Toward Efficient and Simplified Distributed Data Intensive Computing

Yunhong Gu and Robert Grossman

Abstract—While the capability of computing systems has been increasing at Moore's Law, the amount of digital data has been increasing even faster. There is a growing need for systems that can manage and analyze very large data sets, preferably on shared-nothing commodity systems due to their low expense. In this paper, we describe the design and implementation of a distributed file system called Sector and an associated programming framework called Sphere that processes the data managed by Sector in parallel. Sphere is designed so that the processing of data can be done in place over the data whenever possible. Sometimes, this is called data locality. We describe the directives Sphere supports to improve data locality. In our experimental studies, the Sector/Sphere system has consistently performed about 2-4 times faster than Hadoop, the most popular system for processing very large data sets.

Index Terms—Distributed programming, distributed file systems, performance, frameworks.

1 Introduction

Instruments that generate data have increased their capability following Moore's Law, just as computers have. Yet, there is still a growing gap between the amount of data that is being produced and the capability of current systems to store and analyze these data.

As an example, the current generation of high throughput genomic sequencers now produces 1 TB of data per run, and the next generation of sequencers will produce data sets that are 10 TB or larger per run. A sequencing center might have several such systems, with each producing a run or more per week.

Data sets that are hundreds of TBs or more are growing more common. A current rack of commodity computers might contain 32 computers, with each computer containing four 1 TB disks. The challenge is to develop a framework to support data intensive computing that provides persistent storage for large data sets (that require multiple racks to store) as well as balanced computing so that these persistent data can be analyzed.

To put it another way, platforms for data intensive computing must provide persistent storage that spans hundreds to thousands of disks and provide balanced computing so that these persistent data can be analyzed. While there are scalable file systems (e.g., Lustre, GPFS, PVFS, etc.), data processing schedulers (Condor, LSF, etc.), and frameworks (MPI, etc.), few integrate storage and processing together as is required for data intensive computing.

The reason that it is so important to tightly integrate the storage system and the processing system is because data locality is one of the fundamental principles underlying the efficient processing of very large data sets. The requirement that all the data in a computation pass through a single central location (or several such locations) can be so costly that it can reduce performance by several orders of magnitude. For example, a 1,000-node system would require aggregate data IO speed at TB/s in order to achieve an average performance per node comparable to a single node with internal disks.

Supercomputer systems, as they are generally deployed, store data in external storage (RAID, SAN, NAS, etc.), transfer the data to the supercomputer, and then transfer any output files back to the storage system. The data transfer channel between the external storage system and the supercomputer is often a very serious bottleneck.

In contrast, Google has developed a proprietary storage system called the Google File System (GFS) [5] and an associated processing system called MapReduce [4] that has very successfully integrated large-scale storage and data processing.

MapReduce was not intended as a general framework for parallel computing. Instead, it is designed to make it very simple to write parallel programs for certain applications, such as those that involve web or log data. One of the questions that interest us in this paper is whether there are other parallel programming frameworks that are just as easy to use as MapReduce and also applicable for more general classes of problems or for different classes of problems.

Hadoop [17] is an open source implementation of the GFS/MapReduce design. It is now the dominant open source platform for distributed data storage and parallel data processing over commodity servers. While Hadoop's performance is very impressive, there are still technical challenges that need to be addressed in order to improve its performance and increase its applicability.

- *Y. Gu is with the Laboratory for Advanced Computing, University of Illinois at Chicago, 713 Science and Engineering Offices, MC 249, 851 South Morgan Street, Chicago, IL 60607-7045. E-mail: yunhong@lac.uic.edu.*
- *R. Grossman is with the Laboratory for Advanced Computing, University of Illinois at Chicago and the Open Data Group, 715 Science and Engineering Offices, MC 249, 851 South Morgan Street, Chicago, IL 60607-7045. E-mail: grosman@uic.edu.*

Manuscript received 2 Jan. 2010; revised 2 July 2010; accepted 19 Aug. 2010; published online 16 Feb. 2011.
Recommended for acceptance by I. Raicu, I. Foster, and Y. Zhao.
For information on obtaining reprints of this article, please send e-mail to: tpds@computer.org, and reference IEEECS Log Number TPDSSI-2010-01-0008.
Digital Object Identifier no. 10.1109/TPDS.2011.67.

During the last four years, we have designed and implemented a scalable storage system called Sector and a parallel data processing framework called Sphere. Initially, we were not aware of the GFS/MapReduce system and, although there are some similarities between GFS/MapReduce and Sector/Sphere, there are also some important differences. Unlike GFS and traditional file systems, Sector is an application-aware file system and it can organize files in a way to support efficient data processing. Unlike MapReduce, Sphere allows arbitrary user-defined functions (UDFs) to process independent data segments in parallel. A Sector/Sphere segment may be a record, a group of records, a file, or a directory.

Previously, we have described the design of version 1.0 of Sector and Sphere [9]. Based upon our experiences with Sector and Sphere, we have made significant changes to these systems. This paper is about these changes: we describe the approach to data locality in version 2.0 of Sector, and Sphere's new approach to scheduling, load balancing, and fault tolerance.

Version 1.0 of the Sector file system was designed for wide area data storage and distribution [9]. Sector version 2.0 supports new mechanisms for fine tuning data placement to improve data locality so that data can be processed more efficiently by Sphere. Version 2.0 of Sphere can now support multiple inputs/outputs, multiple UDFs, and different ways of composing them. Sphere's load balancing and fault tolerance have also been enhanced with new features.

We start with a brief description of the Sector/Sphere system architecture in Section 2. We explain the application-aware features of Sector in Section 3 and describe the Sphere data processing framework in Section 4. We describe the results from our most recent experimental studies in Section 5. Related work is discussed in Section 6. The paper is concluded in Section 7 with a brief look at future work.

2 Sector/Sphere System Architecture

In this section, we briefly describe the Sector/Sphere system. A more detailed description can be found in [9]. In this section, we also discuss several features of the current version (version 2.3) of Sector/Sphere that were not present in the earlier version (version 1.20) described in [9].

Sector is a distributed file system (DFS) and Sphere is a parallel data processing system designed to work with data managed by Sector. In contrast to most traditional distributed file systems that rely on hardware to provide fault tolerance, Sector implements fault tolerance by ensuring that there are always a (user specified) number of replicas of the data available in the file system and making sure that the replicas are consistent. With this approach, Sector can be installed on less expensive commodity hardware. Sector does not implement a native file system itself, but instead relies on the local file systems that are on each of the nodes. In fact, the local storage on each node need not be a file system service, but instead can be a database service, a memory-based service, etc. All that is needed is a Sector driver for the service.

Sector automatically replicates files to different nodes to provide high reliability and availability. In addition, this strategy favors read-intensive scenarios since clients can read from different replicas. Of course, write operations are slower since they require synchronization between all the replicas.

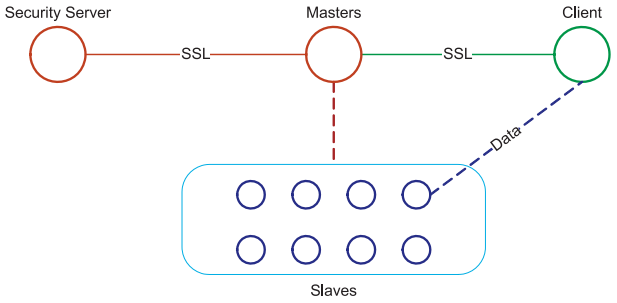

Fig. 1. Sector/Sphere system architecture. The Sector system consists of a security server, one or more master servers, and one or more slave nodes. Sector assumes that the master and slaves are connected with a high-speed network.

Sector can also be deployed over wide area networks. Sector is aware of the network topology when it places replicas. This means that a Sector client can choose the nearest replica to improve performance. In this sense, Sector can be viewed as a content delivery system [10]. For example, Sector is used to store and distribute bulk downloads of the Sloan Digital Sky Survey.

Sphere is a data processing system that is tightly coupled with Sector. Because of this coupling, Sector and Sphere can make intelligent decisions about job scheduling and data location. This is quite different than most distributed file systems that are not coupled to distributed data processing systems.

Sphere provides a programming framework that developers can use to process data stored in Sector. Sphere allows a user-defined function to run on various granularities of data managed by Sector. Specifically, Sphere can process a Sector record, a block of Sector records, a file, or a directory of files. Note that this model is different than the model used by grid job schedulers (such as Condor and LSF) and distributed programming primitives (such as MPI and PVM), both of which are independent of the underlying file system. Most grid schedulers manage tasks or jobs, while Sphere's job scheduling is tightly coupled with data.

Fig. 1 illustrates the architecture of Sector. For simplicity, in the rest of this paper, we use the term Sector to refer to Sector/Sphere. Sector contains one security server that is responsible for authenticating master servers, slave nodes, and users. The security server can be configured to retrieve this information from various sources (such as from a file, database, or LDAP service).

One or more master servers can be started (only one master server was supported in version 1.0 of Sector). The master servers are responsible for maintaining the metadata of the storage system and for scheduling requests from users. All Sector masters are active and can be inserted or removed at runtime. The system load is balanced between the various master servers. Details of metadata synchronization, distributed locking, and load balancing between the masters are beyond the scope of this paper and we will focus on the data processing framework and support from the storage layer.

The slave nodes are the computers that actually store and process the data. Sector assumes that the slaves are racks of

commodity computers that have internal disks or are connected to external storage via a high-speed connection. In addition, all slaves should be interconnected by high-speed networks.

Finally, a Sector client is a computer that issues requests to the Sector system and accepts responses. File IO is performed using UDT [8] to transfer data between the slaves and between the client and slaves.

Sector is written in C++. It is open source and available from sector.sf.net.

3 Data Placement

Although some file systems are optimized for certain IO patterns (e.g., read intensive, a large number of small files, etc.), with traditional file systems, applications cannot directly control the placement of files within the file system. Note that as described in [5], the Google File System does not allow GFS applications to control the placement of files.

In contrast, as we will describe in this section, Sector gives users several ways to specify the location of data in order to better support data locality. These mechanisms are not required, and, when not used, Sector will supply appropriate defaults.

Optimizing data locality is perhaps the most important factor affecting the overall performance of a system such as Sector. In many cases, moving data from remote nodes to an available compute node can take more time than the computation itself. For this reason, GFS exposes data location information to MapReduce so that MapReduce can select the closest compute node to process each data record.

Sector goes a significant step further in this direction. It not only exposes data locations to Sphere, but also allows Sphere applications to specify locations in a convenient way. In this section, we describe in detail the data locality support provided by Sector.

3.1 Block versus File

A Sector data set is a collection of data files. Sector stores these files on the local file system of all the participating nodes. Sector does not split a file into blocks but places a single file on a single node. However, replicas of files are placed on distinct slave nodes.

The Sector no-block strategy means that users must manually split their large data sets into appropriately sized files. For some applications, this can create a small amount of additional work. In practice though, either a large data set is already split into multiple files or this can be done with little effort.

There are two advantages of this approach. First, it is easier to parse data since users know the semantics of the data set and can split the data set into files at record boundaries. In a block-based system, data are randomly split and certain binary data cannot be parsed easily without scanning the data from the beginning of the first block. Second, for applications that process the data by files, a block-based system needs to gather all the required blocks into one node before beginning the computation.

In several of our experimental studies with Hadoop, we noticed that the optimal performance was achieved when the Hadoop block size was increased to roughly the size of a file. Since blocks in the HDFS are not split, this effectively reduces to the same strategy employed by Sector.

There are, of course, disadvantages for the file-based approach that Sector takes. The major disadvantage is limitations on the file size since Sector assumes that files fit on a single slave node. Since Sector uses files for load balancing, Sector also benefits from having enough files for its load balancing algorithms to be effective. In other words, Sector assumes that the files are small enough to fit on slave nodes and that there are enough of them so that it can use them for load balancing.

3.2 Directory and File Family Support

In general, Sector scatters the files of a Sector data set over all the slave nodes. Some applications require that a certain collection of files be colocated in a single directory. To support this, users can create a special system file called ".nosplit" and place it in the directory.

Once this file is present in a directory, all files and subdirectories will be replicated together to the same slave node, or uploaded to the same node during processing by Sphere.

This feature has turned out to be quite useful. For example, for Sphere processes involving multiple inputs, the corresponding data files can all be put in the same directory, and in this way, the multiple inputs to Sphere will be colocated.

This feature has also turned out to be important when porting legacy applications to Sphere. For example, we ported an image processing application that processed directories containing subdirectories containing image files. Placing a ".nosplit" directive in the top level directory was all Sphere required in order to colocate the image data appropriately. Specifically, in this example, a Sphere UDF invoked the legacy application and nothing more was required to colocate the data of a single image and of collections of images that needed to be processed together by the same node. In contrast, a MapReduce application on a non-Sector file system needs an additional step to "map" all related files to one location prior to processing.

Sector supports what it calls *file families* that are defined by applying specific rules about data locality. For example, in a column-based database system on top of Sector that we are developing (roughly comparable to BigTable [20]), each column of a table contains multiple segment files associated with a key column. By specifying the appropriate rule, Sector can place all column files belonging to the same segment together, and, in this way, significantly reduce the amount of data that must be moved when querying multiple columns. Sector also places index files and the associated data files together to reduce search time.

3.3 Wide Area Network Support

Sector is one of the few distributed file systems that can be deployed effectively over wide area networks. This is important because very large data sets are often collected from geographically distributed locations, or need to be shared among users that are geographically distributed. With this ability, a data set can be processed by a single Sphere application, even though the data are geographically distributed.

Two major features in Sector allow it to support wide area networks. First, Sector uses a high-speed transport protocol called UDT [8]. We choose to use UDT instead of TCP since UDT has proven to be better than TCP for moving large data sets over wide area high performance networks [8]. On wide area networks, UDT provides a data transfer throughput that is almost as fast as within local area networks. In contrast, the performance of TCP over wide area networks is often quite poor. In practice, this makes it much more difficult to set up a file system based upon TCP across multiple data centers.

UDT runs on top of UDP with its own reliability control and congestion control, both contributing to its performance. UDT uses explicit negative acknowledgment and periodical acknowledgment, which can significantly reduce control traffic while yielding high performance for bulk data transfer. The congestion control can discover available network bandwidth much faster than TCP does, especially over wide area networks. UDT flows can also share the bandwidth fairly, even if they have different round trip times.

Second, the replication strategy used by Sector automatically integrates knowledge of the topology of the wide area network. As previously mentioned, Sector uses replication to provide data reliability over unreliable components and to improve availability in order to improve data processing performance.

The default Sector replication algorithm chooses slave nodes so that replicas of the same file are as far from each other as possible. This increases data reliability because, for example, even if a data center goes offline, the data can still be retrieved from another data center. An isolated data center can continue to manage and process data as long as there is an active master. When the isolated data center rejoins Sector, any resulting conflicts are resolved.

More importantly, in a wide area system, this approach allows a client to choose a nearby replica, which generally results in better data transfer throughput.

To improve performance, Sector allows the user to choose whether replicas are created immediately when writing data or periodically. For Sphere UDFs, all output files generated are replicated periodically. For this reason, Sphere's performance is almost independent of the number of replicas in the system (more replicas of the input files do increase performance slightly).

Additionally, Sector provides a set of authentication and optional data transfer encryption mechanisms to support security over a wide area network.

3.4 Sector Interfaces and Management Tools

Sector provides an almost POSIX-compliant file system interface and our goal is to make it 100 percent POSIX-compliant in future versions. Sector can be mounted on a client and standard system commands (e.g., ls. stat, mkdir, rm, cp, etc.) can be used to manage and access Sector files. Sector also provides its own client side tools that are similar to the system commands (e.g., sector_ls works just like ls). Additional tools are provided to support file transfer between Sector and the local file system.

Management tools allow users to view the current status of the system (e.g., file system status, resource usage, etc.), start/shutdown part of or the whole system, and update system configurations (e.g., number of replicas for a particular file or directory).

4 Sphere Parallel Data Processing Framework

Sphere applies the same User-defined Function independently to all the data segments comprising a Sector data set. This is one of the basic ways that Sphere processes data in parallel. Sphere automatically and transparently manages the data segments, the load balancing, and the fault tolerance. The application developers only need to write a C++ UDF that specifies the Sphere inputs, outputs, and how the data are processed. Sphere UDFs need to follow certain conventions regarding variable and function names.

4.1 The UDF Model

As mentioned in Section 1, we use the term *data segment* to refer to a portion of a Sector data set. A data segment can be a single record, a contiguous sequence of records, a file, or a directory of files. Sphere processes data by applying the same UDF independently to data segments, which are usually scattered over multiple Sector slave nodes. The input to a Sphere UDF is a Sector data set (or multiple Sector data sets). The output of a Sphere UDF is also a Sector data set. Recall that a Sector data set consists of multiple files.

Consider the following serial program for processing data:

```
for (int i = 0; i < total_seg_num; ++i)
  process(segment[i]);
```

To turn this into a Sphere program, the "process" function is made into a Sphere UDF with the following format:

```
UDF_NAME(input, output, file);
```

Here, "input," "output," and "file" are three Sphere data structures. The "input" is a Sphere programming data structure that contains the input data in a binary format, as well as other information, including an offset index (to parse multiple records) and any required user specified parameters. The UDF writes the result into the "output" data structure. The "file" data structure contains information about the local file system, which is necessary when, for example, the UDF needs to obtain temporary space from the local file system.

A Sphere application developer writes a UDF, compiles it into a dynamic library file, and passes it to the Sphere application. Sphere then uploads the library file to each of the participating slave nodes, which in turn calls the UDF to process all its local data segments.

The above serial program can be written by Sphere as

```
SphereProc.run(file_set, NULL, UDF_NAME);
```

where "file[lowbar]set" is the input data set. In this example, the UDF will process a file at a time. Additional parameters can be used to specify another data segment unit, such as a record or multiple records. To process files, Sphere maintains an index associated with each data file that specifies the offset and length of each record in the file.

An example of a Sphere UDF is given in the Appendix. The example sorts a Sector data set.

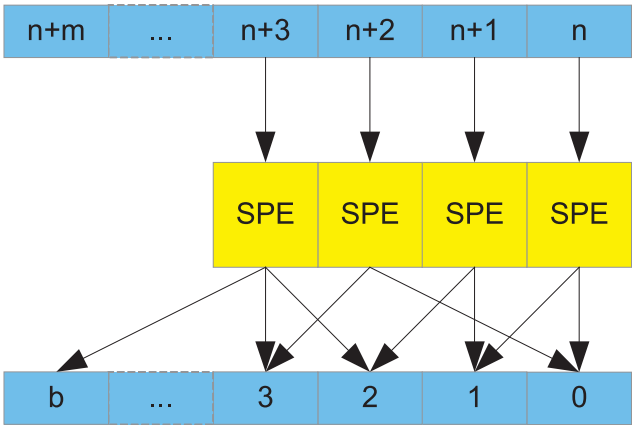

Fig. 2. The hashing (bucket) process in Sphere is similar to a Reduce process. The top part of the diagram is the input data stream that is split into data segments $(n, \ldots, n+m)$. The middle part is the Sphere processing engine associated with each slave node. The bottom part is the output bucket files $(0, \ldots, b)$.

The Sphere UDF in the example above generates local output files only. However, Sphere allows a UDF to specify an ID for each output record. This is usually called a bucket ID because the process is similar to using a hashing function to assign data records to different buckets. Bucket IDs are unique system wide and all records with the same bucket ID are placed by Sector into the same bucket, which is simply a Sector file. The resulting bucket file may or may not be on the same node where the data are processed. In general, bucket files contain data processed by many different slave nodes. The process is illustrated in Fig. 2.

In Fig. 2, a Sphere Processing Engine (SPE) is a Sphere daemon running on each slave node. The input stream is split (by a Sphere scheduler) into multiple segments $(n, n+1, \ldots n+m)$. Each SPE processes one segment at a time and sends the results to the appropriate bucket files $(0, 1, \ldots, b)$. The scheduler always tries to process data segments locally, unless there are idle SPEs; in this case, data segments are sent to nearby SPEs. The application developer can specify the location of bucket files if desired. Otherwise, the processing of segments by SPEs is handled automatically by Sphere.

The local results' files produced by the SPEs, as well as the bucket files, are regular Sector files and can be processed by other UDFs.

4.2 Load Balancing

In addition to data locality, another important factor in performance is load balancing. One of the problems for distributed data parallel applications is that the data may be distributed nonuniformly over the slave nodes during certain stages of the processing.

Since Sphere is usually used to process very large data sets, there are usually many more data segments than SPEs. Because of this, the segments provide a natural mechanism for load balancing. A faster node (either because the hardware is better or because the data on the node require less time to process) can process more data segments. This simple load balancing strategy works well in practice even if the slave nodes are heterogeneous and have different processing power.

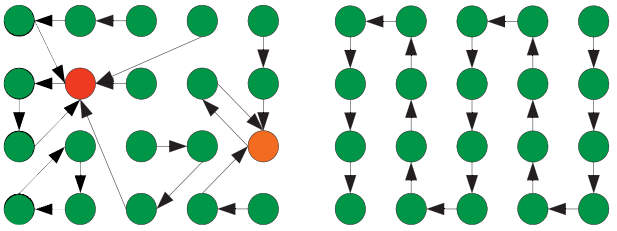

Fig. 3. This figure is a conceptual illustration of how hot spots can arise in a reduce style processing of data. There are hot spots in the left figure, but not in the right figure.

However, in general, Sphere needs to move some data segments to a node that would otherwise be idle. This strategy can be tricky because if the Sphere scheduler waits until the final stage of the computation to assign unfinished data segments to idle SPEs, there is likely to be an IO bottleneck. This is because there will be many idle SPEs reading segments from a relatively small number of still busy SPEs. This bottleneck can be bad enough that the strategy of using idle SPEs in this way can lengthen rather than shorten the computation. In the current implementation, the Sphere scheduler monitors the progress of each file and assigns its data segments to other SPEs as early as possible in order to decrease the likelihood of this type of bottleneck.

When bucket files are used (for example, in a Reduce or Hash/Bucket style processing), if too many records are sent to the same bucket at the same time, there may be a network bottleneck because there may not be enough network bandwidth available. As a simple example, when data are sent from multiple sources to the same destination node, congestion may occur on that node and cause a hot spot. Fig. 3 is a conceptual illustration of the problem. In the left illustration, the data transfers are such that there are several hot spots. On the right, the data transfers are such that there are no hot spots.

Of course, the diagram on the right in Fig. 3 is idealized. In practice, it is very difficult for a centralized scheduler to schedule the computation to avoid any hot spot. An alternative is to use a decentralized method to identify and remove hot spots caused by congestion.

In Sphere, the following decentralized approach is used to eliminate the hot spots. Before a node tries to send results to a bucket file, it queries the destination node and retrieves recent data transfer requests. If the aggregate size of the data transfer requests is greater than a threshold, the source node will attempt to process other buckets first.

With this distributed flow control algorithm, the Terasort benchmark (described in Section 5) was improved by approximately 10-15 percent for clusters of workstations ranging in size from 32 to 128 nodes.

Another approach to removing network bottlenecks in data centers is described in [6]. Note that this is different than removing bottlenecks that occur at hosts.

4.3 Fault Tolerance

As the number of slave nodes increases, so does the importance of fault tolerance. For those jobs that do not generate any bucket files, fault tolerance is relatively easy. If a slave node dies, then the required data segments can

simply be processed by another node. This feature is supported by Sphere and relies on the data replication provided by Sector.

When a UDF sends data to multiple bucket files on other nodes, fault tolerance is more complicated and can result in significant overhead. Because data are exchanged between source and destination nodes, a source node failure will cause all related destination nodes to contain incomplete data, while a destination node failure will lose data contributed by all the associated source nodes.

To provide fault tolerance in this situation, the UDF results sent to each bucket have to be duplicated. Due to the overhead introduced by supporting fault tolerance in this case, the total processing time will increase, which can mean an even higher chance of failure.

For this reason, Sphere does not directly provide fault tolerance for bucket files. There are two user-controlled approaches to support fault tolerance when bucket files are used.

First, users can split the UDF that generates bucket files into two sub-UDFs. The first sub-UDF generates local intermediate files that store the results from the local SPE. The second sub-UDF gathers all related files generated by the first one and places them into the final buckets. Fault tolerance can be provided for each sub-UDF as usual.

The second approach, which is not feasible in all situations, is to split the input into smaller substreams and to process each of the substreams independently. If a subtask fails, results from other subtasks will still be valid and only the failed subtasks need to be re-executed. Eventually, the bucket files of all the subtasks can be merged. Applications can specify the location of bucket files for the different subtasks so that the various bucket files are filled appropriately.

Another issue related to fault tolerance is how to detect poorly performing nodes. A poorly performing node can seriously delay the whole computation when bucket files are generated, since all the other nodes may need to wait for data from, or to, that node. Most fault tolerant systems can detect dead nodes with timeouts, but detecting poorly performing nodes remains a challenge. Hardware issues such as overheating, a hard disk error, problems with the NIC, etc., may cause poor performance. Networking issues can also cause nodes to perform poorly.

Sphere uses a voting system to identify and eliminate poorly performing nodes. Each node records the data transfer rate (ignoring idle time) to all the other nodes. A node is considered poorly performing if the data transfer rate to that node is in the lowest 10 percent. Periodically, each slave sends the list of the lowest performing 10 percent of nodes to the master. If one node gets more than 50 percent of the votes (i.e., more than 50 percent of the slave nodes have voted this node in their lowest 10 percent list) within a given time period, it is eliminated from the system. Note that votes will be cleared after a given time threshold, so this will not lead to a situation where every node will accumulate enough votes to be eliminated.

We found in the benchmarking studies described in Section 5 that poorly performing Hadoop nodes could result in experimental results that differed by 20-25 percent or more. In cases like these, we simply used a different set of nodes since we knew the amount of time the computation "should" take. In practice, poorly performing, but not dead, nodes can seriously impact performance for systems that are not designed to detect and remove them.

4.4 Multiple Inputs/Outputs

Sphere can also process multiple inputs if the input data files are located in the same directory structure. For example, suppose there are n input data sets, the first segment of each data set, input.1.1, $\mathrm{input.2.1}, \ldots, \mathrm{input.n.1}$, can be put together in one directory dir.1. Sphere can then accept dir.1, $\mathrm{dir.2}, \ldots, \mathrm{dir.m}$ as inputs and process all files in each directory at the same time.

The output can be organized into directories as well, which can be processed by other Sphere applications.

As described in Section 3.4, files in these directories will be kept together during replication.

4.5 Iterative and Combinative Processing

Some applications need to apply the same or similar operations on an input data set and its subsequent outputs. This is sometimes called iterative processing. One example is the K-Means clustering algorithm. The K-Means algorithm repeatedly computes the centers of multiple data clusters until the difference between the corresponding centers in Step k and Step $\mathrm{k}+1$ of the iteration is less than a threshold.

Iterative processing is straightforward to implement in Sphere and two Sector file system features can help improve the performance of iterative data processing applications.

First, Sector supports in-memory objects. If the intermediate data sets can fit in the physical memory, using in-memory objects can avoid disk IO and save time in reconstructing certain data structures.

Second, Sector allows developers to specify the location of each output bucket file. With certain applications, the locations can be carefully chosen in order to reduce data movement in the next round of processing.

These two features can significantly improve the performance of some iterative applications. For example, we used Sphere to do a breadth-first search on a graph with 30 M vertices and 270 M edges on a 20-node system. The performance with these two features enabled was about three times faster than the performance without using these two features.

The output locality feature can also benefit those applications in which multiple data processing jobs are combined with each other to reach a final result. One simple example is the "join" operation in databases (Fig. 4). To join two unsorted streams, both of them have to be scanned and the results sent to multiple bucket files determined by the keys. After the scan, the corresponding buckets files can be colocated by Sphere so that the next phase (which joins data from the corresponding bucket pairs) can read data from both data sets from the same location.

4.6 Sphere Programming Interface

In this section, we briefly describe the programming interface that developers use to write parallel data processing

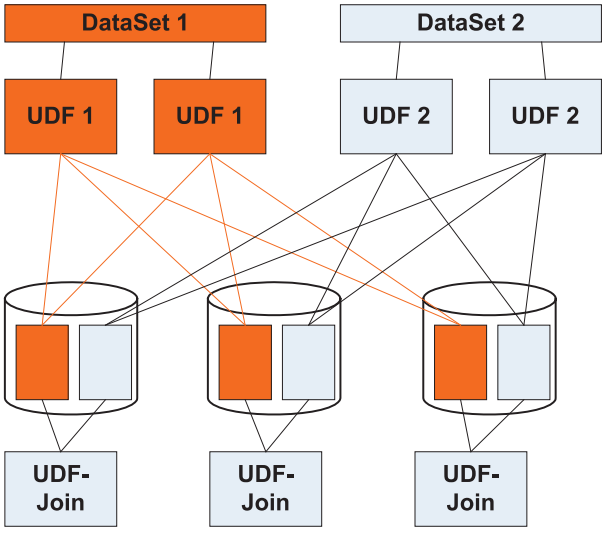

Fig. 4. Joining two large data sets with Sphere UDF. UDF1 and UDF2 are applied to each input data set independently and the results are sent to buckets on the same locations. The third UDF joins the bucket data.

applications. A Sphere application includes two parts: the client application and the UDF.

The client application specifies the input data sets, the output data sets, and the UDF. A typical client application is a simple program containing 100 to 200 lines of code. The data segmentation unit (which may be per record, per block, per file, or per directory) is specified by a parameter in the Sphere API. If the unit is records, an offset index for the input data set is required to exist in the Sector system.

A UDF is written as described in Section 4.1. The UDF processes the data passed to it by an SPE and puts the result in the "output" parameter. In particular, a UDF can be a simple wrapper around an existing application executable or system command (e.g., "grep").

A Sphere application developer does not need to write any code to support data segmentation (except for specifying the segmentation unit), load balancing, and fault tolerance. These functionalities are provided by Sphere automatically.

Sphere also supports a command line tool called "stream," which can execute simple Sphere jobs without having to write any code. The "stream" tool accepts an input file/directory, an optional output file/directory, a data processing command, and any required parameters. For example, the following command will run "grep 'my data'" on all files in the "html" directory on the Sector file system.

```
stream −i html −c grep −p ''my data''
```

The "stream" tool automatically generates a Sphere client application and a UDF that wraps the command.

For applications that generate bucket files, Sphere requires that the command or application to output each record be in a specific format so that the bucket ID can be parsed.

5 Experimental Results

Previously, we have conducted experimental studies on the wide area Open Cloud Testbed [9], [18]. We present here several experiments conducted on four racks within the same data center. This basically removes the performance gains that UDT provides over wide area networks. Each of the four racks contains 30 nodes. The nodes contain a dual single Intel Xeon 5410 Quad Core CPU, 16 GB of memory, four 1-TB SATA disks in a RAID-0 configuration, and a 1-GE NIC. GNU/Debian Linux 5.0 is installed on each node.

TABLE 1
Comparing Hadoop and Sector Using the MalStone Benchmark

	MalStone A	MalStone B
Hadoop	454m 13s	840m 50s
Hadoop Streaming/Python	87m 29s	142m 32s
Sector/Sphere	33m 40s	43m 44s

5.1 MalStone Benchmark

The first benchmark is MalStone. MalStone is a stylized analytic that runs on synthetic data generated by a utility called MalGen [1]. MalStone records contain the following fields:

```
Event ID | Timestamp | Site ID | Entity ID | Flag
```

A record indicates that an entity visited a site at a certain time. As a result of the visit, the entity may become marked, which is indicated by setting the flag to 1 (otherwise, it is 0). The MalStone A-10 and B-10 benchmarks each consist of 10 billion records and the timestamps are all within a one year period. There are also MalStone benchmarks for 100 billion and one trillion records. The MalStone A benchmark computes a ratio for each site w as follows: for each site w, aggregate all entities that visited the site at any time, and compute the percent of visits for which the entity becomes marked at any future time subsequent to the visit.

MalStone B is similar, except that the ratio is computed for each week d: for each site w, and for all entities that visited the site at week d or earlier, the percent of visits for which the entity became marked is computed.

MalStone represents a typical data intensive application that systems such as Sphere and MapReduce are designed to support. The benchmark is designed to measure the performance of systems optimized for the batch processing of large numbers of records that must be scanned. Databases are usually not optimized for these types of applications.

Table 1 lists the results of three different implementations: 1) Hadoop with Malstone computed using MapReduce; 2) Hadoop streaming with MalStone computed using Python; and 3) Sector with MalStone computed using Sphere. All the benchmarks were computed on a single cluster of 20 nodes. The nodes contained an Intel Xeon 5160 3.0 GHz Quad Core CPU, 12 GB of memory, a single 1 TB SATA disk, and a single 1GE NIC. Version 1.21 of Sector and Version 0.18 of Hadoop were used.

5.2 Terasort Benchmark

We also compared Hadoop and Sector using Terasort. Recall that the Terasort benchmark consists of sorting 1 TB of data containing a 10-byte key and a 90-byte payload

TABLE 2
Comparing Hadoop and Sphere Using the Terasort Benchmark

Number of Racks (Nodes)	Sphere UDF	Sphere MR (Push)	Sphere MR (Pull)	Hadoop
1 (30)	28m49s	31m10s	46m2s	85m49s
2 (60)	15m20s	15m41s	25m7s	37m0s
3 (90)	10m19s	12m30s	16m35s	25m14s
4 (120)	7m56s	-	-	17m45s

value. Version 1.24a of Sector and Version 0.20.1 of Hadoop were used.

Table 2 lists the performance of sorting 1 TB of data, consisting of 100-byte records with a 10-byte key, on one, two, three, and four racks (i.e., 30, 60, 90, and 120 nodes).

The performance is consistent with the results of our previous experiments using Sector Version 0.21 and Hadoop Version 0.18 [7].

Furthermore, in this experiment, we also examined the resource usage of both systems. We noticed that network IO plays an important role in Terasort. When Sector is running on 120 nodes, the aggregate network IO is greater than 60 Gb/s, while for Hadoop the number is only 15 Gb/s.

Because sorting the 1 TB requires exchanging almost the entire data set among the participating nodes, the higher network IO is an indication that resources are being utilized effectively. This may explain in part why Sector is more than twice as fast as Hadoop.

Neither Sector nor Hadoop fully utilized the CPU and memory resources in this application because the application is still IO bound. However, Hadoop used much more CPU (200 percent versus 120 percent) and memory (8 GB versus 2 GB). This difference may be due to Hadoop's use of the Java VM.

For both experiments, we limited the number of replicas to 1 (i.e., neither system created replicas). If replicas had been enabled, Hadoop's performance drops approximately 50 percent (in other words, the runtime doubles) [9]. Even with replication enabled, Sphere's performance is approximately the same. This difference is due to the different strategies Hadoop and Sphere use for replication, as discussed above (Sphere does not replicate the result immediately, but rather periodically).

We note that in our experience, tuning Hadoop to achieve optimal performance took some time and effort. In contrast, Sector does not require any performance tuning.

In another series of experiments, we implemented Terasort using using a Sphere-based MapReduce with a push model, and using a Sphere-based MapReduce using a pull model. The MapReduce with Mappers pushing to Reducers has roughly the same performance as using Sphere UDFs. This is to be expected because in both cases, two UDFs are used and in both cases data are pushed between the two UDFs.

In the MapReduce/Pull model, where Mappers generate local results and Reducers read data from different locations, the performance drops by 60-75 percent. This is not only because data are written into local disk (an extra write compared to the push model), but also because the Sphere MapReduce pull implementation does not support overlap between Map and Reduce, so the data transfer is an extra step compared to the push model.

The Sphere MapReduce pull implementation is still faster than Hadoop MapReduce. This is probably due to a combination of many implementation factors, including but not limited to: system architecture, choice of programming language (C++ versus Java), and selection of a large number of different parameters (buffer sizes, etc.). Profiling the Hadoop implementation is beyond the scope of this paper.

The Sphere MapReduce results were not available for the 4-rack configuration. This is because the fourth rack was assigned to another project after the Sphere UDF and Hadoop experiments were completed.

5.3 Illumina Genome Application

We end this section by describing how we used Sphere to parallelize the processing of gene expression data from the Illumina Genome Analyzer [19]. Each run of the Illumina we used produces several hundred thousand TIF files comprising a few TB of data. With the current pipeline [19], these are processed as follows: the images are analyzed using a program called Firecrest; DNA bases are then called using a program called Bustard; and finally, additional quality calibration and filtering is done using a program called Gerald. This results in several tens of thousands of files comprising 20-30 GB of data. For more details, see [19].

Rather than recode the Firecrest-Bustard-Gerald pipeline entirely in Sphere, we simply took the data as output by the Illumina Genome Analyzer and wrapped the existing code in Sphere wrappers. The input to the pipeline consists of many directories each containing a variety of files and other directories.

Recall that Sector has a feature that enables files in a directory to be processed together. Specifically, Sector uses a system file in each directory to prevent the directory from being split during replication. Also, a Sphere UDF looks for a special "seed" file in each directory to locate the required input data.

The Illumina instrument has eight "lanes" and to simplify our experiment, we simply used a different node to process the data for each lane. Proceeding in this way, the Sphere eight-way parallelized version required about 110 minutes to finish processing the data, while the original application took approximately 760 minutes to finish using just one of the nodes. We emphasize that the pipeline was not changed; rather, the data, which were already in Sector, were simply processed with the pipeline invoked from Sphere. Note that parallel workflow systems such as Swift [14] can be used in very similar ways.

Note that this example is easy to code using Sector/Sphere, but much more challenging to code efficiently using the Hadoop framework since images are split into blocks and directories of files cannot be easily processed together.

6 Related Work

We begin this section by comparing Sector/Sphere with GFS/MapReduce and Hadoop. Sector/Sphere is broadly

similar to GFS/MapReduce and Hadoop since they are all designed with similar objectives: processing very large data sets on distributed clusters of unreliable computers. The MapReduce programming model has also been used in other contexts, such as on GPU and multicore CPUs, but these implementations are irrelevant to the work presented in this paper.

As we have described above, Sector/Sphere is designed differently than GFS/MapReduce and Hadoop and has different features. Sector is tightly coupled with Sphere and contains features so that Sphere applications can provide directives to Sector to improve data locality. These features include: directory/file family support (Section 3.2) and in-memory objects (Section 4.5). Also, Sector is file-based instead of block-based (Section 3.1), which also improves data locality. In contrast, both GFS and HDFS are more similar to a traditional distributed file system, although they both support replication.

We note that Sphere's support for UDFs is more general than the MapReduce framework. Specifically, a sequence of Sphere UDFs can duplicate any MapReduce computation. In more detail: a Map operation can be simply expressed by a UDF. A MapReduce operation can be expressed by two UDFs with the first one generating bucket files (similar to a Mapper) and the second one processing the buckets files (similar to a Reducer). It is also important to note that in MapReduce, a sort must be performed in order to run Reduce, even though sorting is not required for many applications. Finally, we note that in the UDF model, developers are free to use any operations on the bucket files, whether it is sorting, hashing, or no processing at all.

For many applications, Sphere's support for multiple input files is also important. Sphere UDFs not only accept multiple inputs, but also can do this in a way that supports data locality. In contrast, with Hadoop, there is no way to guarantee the data locality for multiple inputs, as the input files can be randomly located across many different nodes.

It is also very straightforward for Sphere developers to choose a pull model or a push model for transferring data between the two UDFs that assemble MapReduce style operations. The push model sends data directly to the bucket files, which is faster but does not provide fault tolerance.

Finally, as we have shown in Section 5, Sector/Sphere provides higher performance than Hadoop as measured by the Terasort and Malstone benchmarks.

We now briefly compare Sector/Sphere to some other systems that have been used for data intensive computing.

Grid-based frameworks have been used for data intensive computing for some time, including Condor [16] and Swift [14]. Condor and Swift specify the execution of tasks using a directed acyclic graph (DAG). A node in a DAG represents a process (sometimes called a subroutine) and is roughly comparable to a UDF in Sphere. The processes in a DAG can be organized in more flexible and complex ways than the simple data parallel execution patterns that MapReduce or Sphere UDFs support; on the other hand, this increase in flexibility requires more of the developers' input. While this means a DAG-based approach can support a larger class of applications, it also places a greater burden on the programmers.

More recently, Dryad [11] has emerged as a more general framework for parallel computing that provides language level constructs that generalize MapReduce style computations. Perhaps, the most fundamental difference between Condor/Swift/Dryad and MapReduce/Sphere is data locality. In the former systems, processors are still first-class citizens and data locality is only considered as an optimization method. In MapReduce and Sphere, maintaining data locality is one of the fundamental design principles.

There has also been an interest in data locality for grid systems. Systems such as BAD-FS [3], Stork [12], and iRods [15] all consider data placement and transfer cost as a factor in batch task schedulers. However, these systems use a multitask model and, more importantly, the storage is still separated from the computing nodes.

DataCutter [2] and Active Data Repository (ADR) [13] actually support data processing on storage nodes and the "filter" used in these systems can be broadly compared to the Sphere UDF. However, DataCutter and ADR do not provide the support for transparent fault tolerance and load balancing that Hadoop and Sector do.

7 Conclusion

We have presented a new parallel data processing framework for data intensive computing over large clusters of commodity computers. The Sector distributed file system supports data locality directives so that applications can improve their performance by exploiting data locality. Sector processes segments of data at various levels of granularity, including by record, by groups of records, by files, and by directories of files. The Sphere framework allows User-Defined Functions to be applied in parallel to the data segments managed by Sector. Sphere also supports what are called bucket files that provide a flexible mechanism for different UDFs to exchange data.

We have reported on experimental studies comparing Sector/Sphere with Hadoop. In these studies, Sector/Sphere is about two to four times faster than Hadoop as measured by the Terasort and MalStone benchmarks. Sector/Sphere supports multiple inputs, multiple outputs, and provides several directives so that users can specify data locality. Hadoop does not support these features.

Appendix

Example Sphere Code for Distributed Sorting

This Sphere example shows a Sphere program to perform a distributed sorting on data stored in Sector. The data to be sorted consist of 100-byte long binary records with a 10-byte key. To save space, we removed the error checking code in the program. This program contains two UDFs and one client application that organizes input, output, and UDFs. The first UDF reads each record and assigns it a bucket ID based upon the high order bits of the key. Note that when bucket IDs are assigned in this way, all values in the nth bucket are greater than any value in the n-1 bucket. The second UDT then simply sorts each bucket.

UDF-1. The input is a record, the output is the same record with a bucket ID computed according to the key.

```
struct Key {
    uint32_t v1;
```

```
    uint32_t v2;
    uint16_t v3;
};

// hash k into a value in [0, 2^n - 1), n < 32
int hash(const Key* k, const int& n) {
    return (k->v1 >> (32 - n));
}

int sorthash(const SInput* input, SOutput* output, SFile*
  file) {
    memcpy(output->m_pcResult, input->m_pcUnit, 100);
    *(output->m_pllIndex) = 0;
    *(output->m_pllIndex + 1) = 100;
    output->m_iResSize = 100;
    output->m_iRows = 1;
    *output->m_piBucketID
      = hash((Key*)input->m_pcUnit,
        *(int*)input->m_pcParam);
    return 0;
}
```

UDF-2. Sort each bucket file.

```
void sortbucket(const char* bucket) {
    // bucket is a local file. This function sorts all records
        in the bucket file and writes the result back to the
        same file
   // we omit the sorting code here. It is independent of
       Sphere.
}

int sort(const SInput* input, SOutput* output, SFile* file) {
    string bucket = file->m_strHomeDir + input->m_pcUnit;
    sortbucket(bucket.c_str());
    output->m_iRows = 0;
    file->m_sstrFiles.insert(input->m_pcUnit);
    return 0;
}
```

Sphere client program.

```
int main(int argc, char** argv) {
   // Sphere client initialize and login the system. Code
       omitted here.

   // initialize input data with all files in the "data" directory
   vector<string> files;
   files.push_back("data");
   SphereStream s;
   s.init(files);

   // prepare output stream with 1,024 buckets, use the
       first 10-bit as bucket ID
   int N = 10;
   SphereStream temp;
   temp.setOutputPath("test/sorted",
     "stream_sort_bucket");
   temp.init(pow(2, N));

   SphereProcess* myproc = client.createSphereProcess();

   // upload the two UDFs to the Sector system
   myproc->loadOperator("./funcs/sorthash.so");
   myproc->loadOperator("./funcs/sort.so");

   // run UDF-1
   myproc->run(s, temp, "sorthash", 1, (char*)&N, 4);
   myproc->waitForCompletion();

   SphereStream output;
   output.init(0);

   // run UDF-2
   myproc->run(temp, output, "sort", 0, NULL, 0);
   myproc->waitForCompletion();

   // Sphere client logout and release resources. code
       omitted here.
   return 0;
}
```

ACKNOWLEDGMENTS

The Sector/Sphere software system is funded in part by the US National Science Foundation (NSF) through Grants OCI-0430781, CNS-0420847, ITR-0325013, and ACI-0325013. Collin Bennett and Jonathan Seidman of Open Data Group performed the experimental studies of MalStone described in Section 5. A shorter version of this paper has been published at the Second Workshop on Many-Task Computing on Grids and Supercomputers (MTAGS 2009).

REFERENCES

[1] C. Bennett, R.L. Grossman, D. Locke, J. Seidman, and S. Vejcik, "MalStone: Towards a Benchmark for Analytics on Large Data Clouds," *Proc. 16th ACM SIGKDD Int'l Conf. Knowledge Discovery and Data Mining (KDD)*, 2010.

[2] M.D. Beynon, T. Kurc, U. Catalyurek, C. Chang, A. Sussman, and J. Saltz, "Distributed Processing of Very Large Data Sets with DataCutter," *J. Parallel Computing*, vol. 27, pp. 1457-1478, 2001.

[3] J. Bent, D. Thain, A. Arpaci-Dusseau, and R. Arpaci-Dusseau, "Explicit Control in a Batch-Aware Distributed File System," *Proc. First USENIX/ACM Conf. Networked Systems Design and Implementation*, Mar. 2004.

[4] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," *Proc. Sixth Symp. Operating System Design and Implementation (OSDI '04)*, Dec. 2004.

[5] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The Google File System," *Proc. 19th ACM Symp. Operating Systems Principles*, Oct. 2003.

[6] A. Greenberg, J.R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D.A. Maltz, P. Patel, and S. Sengupta, "VL2: A Scalable and Flexible Data Center Network," *Proc. ACM SIGCOMM*, 2009.

[7] Y. Gu and R. Grossman, "Exploring Data Parallelism and Locality in Wide Area Networks," *Proc. Workshop Many-Task Computing on Grids and Supercomputers (MTAGS)*, Nov. 2008.

[8] Y. Gu and R. Grossman, "UDT: UDP-Based Data Transfer for High-Speed Wide Area Networks," *Computer Networks*, vol. 51, no. 7, pp. 1777-1799, May 2007.

[9] Y. Gu and R. Grossman, "Sector and Sphere: The Design and Implementation of a High Performance Data Cloud," *Theme Issue of the Philosophical Trans. Royal Soc. A: Crossing Boundaries: Computational Science*, vol. 367, no. 1897, pp. 2429-2445, 2009.

[10] Y. Gu, R.L. Grossman, A. Szalay, and A. Thakar, "Distributing the Sloan Digital Sky Survey Using UDT and Sector," *Proc. Second IEEE Int'l Conf. e-Science and Grid Computing,* 2006.
[11] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly, "Dryad: Distributed Data-Parallel Programs from Sequential Building Blocks," *Proc. European Conf. Computer Systems (EuroSys),* Mar. 2007.
[12] T. Kosar and M. Livny, "Stork: Making Data Placement a First Class Citizen in the Grid," *Proc. 24th IEEE Int'l Conf. Distributed Computing Systems (ICDCS '04),* Mar. 2004.
[13] T. Kurc, U. Catalyurek, C. Chang, A. Sussman, and J. Salz, "Exploration and Visualization of Very Large Data Sets with the Active Data Repository," Technical Report CS-TR4208, Univ. of Maryland, 2001.
[14] I. Raicu, Z. Zhang, M. Wilde, I. Foster, P. Beckman, K. Iskra, and B. Clifford, "Toward Loosely Coupled Programming on Petascale Systems," *Proc. 20th ACM/IEEE Conf. Supercomputing,* 2008.
[15] A. Rajasekar, M. Wan, R. Moore, and W. Schroeder, "Prototype Rule-Based Distributed Data Management System," *Proc. HPDC Workshop Next Generation Distributed Data Management,* May 2006.
[16] D. Thain, T. Tannenbaum, and M. Livny, "Distributed Computing in Practice: The Condor Experience," *Concurrency and Computation: Practice and Experience,* vol. 17, nos. 2-4, pp. 323-356, Feb.-Apr. 2005.
[17] Hadoop, hadoop.apache.org/core, Retrieved in, Oct. 2009.
[18] The Open Cloud Testbed, http://www.opencloudconsortium.org, 2011.
[19] Pipeline Documentation of the Illumina Genome Analyzer, watson.nci.nih.gov/solexa, 2010.
[20] F. Chang, J. Dean, S. Ghemawat, W.C. Hsieh, D.A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R.E. Gruber, "Bigtable: A Distributed Storage System for Structured Data," *OSDI '06: Seventh Symp. Operating System Design and Implementation,* Nov. 2006.

Yunhong Gu received the BE degree from Hangzhou Dianzi University, China, in 1998, the ME degree from Beihang University, China, in 2001, and the PhD degree from the University of Illinois at Chicago, in 2005, all in computer science. He is a research scientist at the Laboratory for Advanced Computing at the University of Illinois at Chicago. His research interests include high performance networking, distributed storage and computing, and cloud computing. He is also an open source developer and is the architect and lead developer of the open source projects UDT and Sector/Sphere.

Robert Grossman received the AB degree from Harvard University in 1980 and the PhD degree from Princeton University in 1985. He is the director of the Laboratory for Advanced Computing and a professor of mathematics, statistics, and computer science at the University of Illinois at Chicago. He is also the managing partner of Open Data Group. His research interests include cloud computing, data intensive computing, bioinformatics, predictive modeling and data mining, and high performance networking. He has published more than 150 research papers.