Enabling Active Storage on Parallel I/O Software Stacks

Seung Woo Son* Samuel Lang* Philip Carns* Robert Ross* Rajeev Thakur*
Berkin Ozisikyilmaz† Prabhat Kumar† Wei-Keng Liao† Alok Choudhary†
*Mathematics and Computer Science Division
Argonne National Laboratory
Email: {sson,slang,carns,rross,thakur}@mcs.anl.gov
†Department of Electrical Engineering and Computer Science
Northwestern University
Email: {boz283,pku649,wkliao,choudhar}@ece.northwestern.edu

***Abstract*—As data sizes continue to increase, the concept of active storage is well fitted for many data analysis kernels. Nevertheless, while this concept has been investigated and deployed in a number of forms, enabling it from the parallel I/O software stack has been largely unexplored. In this paper, we propose and evaluate an active storage system that allows data analysis, mining, and statistical operations to be executed from within a parallel I/O interface. In our proposed scheme, common analysis kernels are embedded in parallel file systems. We expose the semantics of these kernels to parallel file systems through an enhanced runtime interface so that execution of embedded kernels is possible on the server. In order to allow complete server-side operations without file format or layout manipulation, our scheme adjusts the file I/O buffer to the computational unit boundary on the fly. Our scheme also uses server-side collective communication primitives for reduction and aggregation using interserver communication. We have implemented a prototype of our active storage system and demonstrate its benefits using four data analysis benchmarks. Our experimental results show that our proposed system improves the overall performance of all four benchmarks by 50.9% on average and that the compute-intensive portion of the k-means clustering kernel can be improved by 58.4% through GPU offloading when executed with a larger computational load. We also show that our scheme consistently outperforms the traditional storage model with a wide variety of input dataset sizes, number of nodes, and computational loads.**

I. INTRODUCTION

Many of the important computational applications in science and engineering, sensor processing, and other disciplines have grown in complexity, scale, and the data set sizes that they produce, manipulate, and consume. Most of these applications have a data intensive component. For example, in applications such as climate modeling, combustion, and astrophysics simulations, data set sizes range between 100 TB and 10 PB, and the required compute performance is 100+ teraops [10], [11]. In other areas, data volumes grow as a result of aggregation. For example, proteomics and genomics applications may take several thousand 10-megabyte mass spectra of cell contents at thousands of time points and conditions, yielding a data rate of terabytes per day [48].

Scientists and engineers require techniques, tools, and infrastructure to better understand this huge amount of data, in particular to effectively perform complex data analysis, statistical analysis, and knowledge discovery. As an example, scientists have been able to scale simulations to more than tens of thousands of processors (and for more than a million CPU-hours), but efficient I/O has been a problem. Indeed, in some cases, once the simulation is finished in a couple of days, several months are needed to organize, analyze, and understand the data [26], [41]. From a productivity perspective, this is a staggering number.

The performance of data analysis tasks that heavily rely on parallel file systems to perform their I/O is typically poor, mainly because of the cost of data transfer between the nodes that store the data and the nodes that perform the analysis. For applications that filter a huge amount of input data, the idea of an active storage system [1], [25], [38], [40] has been proposed to reduce the bandwidth requirement by moving the computation closer to the storage nodes. The concept of active storage is well suited for data-intensive applications; however, several limitations remain, especially in the context of parallel file systems. First, scientists and engineers use a variety of data analysis kernels including simple statistical operations, string pattern matching, visualization, and data-mining kernels. Current parallel file systems lack a proper interface to utilize these various analysis kernels embedded in the storage side, thereby preventing wide deployment of active storage systems. Second, files in parallel file systems are typically striped across multiple servers and are often not perfectly aligned with respect to computational unit, making it difficult to process data locally in the general case. Third, most analysis tasks need to be able to broadcast and/or reduce the locally (partially) read or computed data with the data from other nodes. Unfortunately, current parallel file systems lack server-side communication primitives for aggregation and reduction.

The main contribution of this paper is an active storage system on parallel I/O software stacks. Our active storage system enables data analytic tasks within the context of parallel file systems through three key features:

- **Enhanced runtime interface that uses predefined kernels in parallel file systems:** We *expose* the semantics of predefined analysis kernels, such as the data type of data blocks on the disk, to parallel file systems so that execution of embedded kernels is possible on the server.

978-1-4244-7153-9/10/$26.00 © 2010 IEEE

- **Stripe alignment during runtime:** In order to allow a file server to perform proper computation on striped files, our system adjusts to misaligned computational units by pulling missing bytes, when needed, from the neighboring servers that hold them.
- **Server-to-server communication for aggregation and reduction:** In order to perform computation entirely on the server side, servers need to communicate their partial (local) results with other servers to obtain the complete results. To this end, we augmented the storage servers with basic collective communication primitives (e.g., broadcast and allreduce).

To demonstrate the effectiveness of our approach, we built an active storage prototype on top of a parallel file system, PVFS, and parallel runtime system library, MPICH2. Our proposed system demonstrates a better way of enabling an active storage system to carry out data analytic computations as part of I/O operations. The experimental results obtained with a set of data analytic kernels, including data-mining kernels, demonstrate that our prototype can improve the overall performance by 50.9%, on average. We show that the compute-intensive kernels of the k-means clustering algorithm can be offloaded to a GPU accelerator, leading to 58.4% performance improvement when the algorithm became compute-intensive. Overall, we show that our approach consistently outperforms the traditional storage model with a wide variety of data set sizes, number of nodes, and computational loads.

The remainder of this paper is organized as follows. Section II presents background on active storage systems and their limitations in the context of parallel file systems. Section III introduces our active storage system with analytic capability and discusses the technical details of our approach. Our experimental framework is described in Section IV. Section V presents the results collected from our active storage prototype built atop a parallel file system and a parallel runtime library. Section VI discusses related work, and Section VII summarizes our main findings and discusses possible future work.

II. Active Storage in Parallel File Systems

Our work is based on the active storage concept. In this section, we discuss the implications of active storage in the context of parallel file systems.

In a traditional storage model, shown in Figure 1(a), the analytic calculation is performed on the client. Therefore, all the data residing on the storage server needs to be transferred to the client. While this execution model offers the potential for scalability within the number of client nodes available, it suffers from poor performance when applications are data-intensive and frequently perform data filtering; that is, the output data size is much smaller than the input data size. To illustrate how this data filtering affects the application performance, let us consider an application that reads a three-dimensional array of size $N \times N \times N$ as input data and generates only a one-dimensional array of size N as a result of data processing. Unless the analysis kernels incur a substantial computational load, most of the time will be spent in reading and transferring a total of N^3 input data to the client for processing. This excessive communication overhead can be reduced if we move the computation close to the file server and let the server perform data filtering. In this case, computation on the server side incurs transferring the filtered data of size N, which is only $1/N^2$ of the original data volume.

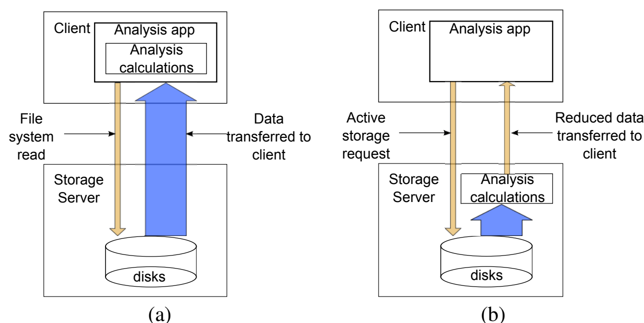

Fig. 1. Two models for performing data analysis: (a) traditional storage model, (b) active storage model. In (a), high-level data information, such as data structures and attributes types, are lost when data reaches lower layers, typically converted to just (striped) byte streams in parallel file systems.

Accordingly, several studies have proposed active storage[1] [17], [37], [38]. Unlike the traditional storage model, an active storage model, shown in Figure 1(b), executes analysis operations on the server, thereby reducing the communication cost. While prior studies have shown the potential benefits of such approaches, several challenges remain in order to realize the active storage concept in parallel file systems. First is the lack of proper interfaces between clients and servers. Existing parallel runtime libraries are designed mainly for providing highly concurrent access to the parallel file system with high bandwidth and low latency. Recently, storage vendors have been using more and more commodity architectures in building their high-end storage servers. Those systems are typically equipped with x86-based multicore processors with large main memory. For example, EMC's Symmetrix V-Max storage system includes 4 Quad-core 2.33 GHz Intel Xeon processors with up to 128 GB of memory [16]. Another important trend in cluster design is to use hybrid CPU-GPU clusters for high-performance computing that leverage the performance of accelerators. Existing parallel runtime interfaces, however, do not expose a way to utilize those extra computing capabilities on the server nodes.

Second, the data that needs to be analyzed, when stored in parallel file systems, is often not aligned perfectly when striped across multiple file servers. Data analysis kernels usually deal with multidimensional or multirecorded data. Each item or data member consists of multiple attributes or features that characterize and specify the individual data item in more detail. As an example, let us consider the data file in Figure 2, which is striped across three file servers. Let us assume that the data item (compute unit) in question consists of 10 attributes (variables), all stored in double-precision floating-

[1] The original concept, proposed circa 1998, utilizes the processing power on disk drives [1], [25], [40].

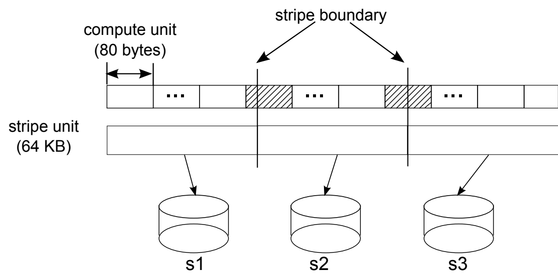

Fig. 2. Example illustrating how file stripes can be misaligned to the compute unit boundary.

point numbers. Each compute unit is 80 bytes long (10×8 bytes). Assuming the stripe unit is 64 KB (or 65,536 bytes), as shown in Figure 2, the original stripe unit contains only 819 variables (i.e., 65,520 bytes). The 820th variable actually spans both $s1$ and $s2$, 16 bytes in $s1$ and the remaining 64 bytes in $s2$. Another factor that exacerbates the alignment problem in parallel file systems is that files are typically stored in a self-describing manner; in other words, the file starts with its own header. Although headers are typically of fixed size, they are rarely aligned with stripe boundaries. Without properly handling these cases, servers will encounter partial data elements that cannot be processed. We note that data alignment also can be done on parallel file systems by manipulating file formats, such as stripe-aligned headers or use of footers instead of headers. These additional format changes, however, could incur huge I/O access time because a separate file open and rewrite is required for conversion.

Another feature that prevents conventional parallel file systems from implementing the active storage concept is the lack of collective communication primitives on the servers. Collective operations are used extensively in many parallel applications. Even in active storage architectures where the computations are performed on the server, we need to have collective primitives in order to enable entire server-side operations. In active storage architectures, the computation cannot be completed without aggregation because the result on each server is *partial*. In simple operations that involve a single pass of execution only, the result can be aggregated on the client side by combining partial results returned from the servers. More complex data analysis, however, requires several passes over the data to finish the computation. In this case, performing aggregation entirely on the server side makes more sense. In fact, storage servers have been using some forms of collective communication, but they are used for different purposes. For example, the use of server-to-server communication can significantly improve metadata operations (e.g., file creation or removal) in parallel file systems [8]. We use this capability to implement analysis routines that use collective operations for general-purpose communication.

III. Design of Active Storage System from Parallel I/O Interface

This section describes our active storage system for executing data analysis kernels in parallel file systems. We begin by presenting our target storage deployment model and an overview of our approach. We then discuss the three major components of our approach in more detail.

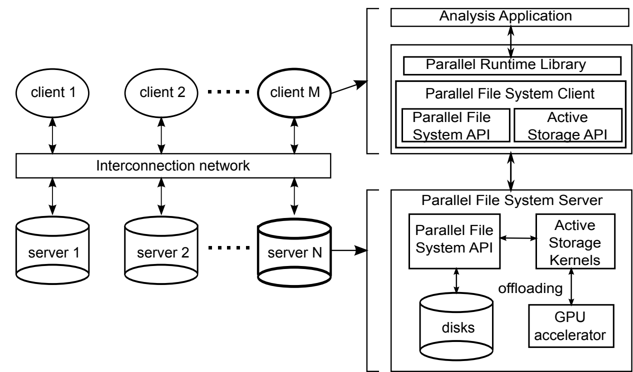

Fig. 3. Overview of our active storage system and our default storage deployment architecture. There are M clients connected to N active storage nodes.

A. Storage Deployment Architecture and Overview of Our Approach

Storage and compute nodes in a cluster can be deployed in two ways. The first, which is also our default storage deployment model as illustrated in Figure 3, is to locate storage nodes separately from compute nodes. This deployment architecture typically creates a pool of storage nodes to provide highly parallel I/O. It is widely used in high-performance computing clusters and cloud storage systems such as Amazon's S3 platform [2]. The second approach is to collocate storage and compute activities on the same cluster node. This deployment model is well suited for the MapReduce/Hadoop programming model, which schedules a compute task on a node that stores the input data being processed by that task [14], [22]. In this paper, we focus mainly on the separate storage deployment model because this is most common in HPC environment, but the impact of the overlapped deployment is discussed in the experimental evaluation section as well.

Figure 3 illustrates the high-level view of our active storage system in our storage deployment architecture. Our active storage approach utilizes a processing capability within the storage nodes in order to avoid large data transfers. Computational capabilities, including optional GPU accelerators, within the storage infrastructure are used to reduce the burden on computationally intensive kernels. The active storage nodes are located on the same communication network as the client (compute) nodes. On the client side, we use MPI for communication.

To provide easy use of our active storage system equipped with these analytic kernels, we enhanced the MPI-IO interface and functionality to both enable analytics and utilize active storage nodes for performing the data analysis. We chose MPI for our prototyping for two reasons. First, MPI is a widely used interface, especially in science and engineering applications, and numerous parallel applications are already written in MPI. Therefore, it would provide an easy migration path for those applications to effectively utilize our approach. Second, MPI provides a hint mechanism by which user-defined information

can be easily transferred to intermediate runtime libraries, thereby making incorporating data analysis kernels easier.

Our proposed approach works as follows. The analysis application on the client nodes uses normal MPI and MPI-IO calls to perform its I/O and computation/communication. For our active storage-based application, the client invokes our enhanced MPI-IO calls to initiate both data read and computation, and the corresponding functions and code are executed on the active storage nodes, which may use any available hardware acceleration functions. An advantage of this design is that it facilitates a runtime decision within the MPI-IO implementation on where to execute analysis – the functions could just read and then analyze on compute nodes, if the cost were lower. The results of active storage operations then are returned to the client in the original file read buffer. We note that higher-level libraries (e.g., Parallel NetCDF [29]) can be easily extended to use this infrastructure. Currently, users are required to develop their own functions (e.g., analytics, mining, statistical, subsetting, and search), develop their own accelerators using the GPUs, and embed them with our active storage APIs so that they can be called from their applications.

In the following sections we describe more details of the proposed approach, which comprises three major components: an enhanced runtime interface utilizing predefined analysis kernels, handling of striped files for proper computation on the server, and server-to-server communication for reduction and aggregation.

B. Enhanced Runtime Interface in Parallel File Systems

The predefined kernels are chosen from common analytics and mining applications and embedded in the storage server. There are two important aspects regarding the selection of the kernels. First, the kernels should represent a large portion of the overall computational workload of each kernel's algorithm. Second, the selected kernels should be common to various algorithms, so that deploying one kernel will benefit multiple algorithms and applications. Each function developed for our active storage node consists of two versions: traditional C/C++ code and accelerator-specific code. The C/C++ codes are executed on the normal storage server, whereas the accelerator-specific code are executed on the accelerator if available and required based on performance considerations.

Since we use MPI-IO as our parallel runtime library, we can explore several options for exposing these new functions. One option is to use the hints mechanism in MPI-IO and define new hints (key/value pairs) that specify the data analysis function that gets executed on the server. Another option is to use the user-defined data representations in MPI-IO. The MPI-IO interface allows users to define their own data representation and provide data conversion functions. MPI will internally call those functions when reading the data. While this method supports both predefined and user-defined operations, it is restricted to MPI-based environments; hence, server codes need to be rewritten in MPI or be interfaced with MPI-based analysis functions. A third option is to define extended

```
sum = 0.0;
MPI_File_open(..., &fh);
double *tmp = (double *)malloc(n*sizeof(double));
offset = rank*nitem*type_size;
MPI_File_read_at(fh, offset, tmp, n,
                 MPI_DOUBLE, &status);

for (i=0; i < n; i++)
    sum += tmp[i];
```

(a) Conventional MPI-based

```
MPI_Info info;
MPI_Info_create(&info);
MPI_Info_set(info, "USE_Accelerator", "TRUE");
MPI_File_open(..., info, &fh);
MPIX_File_read_ex(..., MPI_DOUBLE, SUM, &status);
```

(b) Active storage-based: client code

```
int cnt = sizeof(pipeline_buffer)/sizeof(double);
/* typecasting from char * to double * */
double *buf = (double *)pipeline_buffer;
for (i=0; i < cnt; i++)
    sum += buf[i];
```

(c) Active storage-based: embedded kernel

Fig. 4. Example of two execution models: (a) conventional MPI based, (b) and (c) active storage execution model. In (c), the original I/O buffer (`pipeline_buffer`) points to the byte stream read from the disk. Therefore, it first needs to be typecasted to the proper data type, `double` in this example.

versions of MPI-IO read calls that take an additional argument specifying the operation to be performed. In this paper, we explore the combination of both the extended version of MPI read calls and the hints mechanism. While the former provides a way to specify the operation to be called on the server, the latter allows us to pass application-specific semantics that are required to complete the specified kernels.

The syntax for our modified version of file I/O call, MPIX_File_read_ex()[2], is a simple extension of the existing MPI_File_read call. Specifically, the new API takes all the arguments in the original API and an additional argument that specifies the operations that need to be performed on the server. Figure 4 shows an example of two execution scenarios: conventional MPI-based and our active storage-based. Because of space limitations, the code fragment shown in Figure 4 focuses mainly on the major functionality required to implement common data analysis applications. As shown in Figure 4(a), typical data-intensive analysis applications begin by reading the target data file from the file systems, followed by executing the necessary data transformations and calculating the sum of all the data elements.

The data type and the operation to be executed are sufficient for an embedded kernel to perform simple reduction or statistical calculation such as SUM or MAX. But in some cases, the embedded kernel might need more information. For example, application writers might want to use the accelerator attached to the server, instead of the general-purpose CPU. As another example, if we run a string pattern-matching kernel, then the kernel also needs to know the input search pattern specified by

[2]We want to make it clear that the enhanced APIs are not part of the MPI specification.

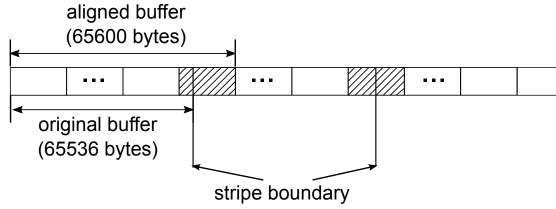

Fig. 5. Example illustrating how our proposed data alignment works. The original buffer is changed to the aligned buffer using `realloc()`, and the content of the buffer is then adjusted using a new offset and length. The missing data region is transferred from the i+1th server within the striping factor.

users. To provide such flexibility in the embedded kernel, we use MPI-IO's hints mechanism. Figure 4(b) gives an example use of the hint that notifies the kernel to use the accelerator.

C. Stripe Alignment during Runtime

As discussed previously, striped files are often not perfectly aligned along the physical stripe boundary. To address this problem, we developed a framework to identify the offset (O) and size length (L) of missing bytes and pull those data from the server that holds them. Our proposed data alignment in the I/O buffer is depicted in Figure 5. As shown in the figure, when each server receives a file read request, it reads the data elements within the stripe boundary and adjusts the size of the request to account for records with only a portion stored locally. To identify whether the data element is aligned to the computational unit, we use $\{O, L\}$ pairs initially passed to the file server. To calculate the location of missing bytes on other file servers, we also expose to each server the file distribution information, such as the number of nodes used for striping and the default stripe unit. The pseudo code for locating a missing data region is provided in Algorithm 1. For simplicity of calculation, the ith server always makes a request to the $i+1$th server within the striping factor, and in our prototype we currently assume data elements do not span more than two servers. We note that Hadoop/HDFS [22] and Lustre library-based active storage [38] also use a runtime stripe alignment to handle stripe boundaries correctly. However, their alignment task is actually performed in the client program that requests file reads because they assume collocated client and server nodes within the same node pool.

To illustrate how the embedded kernels are executed along with data alignment in the I/O buffer, we depict our enhanced I/O pipeline in Figure 6. In PVFS, multistep operations on both the client and server are implemented by using state machines. A state machine is composed of a set of functions to be executed, along with an execution dependency defining the order of execution [6]. The state machine shown in Figure 6 is executed whenever a server receives a read request call. In normal file read operations, it first reads (fetches) the data from the disks and then sends (dispatches) the read data to the clients. In active storage operations, on the other hand, after reading the data, it first checks whether the read buffer is aligned with respect to the computational unit. If it is "MISALIGNED" (see Figure 6), it communicates with the corresponding server and obtains the missing regions.

Algorithm 1 Pseudo code for locating missing bytes and calculating the buffer adjustment. This alignment is performed in each server whenever the data read locally is not perfectly aligned to the computational unit.

Input: (O_i, L_i) pair for the ith server,
where O_i is the offset of the block in the file, and
L_i is the size length of the requested block.
CU is the size of computation unit in bytes.
N is the number of server nodes.
Output: (O_{i+1}, L_{i+1}) pair for the neighboring server, and
(O'_i, L'_{i+1}) pair for the ith server itself.

1: get `stripe_unit` in bytes;
2: **if** (`stripe_unit` % CU) $\neq 0$ **then**
3: `misaligned_size` = `stripe_unit` % CU;
4: L'_i = ⌈ `stripe_size`/CU ⌉ × CU;
5: O_{i+1} = O_i + `stripe_size`;
6: L_{i+1} = CU − `misaligned_size`;
7: **if** $i \neq 0$ **then**
8: O'_i = O_i + `misaligned_size`;
9: **end if**
10: resize the original `buf_size` to `new_buf_size` starting at O'_i;
11: request the $i+1$th server to pull data at (O_{i+1}, L_{i+1});
12: **emit** "MISALIGNED";
13: **else**
14: **emit** "ALIGNED";
15: **end if**

After this alignment, the file server performs the necessary operations on data elements and sends back the results to the client. The pipelining process continues until all data has been read and the results have been transferred. We note that, depending on how a data analysis kernel operates (e.g., a single-pass execution or multiple passes of execution that require aggregation across all servers), the server either sends the result back to the client on every I/O pipelining or accumulates the results across all servers and returns the final result.

D. Server-to-Server Communication Primitives for Reduction and Aggregation

Collective communication has been studied extensively, and MPI collective operations are now universally adopted [21]. Basic interserver communication already exists in PVFS to handle various server operations such as small I/O and metadata operation. Using the basic server-to-server communication primitives in PVFS, we implemented two collective primitives: broadcast and allreduce. We chose these two mainly because they are used in one of our benchmark routines, KMEANS. While several viable algorithms exist, our implementation uses the *recursive distance doubling* algorithm [20] shown in Figure 7 for both collective operations. These two operations each are built as separate state machines in PVFS, so any embedded kernel that needs to use collective operations can use these facilities without having to reimplement them.

Figure 8 shows the k-means clustering kernel we implemented using two collective primitives. The algorithm has several variants, but in general it is composed of multiphased operations: cyclic data updates and potentially large computation until convergence. When the algorithm is parallelized, the compute-intensive portion, which is calculating minimum

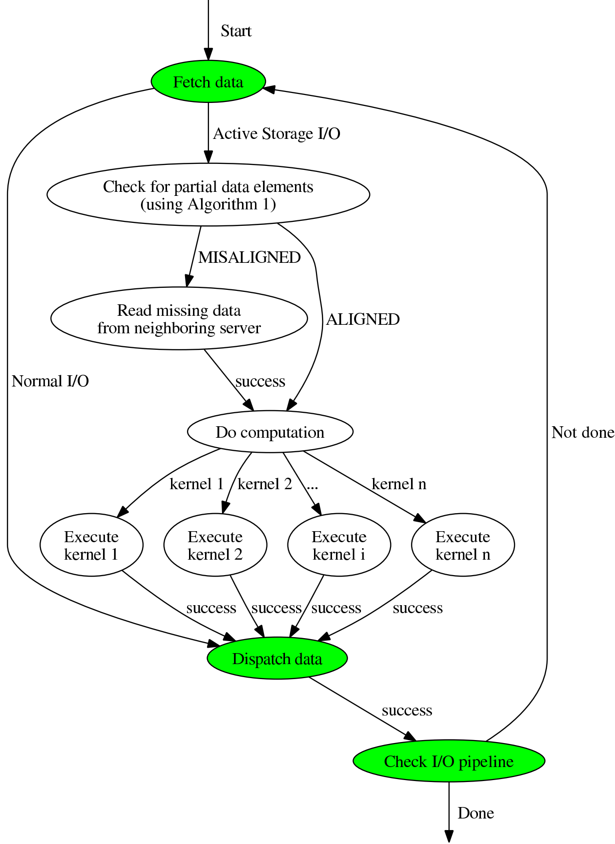

Fig. 6. I/O pipelining depicting how stripe alignment and execution of embedded kernels are combined into normal I/O pipelining flow. In the original (normal) file system operations, only the ovals in green are executed. In the PVFS state machine infrastructure, each oval represents a function to be executed, and directed edges between them denote the execution dependencies.

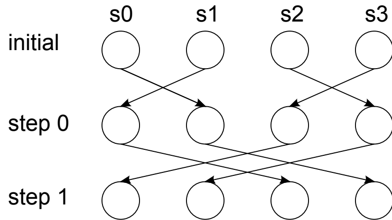

Fig. 7. Recursive distance doubling algorithm we used to implement collective primitives. The basic message passing interface is implemented by using the *message pair array* state machine [6].

distance, can be executed in parallel. But a reduce operation must be used as new cluster centers obtained in each server need to be updated. We note that, while we implemented these two collective primitives mainly to ease embedding more complicated data analysis kernels, we believe that they can be used for other purposes whenever server tasks can benefit by being cast in terms of collective operations.

IV. Experimental Framework

This section describes our experimental platform and the schemes and benchmarks we used in our experiments.

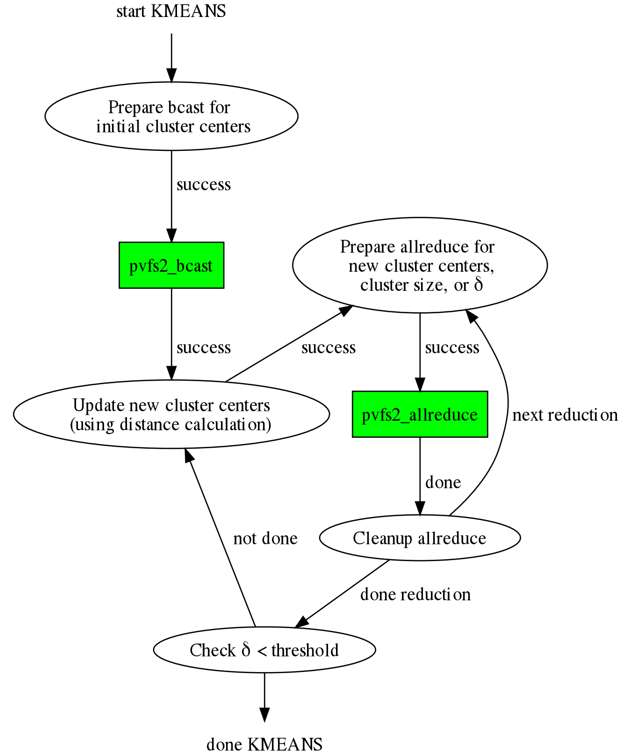

Fig. 8. The k-means clustering kernel implemented within the PVFS server. It uses two collective primitives, `pvfs_bcast` and `pvfs2_allreduce`. The primitive `pvfs_bcast` is used only once for broadcasting initial data centers chosen randomly, whereas `pvfs_allreduce` is used three times for updating new cluster centers, new cluster size, and new δ value.

TABLE I
Default System Configuration

Parameter	Value
Total # of nodes	32
# of client nodes	4
# of server nodes	4
Main CPU	Dual Intel Xeon Quad Core 2.66 GHz per node
Main memory	16 GB per node
Storage capacity	200 GB per node
Stripe unit	(file size)/(# of server nodes)
Interconnect	1 Gb Ethernet
Accelerator board	2 NVIDIA C1060 GPU cards

A. Setup

To demonstrate the benefit of our active storage model, we built a prototype of our active storage system in PVFS [7], an open source parallel file system developed and maintained by Argonne National Laboratory, Clemson University, and a host of other university and laboratory contributors. We also added an extended version of a file read API to the MPI-IO implementation [21]. This API allows us to pass an operator to the PVFS server along with application-specific semantics so that the servers can execute built-in data analysis kernels. We performed our experiments on a cluster consisting of 32 nodes, each of which is configured as a hybrid CPU/GPU system. We configured each node as either a client (running an MPI program) or a server (running a PVFS server), depending on our evaluation schemes (as will be explained in Section IV-B). The default system configuration of our experiment cluster is

TABLE II
CHARACTERISTICS OF DATA ANALYSIS KERNELS

Name	Description	Input Data	Base Results (sec)	% of Data Filtering
SUM	Simple statistical operation that reads the whole input file and returns the sum of the elements.	512 MB (2^{26} of double)	1.38	∼100%
GREP	Search matching string patterns from the input text file.	512 MB (2^{24} of 128 byte string)	1.49	∼100%
KMEANS	Parallel k-means clustering algorithm [28]. We used $k = 20$ and $\delta = 1.0$, where k is number of clusters, and δ is a threshold value.	40 MB (10^6 of 10 dimensional double)	0.44	90%
VREN	Parallel volume rendering written using MPI for both communication and collective I/O [36]. The algorithm consists mainly of I/O, rendering, and compositing.	104 MB (300^3 of float)	2.61	97%

given in Table I. All nodes run the Linux 2.6.27-9 kernel.

Our test platform uses a gigabit network, and one might argue that the benefit of reducing data transfer by active storage will be diminished when a better network, such as 10 GigE or Infiniband, is used. Potentially the use of a better network will improve the overall performance to some degree. However, several other factors, such as storage device performance and the number of software layers involved in I/O, also possibly affect the overall performance. Furthermore, in an active storage execution environment, the client nodes and the active storage nodes cooperatively execute various data analysis tasks. Data filtering on the storage nodes eliminates a massive amount of bandwidth consumption (as well as less contention for the I/O bandwidth), which is then available to better service other requests. Therefore, considering the ever-increasing amount of dataset to analyze, we expect that there will still be the potential of using active storage as compared to the traditional storage model, even with a somewhat higher performance network.

B. Evaluated Analysis Schemes

To demonstrate the effectiveness of our proposed active storage system, we tested three schemes:

- Traditional Storage (TS): In this scheme, the data is stored in a PVFS parallel file system, and the analysis kernels are executed on the client side.
- Active Storage (AS): This scheme is an implementation of our active storage model discussed in Section III. The client simply initiates the file read on the data stored on the active storage node. The operation triggered by the client is executed on the file servers, and only the processed result (data) is transferred to the client.
- Active Storage with GPU (AS+GPU): This scheme is the same as the AS scheme except that the core analytic kernels are executed on the GPU board attached to each server. The choice of this scheme versus AS is made by the application writer through the hints mechanism.

C. Benchmarks

To evaluate the effectiveness of our approach, we measured its impact on the performance of the four benchmarks shown in Table II. We chose these benchmarks because they represent kernels from various classes of data analysis and data-mining applications. For example, the SUM benchmark is one of the statistical operations typically used when the application needs to compute statistics on the data generated in each time step for each variable. The string pattern-matching kernel in GREP is used to search against massive bioinformatics data. VREN is a complete volume renderer for visualizing extremely large scientific data sets. KMEANS is a method of cluster analysis in data-mining tasks. Each benchmark is briefly described in the second column of Table II. The third column gives the default input data size used in each benchmark. The execution time using the default input data set and the default system configuration (shown in Table I) is given in the fourth column of Table II. The last column in Table II gives the percentage of data filtering between the input data set and output data. For example, KMEANS showed the least amount of data filtering, 90%, which is still high. The other three benchmarks (SUM, GREP, and VREN) filter most of their input data.

All benchmarks were written in MPI and MPI-IO [21], and each MPI-IO call invokes I/O operations to the underlying parallel file system, PVFS [7] in our case. We compiled all the benchmarks using mpicc (configured with gcc 4.2.4) with the -O2 optimization flag. The modified PVFS was also compiled by using the same compiler with the same optimization flag. Since the analysis kernels for the AS+GPU scheme are executed in the GPU accelerator, those kernels were written in the CUDA language and compiled by using nvcc 2.1 [32] with the same optimization flag.

We note that, in our evaluation, the ratio of server (file server) nodes to client (compute) nodes for TS is 1:1, and we maintain this ratio in all experiments. Therefore, when the TS scheme is executed using four client nodes, we also use the same number of server nodes. We use the ratio of 1:1 to demonstrate that it is not the bandwidth, but the location of the computation, that makes the difference between TS and AS, although few systems have such a high ratio. As computation is performed on the server side in the case of AS and AS+GPU, we use only one client node, which initiates the active storage operation, regardless of the number of server nodes used for execution.

V. EXPERIMENTAL RESULTS

We now present our experimental results. We begin our analysis by focusing on the impact of our scheme on overall execution time. We then look at the sensitivity analysis results, including the impact of the number of nodes, the data set size, and the effect of offloading compute-intensive kernels onto accelerators.

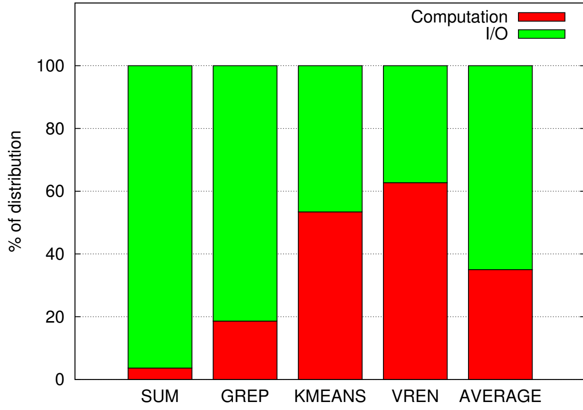

Fig. 9. Distribution of I/O and computation time fraction when the TS scheme is executed using four client nodes. We note that, because the TS scheme runs the client and server on separate nodes, the I/O time depicted here includes the disk access time on the file server and the communication time from the server to the client nodes.

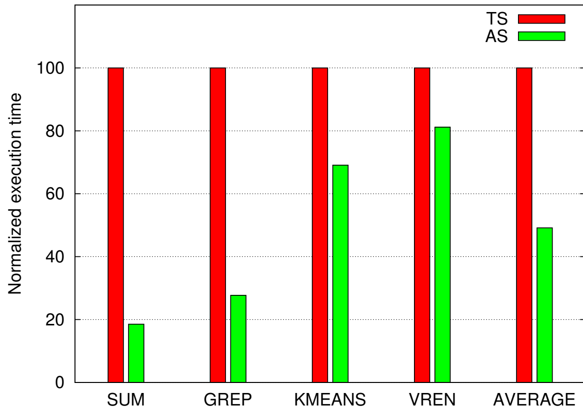

Fig. 10. Normalized execution time of the AS scheme with respect to the TS scheme. Since TS and AS use the same number of nodes to perform I/O and computation, that is, both incur the same amount of computation and disk access time, the improvement by AS is attributed to the reduction of data transfer traffic between the server and the client.

A. Performance Improvement

Before presenting the performance benefits of our approach, let us first look at how much time each benchmark spent on I/O, in other words, the I/O *intensiveness* exhibited by each benchmark. The graphs in Figure 9 present the percentage of I/O and computation time of each benchmark when the TS scheme (original version) executed using four client nodes (with four server nodes). Clearly, each benchmark showed a different amount of I/O time (ranging from 37.3% to 96.4%); but in general, they spent a significant amount of time in I/O, 64.4% on average over all benchmarks.

Figure 10 shows the normalized execution time with respect to the TS scheme for each benchmark. Again, both schemes are executed on four nodes. We see from these graphs that the AS scheme improves the performance significantly, 50.9% on average for all benchmarks. These huge improvements can be attributed to the reduction of data transfer from the server to

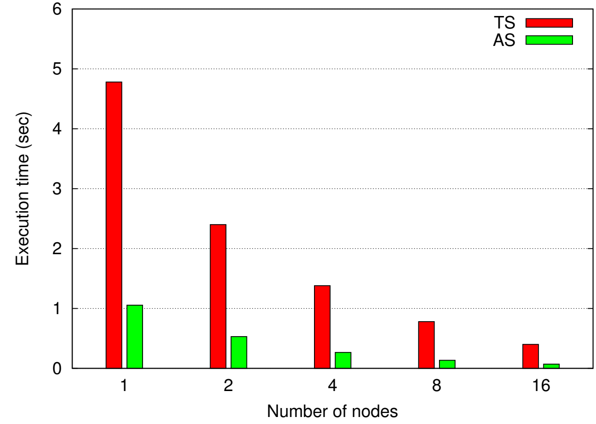

Fig. 11. Execution time for the SUM benchmark as we increase the number of nodes to execute. The input dataset is fixed at 512MB as in Table II.

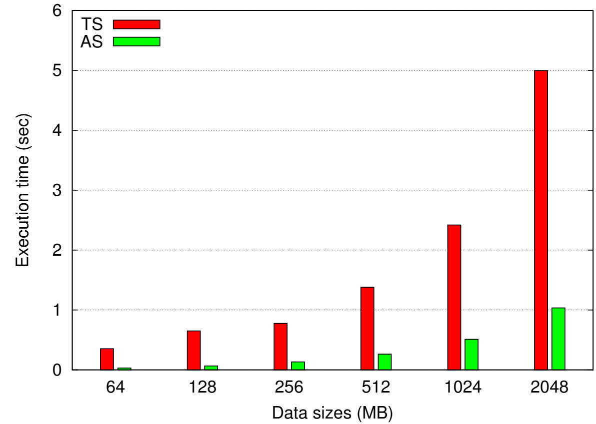

Fig. 12. Execution time for the SUM benchmark as we increase the size of input data set. We used four nodes to execute both the TS and AS schemes.

the client. In fact, these improvements are proportional to the percentage of I/O exhibited by each benchmark as shown in Figure 9. This result indicates that the more an application is I/O bound, the greater are the benefits of using the AS scheme.

B. Scalability Analysis

We now look at the performance when the number of nodes and data set size are varied. We first present the case where the number of nodes to execute is increased. Recall that in our default experimental setup, both the TS and AS schemes are executed by using four nodes. Figure 11 shows the execution time of SUM with different numbers of nodes. The input data set is fixed at 512 MB as in Table I. We present this scalability study using SUM because it filters most of input data so that we can show the maximum benefits of our approach attributable to the reduction in data transfer. Since our benchmarks show similar data filtering behavior (see the last column of Table II), we expect other benchmarks to show similar results. As can be seen in Figure 11, the performance of TS is scalable with a larger number of nodes. The reason is that, as we increase the number of client nodes, we also increase the number of server nodes in the same ratio. We see

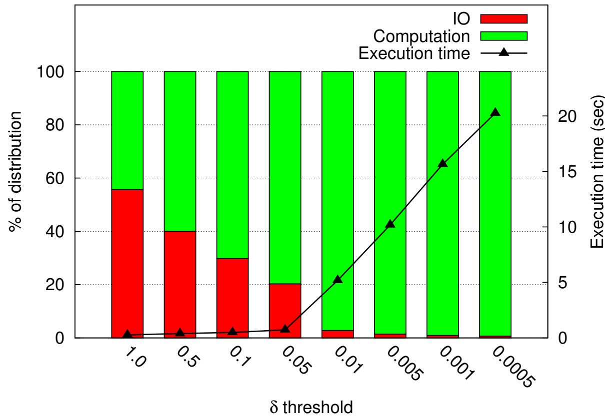

Fig. 13. Execution time and fraction of time in I/O and computation of KMEANS with different δ threshold values when the TS scheme is executed using four client nodes. All other experimental parameters are fixed as in Table I.

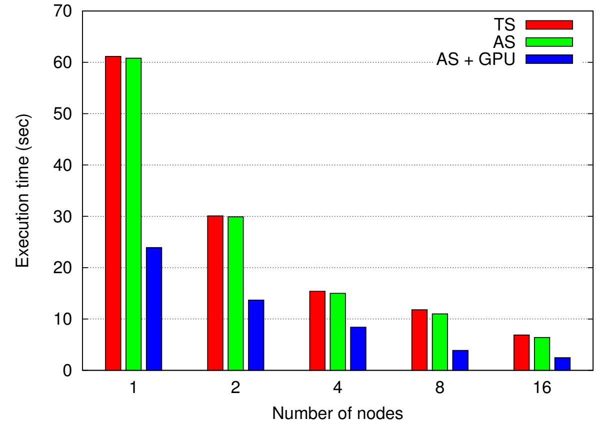

Fig. 14. Execution time for KMEANS with different numbers of nodes. The input data is 1000K data points, and δ is fixed at 0.001. We note that, when the δ value is 0.001, KMEANS is becoming fairly compute intensive (see Figure 13).

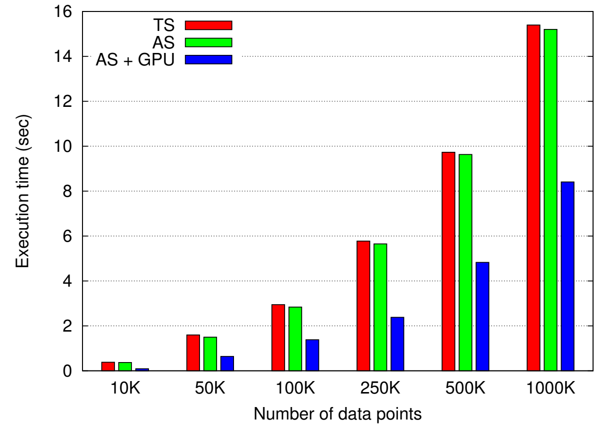

Fig. 15. Execution time for KMEANS with different numbers of data points. We used four nodes for execution, and the δ is fixed at 0.001.

that the AS scheme improves the execution time significantly even with a single active storage node because of the reduction in data transfer – a single active storage node outperforms four client and server nodes in the TS configuration. We also observe that these improvements are consistent with respect to the number of nodes.

In addition to looking at different numbers of nodes, we experimented with different sizes of data sets. Figure 12 shows the performance of SUM where we varied the data set size from 64 MB to 2048 MB. Recall that for SUM, the default input data was 512 MB, and we used four nodes to execute both the TS and AS schemes. The results clearly indicate that AS shows even greater relative improvement with larger data sizes because larger data sets incur more data movement between servers and clients.

C. Impact of GPU Offloading

So far, we have focused on the behavior of our primitive benchmarks and presented the effectiveness of our active storage system by filtering the data transfer between file server and clients. However, certain data-mining kernels may easily become compute-intensive. To study this possibility, we looked at the KMEANS benchmark in detail in terms of different computational workloads. We used KMEANS because we can easily change the amount of computation it needs to perform by changing the threshold value. Recall that the default threshold value (δ) we used for KMEANS so far was 1.0, which leads to executing the KMEANS kernel only once. Figure 13 shows the execution time and percentage of I/O and computation of KMEANS while varying the δ value from 1.0 to 0.0005. In the KMEANS algorithm, the smaller δ means a stricter convergence condition, thereby incurring more computation. As we can see from Figure 13, depending on the user's preference, KMEANS can be compute-intensive, and the overall execution time also increases significantly. For example, changing the δ value from 0.05 to 0.01 resulted in 6.5-fold increase (from 0.8 seconds to 5.2 seconds) in the execution time. In such cases, just embedding the KMEANS kernel in the storage server will not bring any substantial improvement because computation time dominates overall execution time.

Figure 14 shows the total execution time taken to run the KMEANS benchmark while changing the number of nodes. As we can see in this graph, increasing the number of nodes (i.e., increasing the parallelism) improves the performance of all three schemes. However, the improvements brought by AS are negligible with respect to TS. This result clearly demonstrates that the benefits by AS alone are limited by the time spent on computation with the strict convergence condition. The AS+GPU scheme, however, improves the performance significantly, 58.4% on average, as compared to AS. These huge improvements are mainly from the reduction in computation time through the execution of kernels on a GPU board used in each storage server.

Figure 15 gives the total execution time of KMEANS when we vary the data set size. In this set of experiments, we fixed the number of nodes to the default of four. We see in Figure 15 that in all three cases the execution time

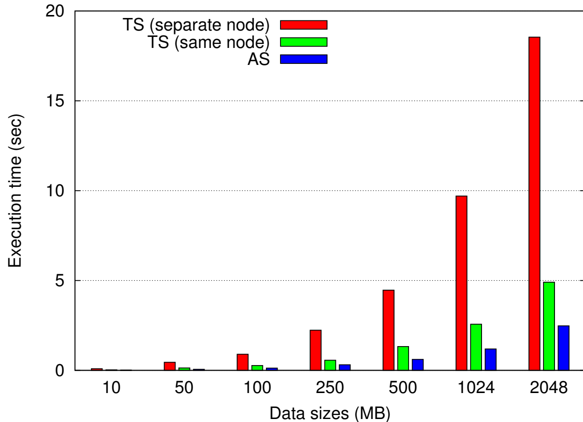

Fig. 16. Performance comparison of TS (separate node), TS (same node), and AS. TS (separate node) is executed using two nodes (one client and one server), and TS (same node) and AS are executed using one node. TS (separate node) incurs internode communication (from server to client), whereas TS (same node) incurs interprocess memory copy overheads (between file server process and client process).

increases as the number of data points increases. The reason is that, when we increase the data set size, the amount of computation also increases accordingly. Again, there is almost no difference between TS and AS since the computation time is dominant. Overall, AS+GPU outperforms AS, and the improvement by AS+GPU becomes larger as the number of data points increases.

D. Collocating Client and Server Activities

Our evaluation so far has been based on a system configuration in which the storage servers and clients are in separate nodes (see Figure 3). However, in large-scale analysis frameworks such as MapReduce [14] and Hadoop [22], both the file servers and client nodes are hosted on the same nodes. In our next set of experiments, we compared our approach with this combined analysis/storage approach.

To see the effect of Hadoop-style execution, we ran the SUM benchmark while varying the amount of input data (10 MB to 2048 MB). Since it filters most of input data (see the last column of Table II) and since the computational load is minor ($\sim$3% of total execution time), the result will show how much benefit the AS scheme can bring in terms of bandwidth saving. We also collected results using an additional configuration (denoted as "TS (same node)" in Figure 16) where the storage servers and client processes are running on the same node, which is how Hadoop works. Figure 16 indicates that the Hadoop approach reduces execution time dramatically, 73% on average. This result is not surprising because executing both server and client programs on the same node will not incur internode communication. The AS scheme, however, improves the performance by 53.1% on top of that. In this case, AS removes not only internode communication, as in the TS (same node) case, but also interprocess memory copy overheads, that is, copying I/O buffer from the server process to the client process's address space. Another advantage of AS over the Hadoop approach is that AS requires minimal modification of existing analysis codes that are already optimized to existing parallel I/O interfaces.

VI. Related Work

The concept of executing operations in the storage system has been investigated and successfully deployed in a number of forms. Most relevant to scientific computing, many groups have investigated the use of *active storage* techniques that locate computation with data [1], [12], [23], [40], [45], [44], [49], [25], [43], [24], [30], [50], [5], [33], [9], [18], [39]. This concept of moving computation closer to the place where the data reside is similar to the processing-in-memory approach for random access memory [27].

The idea of active storage was originally proposed by Acharya et al. [1] and evaluated in the context of active disk architectures. As the name implies, their main idea was to utilize the extra processing power in the disk drive itself. They also proposed a stream-based programming model that allows application code to execute on the disks. Riedel et al. [40] proposed a similar system and identified a set of applications that may benefit from active disks. The application categories include filtering, batching, and real-time imaging. Keeton et al. [25] presented *intelligent* disks (IDISKs), targeted at decision support database servers. IDISKs have many similarities with other active disk work, but their system includes a rudimentary communication component. Huston et al. [24] proposed a concept of *early discard* for interactive search. To filter a large amount of unindexed data for search, they send a *searchlet* to the active storage systems. Note that the idea of placing computation where data reside was proposed even earlier than these active storage efforts in the context of database architectures [15].

Recent work has focused on the application of active storage principles in the context of parallel file systems [37], [38], [17]. Piernas et al. [38] proposed an approach to extend active disk techniques to the parallel file system environment in order to utilize the available computing power in the storage nodes. In their more recent work, they showed that an active storage system can support scientific data, which is often stored in striped files and files with complex formats (e.g., netCDF [47]) [37]. These approaches are most relevant to our work in that the active storage concept is built within the context of parallel file systems, but their active storage is implemented in the user space of the Lustre file systems. Therefore, they need to implement separate libraries to communicate with other servers. Because of this limitation, both client and server should be hosted in the same pool of nodes. Our approach, on the other hand, does not make such an assumption about the server and client deployment architecture. Further, because of the lack of server-to-server communication primitives on the server side, their approach is restricted to the case where the data set is distributed in independent files in each server. Our approach, however, can be used with both independent files and striped files.

One of the major issues in designing a usable active storage system is to provide general software support for identifying

and deploying a set of operations that are being embedded on the server side. Studies that have tackled this issue have proposed the use of stream-based disklet [1], early discard through searchlet [24], dynamic function load-balancing [3], scriptable RPC [43], and hosting application extensions called adjuncts on the storage server [9]. While general-purpose function/application offloading is a crucial building block for an active storage system, these approaches are orthogonal to our approach. We also mention that, as studied by Wickremesinghe et al. [50] and Oldfield and Kotz [33], the effect of using active storage could be restricted when multiple users simultaneously access active and traditional storage services, mainly because of resource contention. Again, extending the active storage model for managing computational and storage resources is not the focus of our study. In this paper, we assume the existence of system support to write and deploy custom server-resident code. Our approach focuses on the mechanisms to enable active storage operation on parallel file systems.

Recently, the MapReduce and Hadoop paradigms have received considerable attention for large-scale data analysis on commodity hardware [14], [22], [51]. To handle large input data and computation distributed across many machines, researchers designed an abstraction that allows one to write large-scale data processing routines while alleviating the details of parallelization, fault tolerance, data distribution, and load balance. Patil et al. [34] studied key file system abstractions that need to be exposed for cluster file systems such as GPFS [42], PVFS [7], and Lustre [46], to accommodate MapReduce types of workloads on top of them. Pavlo et al. [35] recently evaluated both the MapReduce and parallel database abstractions in terms of performance and development complexity. Both MapReduce and Hadoop are typically layered on top of distributed file systems—GoogleFS [19] and Hadoop distributed FS [4], respectively—that are designed to meet scalable storage requirements. Therefore, these approaches are orthogonal to our approach. In fact, as shown in Section V-D, our approach can complement a Hadoop-style execution environment by removing interprocess communications within the same node.

Modern visualization and data analysis clusters are often built by using a hybrid of general-purpose CPUs and accelerators such as FPGAs and GPUs for interactive visualization and data analysis. For example, Netezza's data warehousing and analytic platform is made of multicore Intel-based blades, implemented in conjunction with commodity disk drive and data filtering units using FPGAs [31]. Their analytic platform is optimized mainly for database applications. Curry et al. [13] used a GPU to perform parity generation in a RAID system and showed that their approach can support the RAID workload without affecting performance through offloading Reed-Solomon coding into GPUs. Our approach also uses a GPU for offloading compute-intensive kernels, but it differs from these prior works because we make GPU data analytic kernels available to the parallel runtime interface for facilitating integration with existing parallel I/O applications.

VII. Conclusion and Future Work

This paper has proposed an active storage system in the context of parallel file systems and has demonstrated that, with enhanced runtime interfaces, we can improve the performance of data analysis kernels significantly. In the proposed approach, the parallel runtime interface enables application codes to utilize the data analysis kernels embedded in the parallel file system. In order to enable an individual file server to perform active storage operations without client intervention, our proposed approach dynamically adjusts to deal with data elements that cross stripe boundaries. We have also implemented server-to-server communication for reduction and aggregation to allow pure server-side computation. Our experimental results using a set of data analysis kernels demonstrate that our scheme brings substantial performance improvements as compared to the traditional storage system. We also demonstrated that the improvements brought by our approach remain consistent across variations in number of nodes and data set size. Moreover, we showed that using a GPU accelerator improves the performance of compute-intensive kernels considerably.

We are currently extending our research in several directions. We are implementing more diverse data analysis kernels in both general-purpose CPUs and GPUs, and we plan to embed them in the file servers. To address the challenges of programming models in the context of the active storage model, we are exploring new ways to allow users to write their own data analysis kernels and download them to our active storage system dynamically. Moreover, we are investigating how to extend our runtime interfaces to higher-level interfaces (e.g., Parallel netCDF [29]).

Acknowledgments

We thank the anonymous reviewers for their valuable feedback and suggestions. We also thank Tom Peterka for providing us with parallel volume-rendering applications. This work was supported by the Office of Advanced Scientific Computing Research, Office of Science, U.S. Department of Energy, under Contract DE-AC02-06CH11357. This work was also supported in part by DOE FASTOS award number DE-FG02-08ER25848, DOE SCIDAC-2: Scientific Data Management Center for Enabling Technologies grant DE-FC02-07ER25808, NSF HECURA CCF-0621443, NSF SDCI OCI-0724599, and CNS-0830927.

References

[1] A. Acharya, M. Uysal, and J. H. Saltz. Active Disks: Programming Model, Algorithms and Evaluation. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 81–91, 1998.

[2] Amazon. Amazon Simple Storage Service (Amazon S3). http://aws.amazon.com/s3/.

[3] K. Amiri, D. Petrou, G. R. Ganger, and G. A. Gibson. Dynamic Function Placement for Data-Intensive Cluster Computing. In *Proceedings of the USENIX Technical Conference*, pages 25–25, 2000.

[4] D. Borthakur. HDFS Architecture. http://hadoop.apache.org/common/docs/current/hdfs_design.html.

[5] J. B. Buck, N. Watkins, C. Maltzahn, and S. A. Brandt. Abstract Storage: Moving File Format-Specific Abstractions Intopetabyte-Scale Storage Systems. In *Proceedings of the Second International Workshop on Data-Aware Distributed Computing*, pages 31–40, 2009.

[6] P. H. Carns. *Achieving Scalability in Parallel File Systems*. PhD dissertation, Clemson University, Department of Electrical and Computer Engineering, May 2005.

[7] P. H. Carns, W. B. L. III, R. B. Ross, and R. Thakur. PVFS: A Parallel File System for Linux Clusters. In *Proceedings of the Annual Linux Showcase and Conference*, pages 317–327, 2000.

[8] P. H. Carns, B. W. Settlemyer, and I. Water B. Ligon. Using Server to Server Communication in Parallel File Systems to Simplify Consistency and Improve Performance. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*, November 2008.

[9] D. Chambliss, P. Pandey, T. Thakur, A. Fleshler, T. Clark, J. A. Ruddy, K. D. Gougherty, M. Kalos, L. Merithew, J. G. Thompson, and H. M. Yudenfriend. An Architecture for Storage-Hosted Application Extensions. *IBM J. Res. Dev.*, 52(4):427–437, 2008.

[10] J. Chen. Terascale Direct Numerical Simulations of Turbulent Combustion. In *Proceedings of the ACM/IEEE Conference on Supercomputing*, page 55, 2006.

[11] A. Chervenak, J. M. Schopf, L. Pearlman, M.-H. Su, S. Bharathi, L. Cinquini, M. D'Arcy, N. Miller, and D. Bernholdt. Monitoring the Earth System Grid with MDS4. In *Proceedings of the IEEE International Conference on e-Science and Grid Computing*, 2006.

[12] S. Chiu, W. keng Liao, and A. N. Choudhary. Design and Evaluation of Distributed Smart Disk Architecture for I/O-Intensive Workloads. In *Proceedings of the International Conference on Computational Science*, pages 230–241, 2003.

[13] M. L. Curry, A. Skjellum, H. L. Ward, and R. Brightwell. Arbitrary Dimension Reed-Solomon Coding and Decoding for Extended RAID on GPUs. In *Proceedings of the Petascale Data Storage Workshop*, 2008.

[14] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *Proceedings of the USENIX Symposium on Operating System Design and Implementation*, pages 137–150, 2004.

[15] D. J. DeWitt and P. B. Hawthorn. A Performance Evaluation of Data Base Machine Architectures. In *Proceedings of the International Conference on Very Large Data Bases*, pages 199–214, 1981.

[16] EMC Corporation. EMC Symmetrix V-Max Storage System Specification Sheet, 2009.

[17] E. J. Felix, K. Fox, K. Regimbal, and J. Nieplocha. Active Storage Processing in a Parallel File System. In *Proceedings of the 6th LCI International Conference on Linux Clusters: The HPC Revolution*, 2005.

[18] B. G. Fitch, A. Rayshubskiy, M. C. Pitman, T. C. Ward, and R. S. Germain. Using the Active Storage Fabrics Model to Address Petascale Storage Challenges. In *Proceedings of the 4th Petascale Data Storage Workshop*, 2009.

[19] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google File System. In *Proceedings of the ACM Symposium on Operating Systems Principles*, pages 29–43, 2003.

[20] A. Grama, G. Karypis, V. Kumar, and A. Gupta. *Introduction to Parallel Computing (2nd edition)*. Addison Wesley, 2003.

[21] W. Gropp, E. Lusk, and R. Thakur. *Using MPI-2: Advanced Features of the Message-Passing Interface*. The MIT Press, Cambridge, MA, 1999.

[22] Hadoop. http://hadoop.apache.org/.

[23] W. Hsu, A. J. Smith, and H. C. Young. Projecting the Performance of Decision Support Workloads on Systems with Smart Storage (SmartSTOR). In *Proceedings of the International Conference on Parallel and Distributed Processing*, pages 417–425, 2000.

[24] L. Huston, R. Sukthankar, R. Wickremesinghe, M. Satyanarayanan, G. R. Ganger, E. Riedel, and A. Ailamaki. Diamond: A Storage Architecture for Early Discard in Interactive Search. In *Proceedings of the USENIX Conference on File and Storage Technologies*, pages 73–86, 2004.

[25] K. Keeton, D. A. Patterson, and J. M. Hellerstein. A Case for Intelligent Disks (IDISKs). *SIGMOD Record*, 27(3):42–52, 1998.

[26] S. Klasky. Personal Correspondence, June 2008.

[27] P. M. Kogge, S. C. Bass, J. B. Brockman, D. Z. Chen, and H. S. E. Pursuing a Petaflop: Point Designs for 100TF Computers Using PIM Technologies. In *Proceedings of the Symposium on Frontiers of Massively Parallel Computation*, pages 25–31, October 1996.

[28] D. T. Larose. *Data Mining Methods and Models*. John Wiley & Sons, 2006.

[29] J. Li, W.-k. 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 *Proceedings of the ACM/IEEE Conference on Supercomputing*, page 39, 2003.

[30] G. Memik, M. T. Kandemir, and A. N. Choudhary. Design and Evaluation of Smart Disk Architecture for DSS Commercial Workloads. In *Proceedings of the International Conference on Parallel Processing*, pages 335–, 2000.

[31] Netezza Corporation. The Netezza Data Appliance Architecture: A Platform for High Performance Data Warehousing and Analytics, 2009.

[32] NVIDIA. CUDA Programming Guide 2.3. http://www.nvidia.com/.

[33] R. Oldfield and D. Kotz. Improving Data Access for Computational Grid Applications. *Cluster Computing*, 9(1):79–99, 2006.

[34] S. Patil, G. A. Gibson, G. R. Ganger, J. Lopez, M. Polte, W. Tantisiroj, and L. Xiao. In Search of an API for Scalable File Systems: Under the Table or Above It? In *USENIX HotCloud Workshop*, June 2009.

[35] A. Pavlo, E. Paulson, A. Rasin, D. J. Abadi, D. J. DeWitt, S. Madden, and M. StoneBraker. A Comparison of Approches to Large-Scale Data Analysis. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 2009.

[36] T. Peterka, H. Yu, R. Ross, K.-L. Ma, and R. Latham. End-to-End Study of Parallel Volume Rendering on the IBM Blue Gene/P. In *Proceedings of the International Conference on Parallel Processing*, 2009.

[37] J. Piernas and J. Nieplocha. Efficient Management of Complex Striped Files in Active Storage. In *Proceedings of the Euro-Par Conference*, pages 676–685, 2008.

[38] J. Piernas, J. Nieplocha, and E. J. Felix. Evaluation of Active Storage Strategies for the Lustre Parallel File System. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage, and Analysis*, page 28, 2007.

[39] L. Qiao, V. Raman, I. Narang, P. Pandey, D. Chambliss, G. Fuh, J. Ruddy, Y.-L. Chen, K.-H. Yang, and F.-L. Lin. Integration of Server, Storage and Database Stack: Moving Processing Towards Data. In *Proceedings of the IEEE 24th International Conference on Data Engineering*, pages 1200–1208, 2008.

[40] E. Riedel, G. A. Gibson, and C. Faloutsos. Active Storage for Large-Scale Data Mining and Multimedia. In *Proceedings of the International Conference on Very Large Data Bases*, pages 62–73, 1998.

[41] K. Riley. Personal Correspondence, July 2008.

[42] F. Schmuck and R. Haskin. GPFS: A Shared-Disk File System for Large Computing Clusters. In *Proceedings of the USENIX Conference on File and Storage Technologies*, pages 231–244, 2002.

[43] M. Sivathanu, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Evolving RPC for Active Storage. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 264–276, 2002.

[44] M. Sivathanu, L. N. Bairavasundaram, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Database-Aware Semantically-Smart Storage. In *Proceedings of the USENIX Conference on File and Storage Technologies*, pages 239–252, 2005.

[45] M. Sivathanu, V. Prabhakaran, F. I. Popovici, T. E. Denehy, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Semantically-Smart Disk Systems. In *Proceedings of the USENIX Conference on File and Storage Technologies*, pages 73–88, 2003.

[46] SUN Microsystems. Lustre File System. http://www.sun.com/software/products/lustre/.

[47] Unidata. NetCDF home page, 2001.

[48] U.S. Department of Energy. DOE Genomics:GTL Roadmap – Systems Biology for Energy and Environment. DOE/SC-0090, August 2005.

[49] M. Uysal, A. Acharya, and J. H. Saltz. Evaluation of Active Disks for Decision Support Databases. In *Proceedings of the International Symposium on High-Performance Computer Architecture*, pages 337–348, 2000.

[50] R. Wickremesinghe, J. S. Chase, and J. S. Vitter. Distributed Computing with Load-Managed Active Storage. In *Proceedings of the 11th IEEE International Symposium on High Performance Distributed Computing*, pages 13–23, 2002.

[51] M. Zaharia, A. Konwinski, A. D. Joseph, R. H. Katz, and I. Stoica. Improving MapReduce Performance in Heterogeneous Environments. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation*, pages 29–42, 2008.