O. For example, if we run with 10 processes, all the processes will collaborate with each other to access such a sub-array.

PnetCDF also implements a nonblocking collective I/O interface [9]. For the nonblocking collective I/O, similar with MPI-IO's nonblocking communication routines, a program posts one or more operations and then waits for completion of those operations. The PnetCDF, however, focuses more on optimizing multiple I/O operations in processing an array-based scientific dataset. Once the PnetCDF has a list of all outstanding I/O operations, it can construct a single I/O request encompassing all operations. The interface is designed for multiple variable accessing but can be also used for single variable accessing. Typically, more information about I/O activity results in more opportunities for optimization. Storage systems perform better with larger I/O requests, so coalescing I/O operations in this way can yield performance improvements.

III. Motivation and Idea

In this section, we discuss the collective I/O and its drawback in the sub-array accesses through an illustrating example. We also present the results of a set of tests that compared three I/O strategies, i.e., independent I/O, collective I/O, and nonblocking collective I/O. We show how these I/O strategies were observed with performance degradation when there are concurrent overlapping sub-array calls. Finally, with rethinking the current collective I/O strategy, we propose a new idea of *High-level Aggregation* with the consideration of locality. The proposed locality-driven high-level I/O aggregation complements existing I/O strategies and can have a profound impact.

A. Collective I/O Performance with Concurrent Calls

In an MPI program, multiple read/write calls can be issued, where each call can have multiple processes associated with it and access different subsets. Such an access pattern is not rare in real applications. For example, in climate science applications, domain scientists often write programs to read subsets of different variables for the same time-step or various shapes' subsets of one variable. Another example is the query system for scientific data analysis. Both the FastQuery [7] and our prior work FASM [14] have situations that multiple queries run simultaneously. Each query is one call, and optimizations across calls need careful thinking. Traditional two-phase collective I/O, however, often performs inadequately in this situation, such as shown in Figure 2.

In Figure 2, there are multiple concurrent collective calls, and each call accesses a sub-array. When using the traditional two-phase collective I/O, we can see that the 'collective' operation is only performed within each individual call. In other words, there is no optimization for such 'collective' operations across multiple calls (except a nonblocking version, we will discuss it later). Recall that a collective operation takes the requests' offset list as input. The problem is, without a global

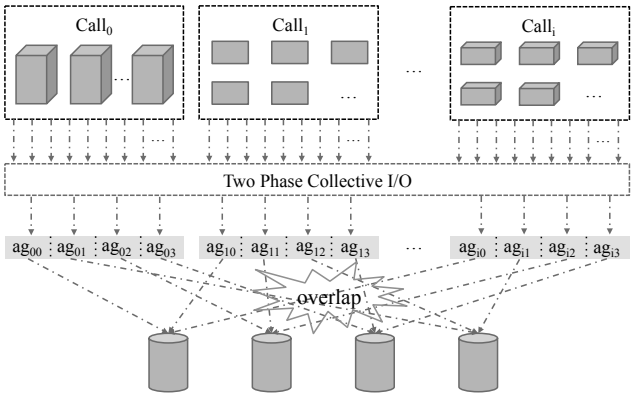


Fig. 2: Decreased Locality and Increased Concurrency due to the Lost of Layout Information in Two-phase Collective I/O

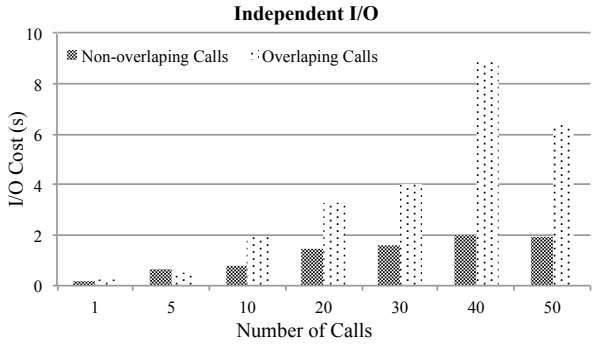
offset list across multiple calls, different calls' aggregators will have overlap. Therefore, the aggregators from different calls will compete for the same storage resources and cause contention. The collective I/O's performance can be poor in given concurrent calls situations.

B. Nonblocking Collective I/O Meets Challenges

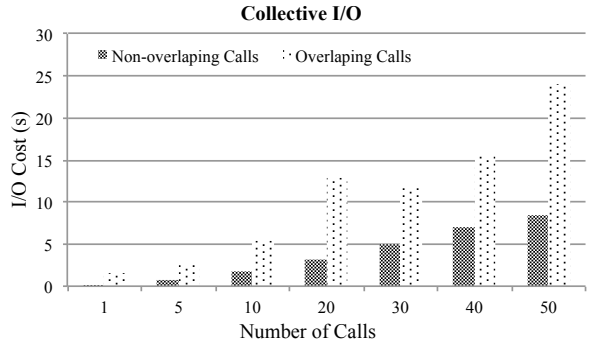
A straightforward idea is to combine multiple calls and perform a global collective operation. One solution is the nonblocking collective I/O, in which, a 'begin/wait' function is used and all collective calls are cached and the global offset list will be sorted to form a global file domain. Then the final I/O aggregators will be assigned with non-overlapping sub-domains divided from the global domain. In this way, the contention can be reduced and performance can be improved. Such a benefit was confirmed in our test. In addition, we have new finds.

In this test, we generated a 3.8GB 4-D NetCDF dataset and stored it in a Lustre file system. The data are striped across 40 storage nodes, with a stripe size of 100MB. We developed an MPI program, in which, multiple read calls were carried out. Each call was invoked by 50 processes for accessing a different sub-array. We tested three I/O methods separately. For each I/O method, we also performed two groups of tests, i.e., non-overlapping access tests and overlapping access tests. The non-overlapping and overlapping accesses are specified with same dimension and same size for each dimension, only difference is the start position on the record dimension, in which overlapping accesses are designed to have overlapping start positions. The reason why we did this is to eliminate the effects of accessing different dimension, e.g., fast dimension or slow dimension [15]. For the latter test, different calls access a small portion of common data. As we have shown in Figure 2, one call may access a 3-D subset, and another call accesses a 2-D plane. There are overlapping accesses but they are unknown to each call.

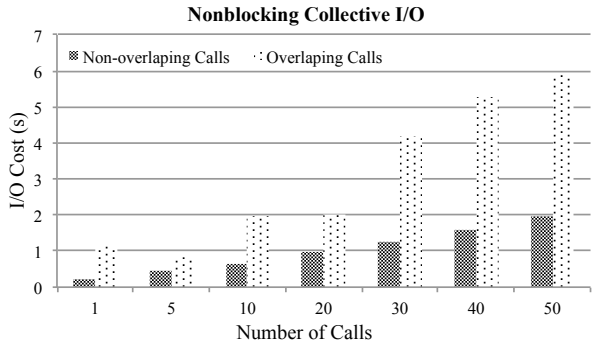
For independent I/O, each call was issued by multiple processes to access the data. There is no communication among processes, including among different calls. For collective I/O, processes are coordinated within each call, but there is no



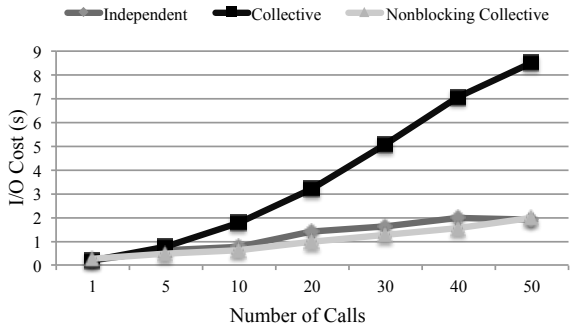
(a)



(b)



(c)



(d)

Fig. 3: Initial Performance Comparison

coordination across calls. For nonblocking collective I/O, all calls work together to conduct collective data accesses. The size of each call in this test was 77.82MB. A comprehensive evaluation is presented in Section V. As shown in Figure 3, we have three observations as discussed below:

1) As the number of calls was increased, the I/O cost increased too;
2) As shown in subfigure (d), nonblocking I/O performed best among all I/O methods;
3) As shown in subfigures (a)-(c), none of three I/O methods' performance was impressive when there were overlapping calls. The performance tended to be worse as the number of calls was increased.

The results confirm our analysis in subsection A. When the number of calls keeps increasing, the contention among calls and processes cause performance degradation, especially for two-phase collective I/O. When there is overlapping among calls, all the current I/O methods double their costs and tend to be even worse. For independent I/O, the performance degradation was caused by locking overhead. For nonblocking I/O, the overlapping among different calls generates non-monotonic offsets. As a result, multiple two-phase collective I/O were carried out instead of one global collective call. The current MPI-IO library is still being developed, and this issue may be addressed in future release. For scientists, however, not just one I/O method is used in their codes. From the analysis and initial evaluation presented in this section, we believe that the overlaps among concurrent calls, no matter which I/O method used, should be removed.

C. Our Idea: An Alternative Solution

Through the analysis of those existing I/O strategies, we can conclude that the performance issue is essentially caused by the poor locality, i.e., when storing the datasets onto storage nodes, there is an mismatching between the physical layout and the datasets' logical structure. Such mismatching messes up the upper layers' requests. Therefore, overlap and contention in accesses occurred. The dataset's structure information is lost when the data is stored onto the physical storage. The relationship between the physical layout and the logical structure is unknown to the MPI-IO library. By figuring out how the dataset's logical structure is mapped with its physical layout, we can optimize sub-array accesses in scientific data processing.

It should be noted that currently the two-phase I/O is performed at byte level, which means a list of offsets (with units of bytes) from different processes are taken as input for the aggregation. Instead, our idea is to perform a high-level collective operation, which means that each call's array structure is collected first in terms of the start and length on each dimension, then the requests are decomposed to sub-requests and those *physically close* requests are aggregated. In other words, a high-level aggregation is performed based on the logical information. This strategy can efficiently reduce the underlying contention and overlaps. For example, as shown in Figure 4, on the left hand, each call will access two

nodes independently without a high-level aggregation. On the right hand, four decomposed sub-calls will reform two new aggregated calls, and each node will receive only one collective call, which is more efficient than the case without a high-level aggregation.

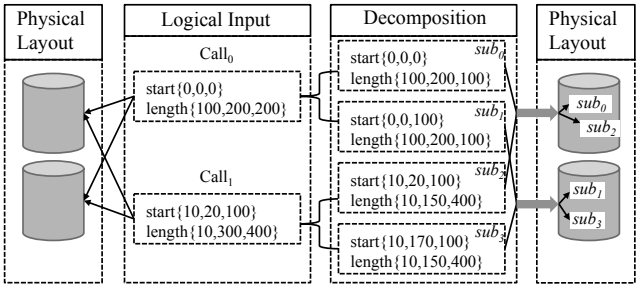


Fig. 4: Physical-Layout-Aware Decomposition and Aggregation of High-level Structure

A high-level I/O aggregation can have three challenges: 1) how to decompose the sub-array call; 2) how to aggregate the sub-arrays at a high level; and 3) how to calculate and utilize the physical layout information. We will present our solution to address these challenges in the next section.

IV. Locality Driven High-level Aggregation

We can see in the previous sections that, when there are multiple concurrent sub-array accesses, the inconsistency between logical access and physical layout results in poor performance. In order to map the physical layout with the high-level's logical accesses, we need to know the exact location of the data. Since the datasets are stored in a structured order, the location information can take a subset as a unit, which is easier to maintain than taking each data item as a unit. Using each subset's location, we can derive the correlation of different subsets, e.g., whether two subsets are placed on one node or not. Previously, this information is not known to the upper layers' access. Knowing the correlation among different subsets, we can provide a clear mapping between logical access and physical placement. The upper layer's subsets requests can utilize such information to determine which subset requests or sub-subsets within the requests should be aggregated.

First, we introduce a *sub-correlation set* to support the mapping. For an n-D dataset, the sub-correlation set contains the subsets details within each stripe. The subsets details reveal that how many subsets are within the stripe, as well as the subsets' start and size on each dimension in this stripe. Second, we design the *High-level Aggregation* algorithm through utilizing the *sub-correlation set*.

A. Sub-correlation Set

When the high dimensional dataset is stored into the storage nodes, the logical information is lost. For example, it is not clear which node holds the 2-D subsets that abide the following requirements:

$$\begin{aligned} 10 < Dimension[time] < 15 \\ 15 < Dimension[latitude] < 35 \\ \dots \end{aligned} \tag{1}$$

To make the upper layer's request aware of the underlying layout, we need to retain the information. First, it is necessary to know how the data are distributed in the storage nodes. For simplicity, we assume the data are striped across the nodes in a round-robin fashion. Then an equation can be derived to calculate such distribution information.

Taking an n-D record dataset as an example, there is a slowest changing dimension, which is usually an unlimited dimension, also known as the 'record dimension' [3]. The data are stored along this record dimension and striped across the storage nodes. The (n-1)-D subset can be seen as a unit of the n-D dataset. Suppose the size of such a subset is m, the stripe size is t, and the stripe count is l. In the first round of striping, the $node_0$ will contain the first subset, the second subset, and till the $\lfloor \frac{t}{m} \rfloor$ subset, which could be expressed as $\{1, 2, 3, .., \lfloor \frac{t}{m} \rfloor\}$. In the second round of striping, it is not difficult to have

$$\{\lfloor \frac{t \times l}{m} \rfloor + 1, \lfloor \frac{t \times l}{m} \rfloor + 2, \dots, \lfloor \frac{t \times (l+1)}{m} \rfloor\} \tag{2}$$

Finally, all the (n-1)-D subsets on $node_k$ can be expressed using one union function,

$$\begin{aligned} f(Node_k) = \bigcup_{i=0,j=1}^{I,J} (\lfloor \frac{t \times (l+i)}{m} \rfloor + j + k \times \lfloor \frac{t}{m} \rfloor) \\ , (I = \frac{t \times l}{m}, J = \frac{t}{m}) \end{aligned} \tag{3}$$

Equation 3 is named as a *Subset Correlation Function*.

Using the *Subset Correlation Function* (3), we can determine which (n-1)-D subset is stored in a certain node. The input of the function (3) only includes the subset size m, the stripe size t, as well as the stripe count l, which makes it easy to calculate. However, this function has multiple 'floor' operations, which indicates that there will be plenty of subsets that stripe across two or more consecutive nodes, especially when the stripe unit is less than the (n-1)-D subset unit. To know the exact starting and ending positions of the data on each node, we further derive the *Sub-correlation Set* (short of subset correlation set) using this function. The *Sub-correlation Set* contains the start and length of each stripe on the node. From equation (3), we know the first and the last subset within each stripe. For example, the first stripe on the first node is bounded by the range [subset 1 : subset $\lfloor \frac{t}{m} \rfloor$]. Using such information, it is not difficult to calculate the exact array size in terms of 'start' and 'length'. For instance, the array's starts in every dimension are all 0s, and the length on each dimension is the maximum length for subset 1. For subset $\lfloor \frac{t}{m} \rfloor$, there is some remaining data that belongs to the $(\lfloor \frac{t}{m} \rfloor + 1)$ subset. Therefore, we calculate the size of such remaining data as

$$s = t \bmod m \tag{4}$$

The length on each dimension of subset $\lfloor \frac{t}{m} \rfloor$ is,

$$length_{n-1} = s \bmod \Pi_{i=0}^{n-2} d_i, \quad (5)$$

where d_i is the maximum length of i_{th} dimension.

Knowing the array start and dimension length of the subsets within one stripe, the exact start and ending position can be determined. Other stripes' boundaries in terms of start and length on each dimension can also be calculated. Thus we can derive the *Sub-correlation Set*, as shown in the equation (6),

$$stripe_i = (start, length) \quad (6)$$

where $start$ and $length$ are arrays of size n, which can specify the start and length on each dimension.

We will discuss how to perform a high-level aggregation over multiple sub-arrays requests using the *Sub-correlation Set* in the following subsection.

B. High-level Aggregation Algorithm

input :

n:	dataset dimension	m:	n-1 subset size
t:	stripe size	l:	stripe count
k:	the k-th node	M:	dataset size
K:	number of nodes	sub:	sub-array requests

output: Aggregated sub-array request

Prior Step: calculate sub-correlation set, one time analysis.

for $k \leftarrow 0$ **to** K **do**
  //calculate sub-correlation set, one time analysis.
  $Node_k = f(k)$; //using (3).
  for $i \leftarrow 0$ **to** I **do**
    $start = Node_k(i, j = 0)$;
    $length = length_{n-1}$ //using (5).
    $stripe_i = (start, length)$;
  end
end

Step One: decompose the sub-array requests, repeat the procedure till no outstanding requests left.

for $i \leftarrow 0$ **to** M/t **do**
  /*(M/t) is the number of strips.*/
  find(x)={x | node x contains $stripe_i$};
  while *(sub)* **do**
    if *($sub \cap stripe_i \neq \emptyset$)* **then**
      $pair[j++] =< x, sub \cap stripe_i >$;
      $sub = sub - sub \cap stripe_i$;
    end
  end
end

Step Two: aggregate the decomposed sub-arrays.

sort($pair$) by key;
/*merge pairs with same key*/
array_merge($pair$);

Algorithm 1: High-level Aggregation Algorithm

As we mentioned in section III.C, it is needed to determine how to decompose the incoming requests and perform an aggregation with the consideration of physical layout. We have introduced the sub-correlation set as an initial step to construct the mapping relationship between the physical layout and the logical input. In this subsection, we present the High-level Aggregation algorithm (HiLa) 1 for the proposed high-level aggregation approach. This algorithm includes one prior step and two major steps. The prior step is a one-time analysis for a specific system and dataset, where the sub-correlation set using (3) and (6) is calculated. The Step One shows how to decompose the requests. The request is compared with each sub-correlation set through an intersection operation. Note that both the request and the sub-correlation set represent a multi-dimensional subset. So the intersection operations is performed in a multi-dimensional way. If there is overlap between the request and the sub-correlation set, then this overlap is recorded as a value and the physical location as a key. This procedure is repeated until there are no outstanding requests. In the second step, we aggregate the decomposed requests that reside on the same node.

After the requests are decomposed and reformed, the aggregated requests are sent to the existing I/O methods, i.e., independent or collective I/O, to be carried out.

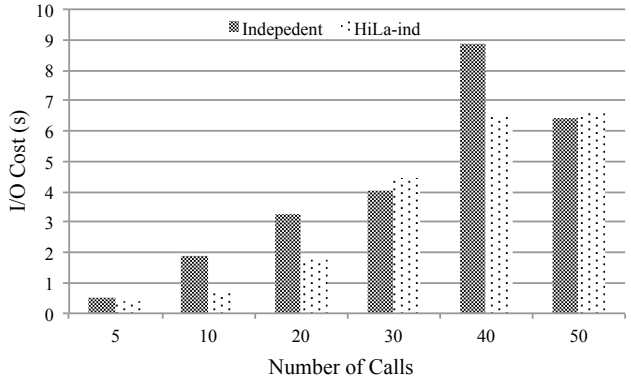
V. Experiments and Analyses

A. Experiments Setup

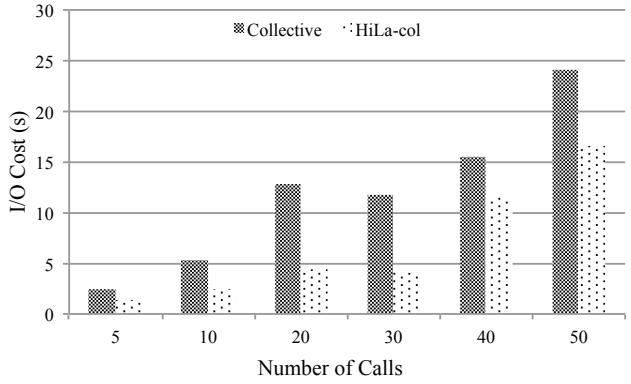
We have conducted tests on a 640-node Linux cluster. Each node in the cluster contains two Intel Xeon 2.8 GHz 6-core processors with 24 GB of memory. The nodes are connected with DDR Infiniband. Overall the cluster has a peak rating of 86 teraflops and a High Performance Linpack (HPL) score of 68 teraflops. The dataset we used is a synthetic NetCDF dataset, which is about 3.8GBs and includes four dimensions, i.e., time, level, latitude and longitude. First, we show an initial performance comparison with previous methods. We then conduct evaluations via various settings. Finally, we show the potential of the new approach in big data analysis using a query system.

B. Performance Improvement and Results Analysis

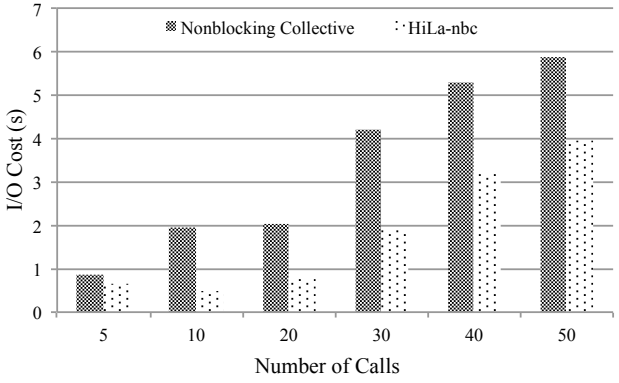
We have conducted several groups of tests. The first test deployed the same configuration as that in Figure 3. We compared the I/O cost in Figure 5, in which the 'HiLa-ind' refers to the independent I/O with High-level Aggregation, 'HiLa-col' refers to the collective I/O with High-level Aggregation, and 'HiLa-nbc' is the nonblocking collective I/O integrated with the proposed HiLa approach. We can observe a clear performance improvement comparing to the previous I/O strategies. The HiLa approach improved all the three access strategies, in which independent I/O was improved by up to 62%, the two-phase collective I/O was improved by 25%-65%, and a 47% average improvement for nonblocking collective I/O. This result shows that the HiLa approach can dramatically reduce the overlap and contention among concurrent sub-array



(a)



(b)

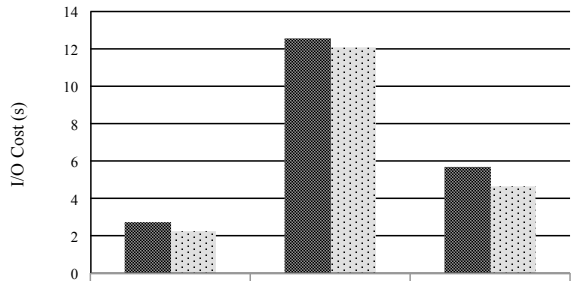


(c)

Fig. 5: Performance Improvement with HiLa

calls. When the overlapping is eliminated, the requests are decomposed and re-aggregated based on the sub-correlation set. The aggregated requests in the high level are not only logically close but also physically close, therefore when passing through the underlying I/O libraries, the locality is largely improved. These tests confirm the potential of the proposed approach. As a general high-level optimization, the current existing I/O methods can be further improved for big data analysis. The tests reported in Figure 3 and Figure 5 were both conducted with same configuration. We also evaluated the new approach and algorithm by varying stripe sizes.

In previous evaluations, we set a relatively large stripe size, i.e., 100MB. We have also varied the stripe size as 10MB, and we would like to observe how HiLa algorithm performs under smaller stripe size. As shown in Figure 6, we plotted results of three I/O methods, and we compare their traditional pattern with the HiLa-integrated pattern. In each test, the number of calls was 50 and the number of processes within each call was also 50. The stripe count (number of storage nodes) remained 40. We can see that the independent I/O performed well in such smaller stripe size, which was due to the smaller locking overhead. We also found that the HiLa approach improved the three I/O methods about 3%-18%. This observation confirms the proposed HiLa approach considers the underlying data organization well. When the striping manner is changed, the HiLa algorithm can constantly reveal the physical layout through its 'sub-correlation set'.



	Indepedent	Collective	Nonblocking Collective
Traditional	2.769361	12.567792	5.693901
HiLa	2.262536	12.118085	4.613422

Fig. 6: HiLa with Smaller Stripe Size

C. Performance Improvement for Query in FASM

The HiLa is also integrated as a component of the Fast Analysis with Statistical Metadata (FASM) system [14] that was developed in our previous work. The FASM system integrates a small amount of statistical metadata into the original datasets to speedup the data analysis. Compared to the original datasets, the FASM provides datasets with rich metadata, which has the pre-calculated statistical knowledge. With such statistics, the system can first filter the useless subsets that has a range beyond the query range, effectively improving the query performance.

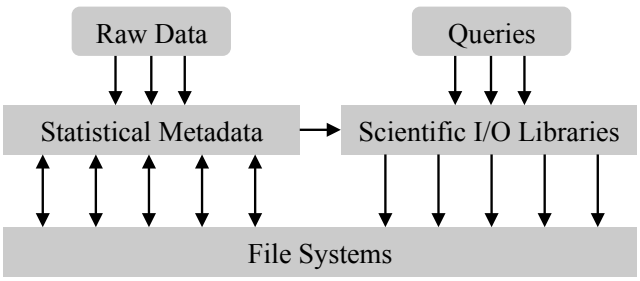


Fig. 7: Simplified FASM Architecture

We illustrate a simplified FASM architecture in Figure 7. The FASM is designed for Write Only Read Many (WORM) applications. The raw data and statistical metadata are stored together in the file systems, as shown in the left part of Figure 7. The statistical metadata will be read out and passed to a high-level I/O library, such as PnetCDF. The filtered I/Os are then issued to the file systems. It has been proven through experiments that the FASM is a promising technique to speedup the scientific data analysis. However, in our previous work, we found that when there are multiple subset queries, the performance is poor. Even though the amount of scanning is reduced after filtering useless subsets, the overlap and competition among different queries is unavoidable. The FASM accepts subset queries from the users and filters out the useless subsets using the integrated statistics, therefore reducing the number of hit subsets. In the case of multiple concurrent queries, the system will first filter each query separately. By integrating the HiLa algorithm, the system further reduces the overlapping across all of the concurrent queries. As shown in Figure 8, the FASM accepts multiple queries at the same time, the performance improvement using the HiLa approach achieved a 20%-55% speedup.

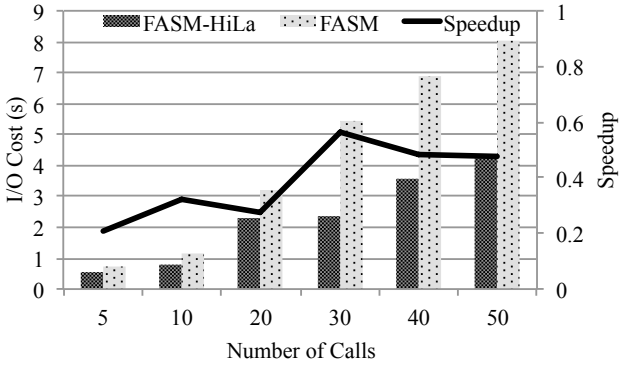


Fig. 8: FASM with HiLa

VI. Related Work

Big data analysis has been a critical research topic recently; the issues related to big data are usually about how to manage the large volume of data and provide an insightful understanding of the data. In scientific applications, research efforts are still continuing to improve the I/O performance. We discuss most of the related work in the following subsections.

A. Scientific I/O Libraries

PnetCDF is designed to provide high performance I/O for serial NetCDF [4], it has been commonly used in scientific application. Our current experiments are also conducted using PnetCDF liabrary. The Adaptable IO System (ADIOS) is an easy-to-use, fast, scalable, and portable I/O library, which provides a flexible and simple way for scientists to describe the datasets [1, 6]. HDF5 (Hierarchical Data Format) is also a widely used library by many scientific applications [8]. In [12], the ISABELA compression algorithm is applied into HDF5 to store the dataset. All these work have shown the importance of scientific datasets and data management libraries. In this study, we designed a high-level aggregation algorithm to further optimize the I/O performance. Our study complements these existing scientific libraries and can be advantageous to many data-intensive scientific applications.

B. Data Organization

Many parallel and distributed file systems have been developed in the past. In general, these file systems provide storage backbones to applications and can deliver high-performance accesses for well-formed and large I/O requests. Recently, several research efforts have started investigations on designing an even better file system that considers scientific dataset organization and applications' access characteristics [15, 16, 17]. There also exist studies applying space filling curves in the data placement. The Elastic Data Organization (EDO) was proposed to address the issue of accessing slow dimensions of datasets via providing various data organization schemes [19]. All these work demonstrates the importance of data organization, while our work designed a mapping scheme that can reveal the underlying data organization to serve the high-level I/O aggregation.

VII. Conclusion and Future Work

In processing the highly structured scientific dataset, most scientific I/O libraries leverage MPI-IO to facilitate the concurrent data retrieval/store. However, the inconsistency between logical access and physical layout can lead to poor performance, as observed in our experimental tests. In the case of concurrent I/O requests, the overlap among requests results in high contention on storage nodes.

In this work, we propose a locality-driven high-level aggregation approach (HiLa) to facilitating the existing I/O methods. The HiLa decomposes the multiple I/O requests and aggregate with the consideration of locality. The aggregation is performed on the logical level, which means no data movement overhead is caused. When combining with existing I/O strategies, HiLa demonstrates increased performance. Currently, the HiLa was evaluated with read operations and has demonstrated its potential. In the future, we will evaluate the approach with write operations. As the data volume keeps increasing in scientific applications and simulations, high-level I/O optimization like the proposed HiLa approach is imperatively needed and important. It can have a profound impact on big data management and analysis.

Acknowledgment

This research is sponsored in part by the National Science Foundation under grant CNS-1162488 and the Texas Tech University startup grant. 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] ADIOS. http://www.olcf.ornl.gov/center-projects/adios/.

[2] HDF5. http://www.hdfgroup.org/HDF5/doc/index.html.

[3] NetCDF. http://www.unidata.ucar.edu/software/netcdf/.

[4] Parallel NetCDF. www.mcs.anl.gov/parallel-netcdf.

[5] The Global Cloud Resolving Model (GCRM) project. http://kiwi.atmos.colostate.edu/gcrm/.

[6] H. Abbasi, J. F. Lofstead, F. Zheng, K. Schwan, M. Wolf, and S. Klasky. Extending I/O through high performance data services. In *CLUSTER*, pages 1–10. IEEE, 2009.

[7] J. Chou, K. Wu, and Prabhat. Fastquery: A general indexing and querying system for scientific data. In J. B. Cushing, J. C. French, and S. Bowers, editors, *SSDBM*, volume 6809 of *Lecture Notes in Computer Science*, pages 573–574. Springer, 2011.

[8] M. Folk, G. Heber, Q. Koziol, E. Pourmal, and D. Robinson. An overview of the HDF5 technology suite and its applications. In P. Baumann, B. Howe, K. Orsborn, and S. Stefanova, editors, *EDBT/ICDT Array Databases Workshop*, pages 36–47. ACM, 2011.

[9] 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.

[10] W. Gropp, E. Lusk, and R. Thakur. *Using MPI-2: Advanced Features of the Message Passing Interface*. Scientific and engineering computation. MIT Press, pub-MIT:adr, 2000.

[11] T. Hoefler, A. Lumsdaine, and W. Rehm. Implementation and performance analysis of non-blocking collective operations for MPI. In *SC'07 USB Key*. ACM/IEEE, Reno, NV, Nov. 2007.

[12] 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 S. Lathrop, J. Costa, and W. Kramer, editors, *SC*, page 31. ACM, 2011.

[13] J. Li, W. keng Liao, A. Choudhary, R. Ross, R. Thakur, W. Gropp, R. Latham, A. Siegel, B. Gallagher, and M. Zingale. Parallel netCDF: A high-performance scientific I/O interface. In ACM, editor, *SC2003: Igniting Innovation. Phoenix, AZ, November 15–21, 2003*, pub-ACM:adr and pub-IEEE:adr, 2003. ACM Press and IEEE Computer Society Press.

[14] 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.

[15] 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 A. B. Maccabe and D. Thain, editors, *HPDC*, pages 49–60. ACM, 2011.

[16] J. Soumagne, J. Biddiscombe, and A. Esnard. Data redistribution using one-sided transfers to in-memory HDF5 files. 2011.

[17] M. Stonebraker, J. Becla, D. J. DeWitt, K.-T. Lim, D. Maier, O. Ratzesberger, and S. B. Zdonik. Requirements for science data bases and sciDB. In *CIDR*. www.cidrdb.org, 2009.

[18] 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.

[19] 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.