Lessons Learned From a Year's Worth of Benchmarks of Large Data Clouds

Yunhong Gu and Robert L Grossman*
Laboratory for Advanced Computing, University of Illinois at Chicago

yunhong@lac.uic.edu, grossman@uic.edu

ABSTRACT

In this paper, we discuss some of the lessons that we have learned working with the Hadoop and Sector/Sphere systems. Both of these systems are cloud-based systems designed to support data intensive computing. Both include distributed file systems and closely coupled systems for processing data in parallel. Hadoop uses MapReduce, while Sphere supports the ability to execute an arbitrary user defined function over the data managed by Sector. We compare and contrast these systems and discuss some of the design trade-offs necessary in data intensive computing. In our experimental studies over the past year, Sector/Sphere has consistently performed about 2 – 4 times faster than Hadoop. We discuss some of the reasons that might be responsible for this difference in performance.

Categories and Subject Descriptors

C.4 [**Computer System Organization**]: Performance of System

General Terms

Performance, Experimentation

Keywords

Cloud Computing, Data Intensive Computing, High Performance Computing, Grid Computing, MapReduce, Multi-Task Computing

1. INTRODUCTION

Instruments that generate data have increased their capability following Moore's Law just as computers have. The result of this is a gap between the amount of data that is being produced and the capacity of current systems to store and analyze this data.

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

The result is that analyzing a large scientific dataset might require analyzing hundreds of TB or more of data. A current rack of commodity computers might have four 1 TB disks per computer and 32 computers per rack. The challenge is to develop a framework to support data intensive computing that provides persistent storage for large datasets (that require multiple racks to store) as well as balanced computing so that this persistent data can be analyzed.

*Robert Grossman is also the Managing Partner at Open Data Group.

To say 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 this 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 integrate the storage system and the processing system tightly is because data locality is the fundamental principle underlying the efficient processing of very large datasets. 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 1000-node system would require aggregate data IO speed at TB/s in order to achieve a performance comparable to a single node with internal disks.

Supercomputer systems as generally deployed store data in external storage (RAID, SAN, NAS, etc.) and load data into a processing system consisting of a very large number of processors. The data transfer channel between the external storage system and the computing system can be a very serious bottleneck.

Google has developed an internal proprietary storage system called the Google File System (GFS) [6] and an associated processing system called MapReduce [5] that has very successfully integrated large scale storage and data processing. Hadoop [15] is an open source implementation of the GFS/MapReduce design. Other earlier attempts to integrate data storage and data processing for large data sets include DataCutter [2, 12], BAD [3] and Stork [11]. Other systems that support scheduling for multi-task applications [17] are Condor [14] and Falkon [13].

Hadoop 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 many technical challenges that need to be addressed in order to improve its performance and usability. Hadoop was designed for processing web data and it is not surprising that its performance is less impressive for certain types of scientific applications.

During the last three years, we have designed and implemented a scalable storage system called Sector and an associated data processing system 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. In this paper, we identify these differences and investigate the impact of these design choices on performance and usability. The paper also includes

some experimental studies that we have performed during the past year and some lessons learned.

We start with a brief description of the Sector/Sphere system in Sector 2. The Hadoop system follows the design of GFS [6] and MapReduce [5]. We compare the file system design between Sector and Hadoop in Section 3 and the data processing design of Hadoops's MapReduce and Sphere in Section 4. In Section 5 we discuss how Hadoop and Sector/Sphere can interoperate. We describe the results from our most recent experimental studies in Section 6 and summarizes the lessons learned in Section 7. The paper is concluded in Section 8 with a brief look at future work.

2. SECTOR/SPHERE

Sector is a distributed file system (DFS) and Sphere is a parallel data processing engine designed to work with data managed by Sector. In contrast to most traditional distributed file systems that require hardware to provide fault tolerance, Sector implements fault tolerance by replicating data in the file system and managing the replicas. 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.

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.

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].

Sphere, on the other hand, 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 (UDF) to run on each data unit (a record, block, file, or directory). 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.

Figure 1 illustrates the architecture of Sector. For readability, in the rest of this paper, we usually 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. One or more master servers can be started. The master servers are responsible for maintaining the metadata of the storage system and for scheduling requests from users. The slave nodes are the computers that actually store and process the data. The slaves are racks of commodity computers with internal disks. A client is the user's computer that issues requests to the Sector system and accepts responses.

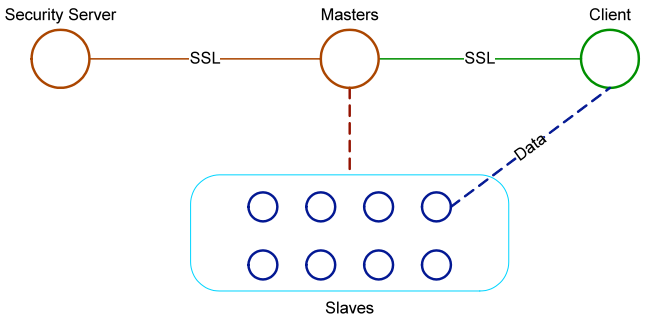

Figure 1: Sector/Sphere System Architecture.

Sector/Sphere is open source software developed in C++. It is available from sector.sf.net.

3. FILE MANAGEMENT STRATEGY

3.1 Block vs. File

Like the Hadoop Distributed File System (HDFS), the Sector Distributed File System (SDFS) stores files on the local file system on all participating nodes. The HDFS splits files into fixed sized blocks and scatters the blocks over multiple servers. In contrast, the SDFS does not split files into blocks. A dataset in the SDFS can consist of multiple files.

To summarize: a dataset in the SDFS consists of multiple files that are not split into blocks. A dataset in HDFS consists of a single file that is split into multiple blocks.

The SDFS does not split a dataset into multiple files automatically. Instead, the user must do this. For some applications, this can create a small amount of additional work. In practice though, either a large dataset is already split into multiple files or this can be done with little effort.

We noticed in several of our experimental studies that in order to achieve optimal performance with Hadoop we needed to increase the block size so that an entire Hadoop file could fit into a single block and was therefore not split. Notice that this effectively reduces to the same strategy that Sector employs.

In general, the SDFS scatters the files of a Sector dataset over all the nodes in a cluster. Some applications though require that a certain collection of files be co-located in a single directory. To support this, Sector will keep all the files in directory together if a special file is put into the directory.

3.2 Replication

Both Sector and Hadoop use replication to provide data reliability and availability, and in certain cases, improve performance. Hadoop is designed to work within a data center, while Sector is designed to work over multiple data centers over wide area networks. This does not prevent Sector from being used within a single data center.

Sector chooses slave nodes so that replications of the same file are as far from each other as possible. This increases data reliability because even if a complete data center goes offline, the data can still be retrieved from other data centers. Moreover, this strategy allows a client to choose a nearby replication in a wide area system, which generally results in better data transfer throughput.

3.3 Data Transfer

To support wide area clouds, Sector not only uses a different replication strategy, but also uses a high speed transport protocol called UDT [9]. We choose to use UDT instead of TCP since UDT has proven to be better than TCP when moving large datasets over wide area high performance networks [9].

Sector supports the option of encrypting data when transporting it. Hadoop does not currently support encryption.

4. UDF vs. MapReduce

4.1 The UDF Model

We use the term data segment to refer to a portion of a Sector dataset. 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 user defined function (UDF) independently to data segments, which are usually scattered over multiple Sector slave nodes. The input to a Sphere UDF is a Sector dataset, as is the output. Recall that a Sector dataset consists of multiple files.

Consider the following serial program for processing data:

```
for (int i = 0; i < total_segment_num; ++ i)
        UDF(segment[i]);
```

In Sphere, the UDF is uploaded to slave nodes and each slave node calls this UDF to process all its local data segments.

Sphere allows a UDF to specify an ID for each output record. This is usually called a bucket ID and is a standard technique when processing data in parallel. All records with the same bucket ID are placed by Sector into the same bucket, which in Sector is a file. The resulting bucket file may or may not be on the same node where the data is processed, since different slaves may generate data with the same bucket IDs.

Sphere's computing paradigm can be thought of as a generalization of MapReduce. When a Sphere UDF generates local data only, it can be thought of as a Map process. When a Sphere UDF generates bucket files, it can be thought of as a Reduce process.

It is important to note that MapReduce can only processes data records, while Sphere's UDFs can process Sector data segments, which can be records, contiguous collections of records, files, or directories of files. When the data segments are directories, it enables Sphere to process multiple input streams, since each directory can contain multiple files from different datasets.

Here is an example. Sphere can process the following serial program in parallel over multiple slave nodes:

```
for (int i = 0; i < total_segment_num; ++ i)
        UDF(segment_A[i], segment_B[i]);
```

Hadoop's MapReduce always performs a sort operation before it merges records with the same key. In contrast, Sphere can run any UDF on the bucket files and perform a sort only when necessary. For applications that do not require a sort, this flexibility can be quite useful.

Another difference between Hadoop and Sector is how datasets are divided into records. Sphere uses a record offset index for each data file in order to parse the file into data records. The index contains the offset and size of each record in the file. When an index is not present, the minimum processing unit is a file or a directory. In contrast, MapReduce uses a user defined parser that is invoked at run time to parse the data file into data records.

Table 1 compares the steps when processing data using Hadoop's MapReduce and Sphere's UDFs.

Table 1. Comparing the steps in MapReduce and Sphere

Sphere	MapReduce
Record Offset Index	Parser / Input Reader
UDF	Map
Bucket	Partition
-	Compare
UDF	Reduce
-	Output Writer

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 during certain stages of the processing data, the data may be distributed over the slave nodes non-uniformly.

Both Sphere and Hadoop split jobs into relatively small data segments so that a faster node (either because the hardware is better or because the data on the node requires less time to process) can process more data segments. This simple strategy works well even if the slave nodes are heterogeneous and have different processing power.

However, Sphere and Hadoop may need to move some data segments to a node that would otherwise be idle. Both systems try to move data to the nearest available node in order to minimize data traffic.

When processing results need to be sent to bucket files (e.g., in a Reduce style processing), the available network bandwidth may become the bottleneck. As a simple example, when data is sent from multiple sources to the same destination node, congestion may occur on that node and cause a hot spot. Figure 2 illustrates the problem. In the top illustration, the data transfers are such that there are several hot spots (in red). In the bottom illustration, the data transfer is such that there are no hot spots.

Of course, the bottom diagram in Figure 2 is idealized. In practice, it is very difficult for a centralized scheduler to schedule the computation so that there are no hot spots. An alternative is to use a decentralized scheme 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.

There are some research projects designed to removing network bottlenecks in data centers [7]. Note that this is different than removing bottlenecks that occur at hosts.

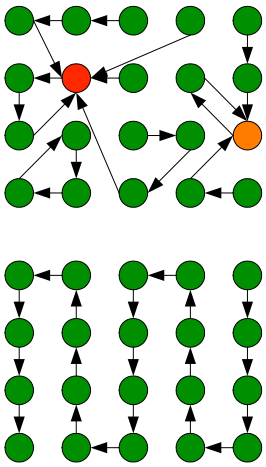

Figure 2. Data movement in reduce style processing.

There are hot spots in the top figure, but not in the bottom figure.

4.3 Fault Tolerance

Finally, as the number of slave nodes increases, so does the importance of fault tolerance. In a Map style processing, fault tolerance is fairly easy. A failed data segment will be re-processed on another node, if the failure is caused by the failure of the slave node hardware, rather than by problems with the data or the UDF. This feature relies on data replication provided by Sector.

In a Reduce style processing, 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 is 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 related source nodes.

To provide fault tolerance in this situation, the intermediate data needs to be replicated. Ironically, due to the overhead introduced by fault tolerance, the total processing time will increase, which can mean a higher chance of failure.

For this reason, Sphere does not currently provide fault tolerance for Reduce style processing. One option under investigation is to find a method that minimizes the cost of re-processing a reduce when a failure occurs. One possible solution is to split the input into smaller sub-streams and process them independently. If one fails, results from other sub-tasks will still be valid and only the failed sub-tasks will need to be re-executed. Eventually, the bucket files of all sub-tasks can be merged.

Another issue related to fault tolerance is how to detect poorly performing nodes. A poorly performing node can seriously delay the whole computation if Reduce or bucket style data processing is involved, since in this case all other nodes may need to wait for data from or to that node. Most fault tolerant systems can detect dead nodes, but detecting poorly performing nodes remains a challenge. Hardware issues such as overheating, hard disk error, problems with the NIC, etc. may cause poor performance. Problems with the network are other common reasons that nodes perform poorly.

Sector uses a voting system to eliminate bad nodes. Each slave node periodically sends performance statistics to the master node. In particular, each node sends the data transfer rate (ignoring idle time) to all the other nodes. Voting is used to identify poorly performing nodes (a node is considered poorly performing if it is in the lowest 10%). If one node gets more than 50% of the votes, it is eliminated from the system.

4.4 Streaming

The name "Streaming" is used by Hadoop for a utility that is able to run applications or system commands directly using a MapReduce style of processing. This greatly increases the usability of Hadoop because in many cases users no longer need to write Java code but instead can simply pass binaries of existing applications to Hadoop.

Sphere provides a similar utility. Sphere streaming accepts a system command or application executable and runs it as a UDF. The streaming utility automatically generates a C++ wrapper that calls the system command or application executable and processes the specified data stored in Sector.

For bucket-based processing (e.g., Reduce operation), Sphere requires the command or application to put the bucket ID as the first value of each output record. Future versions of Sector will provide additional mechanisms for specifying the bucket ID.

5. INTEROPERABILITY

While there are many differences in design and implementation of Sector and Hadoop, it is possible to interoperate the two systems. For example, Hadoop's MapReduce can run on top of the SDFS, with a Sector interface for Hadoop. As another example, running Sphere on top of HDFS is also possible. In this case, blocks may need to be moved during a Sphere process.

It is also possible to write applications that can run on both Sector and Hadoop. Currently this is limited to applications that can be supported by Hadoop streaming and Sphere streaming, both of which can accept application binaries (including system commands) directly and run them in a predefined framework.

More discussions and related work can be found at http://code.google.com/p/cloud-interop/.

6. EXPERIMENTAL RESULTS

Previously we have conducted experimental studies on the wide area Open Cloud Testbed [16]. We present here two experiments conducted on racks within the same data center. This basically removes the performance gains that UDT provides over a wide area high performance networks.

MalStone is a benchmark that 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. 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 each week *d*, and computes: 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. MalStone A-100, A-1000, etc. and MalStone B-100, B-1000, etc. are similar except the dataset consists of 100 Billion, 1 Trillion, etc. records.

Table 2 lists the results of three different implementations: 1) Hadoop; 2) Hadoop streaming with Python code implementing MalStone; 3) Sector/Sphere. The results are obtained from a single cluster of 20 nodes. The nodes contain an Intel Xeon 5160 3.0 GHz Quad Core CPU, 12GB memory, a single 1TB SATA disk, and a single 1GE NIC. Version 1.21 of Sector and Version 0.18 of Hadoop were used.

Table 2. MalStone Benchmark for Sphere and Hadoop

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

We also compared Hadoop and Sector using Terasort running on 4 racks within the same data center. We used 30 nodes of each rack for the test. The nodes contain a single Intel Xeon 5410 Quad Core CPU, 16GB memory, a 1 TB SATA disk in a RAID-0 configuration, and a 1 GE NIC. GNU/Debian Linux 5.0 was installed on each node. Version 1.24a of Sector and Version 0.20.1 of Hadoop were used.

Table 3 lists the performance of sorting 1 TB of data, consisting of 100-byte records with a 10-byte key, on 1, 2, 3, and racks (i.e., 30, 60, 90, and 120 nodes).

Table 3. MalStone Benchmark for Sphere and Hadoop

Number of Racks	**Sphere**	**Hadoop**
1	28m 49s	85m 49s
2	15m 20s	37m
3	10m 19s	25m 14s
4	7m 56s	17m 45s

The performance is consistent with the results of our previous experiments using Sector Version 0.21 and Hadoop Version 0.18 [8], although both systems have been improved significantly since then.

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 complete dataset among all the participating nodes, the higher network IO is an indication that resources are being utilized effectively. This may explain why Sector is over 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% vs. 120%) and memory (8GB vs. 2GB). This is probably caused by the Java VM.

Tuning Hadoop to achieve optimal performance can take some time and effort. In contrast, Sector does not require any performance tuning.

7. LESSONS LEARNED

In this section, we summarize some of the lessons we have learned from our experimental studies over the past year.

The importance of data locality. It is well known that locality is the key factor to support data intensive applications, but this is especially important for systems such as Sector and Hadoop that rely on inexpensive commodity hardware. More expensive specialized hardware can provide higher bandwidth and lower latency access to disk.

Generalizations of MapReduce. MapReduce has quickly emerged as one of the most popular frameworks for data intensive computing. Both the Map operation and the Reduce operation have been used previously in parallel computing; the reason for their current popularity is the combination of being easy to use and their proven ability to be useful for an unexpectedly large number of applications. From our experience to date, Sector's ability to apply a UDF to the data managed by a distributed file system is also very easy to use and also very broadly applicable. A MapReduce can be realized as an easy special case.

Load balancing and the importance of identifying hot spots. In a system with hundreds or thousands of commodity nodes, load balancing is very important – with poor load balancing, the entire system can be waiting for a single node. It is important to eliminate any "hot spots" in the system, such as hot spots caused by data access (accessing data from a single node) or network IO (transferring data into or out of a single node).

Fault tolerance comes with a price. Both the original MapReduce paper [5] and the Hadoop communities have emphasized the importance of fault tolerance since the systems are designed to run over commodity hardware which fails frequently. However, in certain cases, such as Reduce, fault tolerance introduces extra overhead in order to replicate the intermediate results. In some cases, Hadoop applications are actually run on small to medium sized clusters, and hardware failure during MapReduce processing is rare. It is reasonable in these case to favor performance and re-run any failed Reduces when necessary.

Balanced systems. Although it is obvious to anyone who has set up a Hadoop or Sector cluster, it does not hurt to emphasize the importance of using a design in which the CPU, disk, and networked are well balanced. Many systems we have seen have too many cores and not enough spindles for data intensive computing.

Streams are important. We were a bit surprised by the usefulness of the streaming interface provided by Hadoop, and more recently Sector. With this interface, it is quite easy to support many legacy applications, and, for some of these, the performance is quite respectable. Since Sector does not split files, working with streams is quite efficient.

8. SUMMARY

We have compared and contrasted two systems for data intensive computing – Sector/Sphere and Hadoop. Hadoop was designed originally for processing web data, but has proved useful for a number of other applications. Sector supports a more general parallel programming framework (the ability to apply Sphere UDFs over the data managed by Sector), but is still very easy for most programmers to use. We have discussed some of the design differences between the two Sector Distributed File System and the Hadoop Distributed File Systems. In our experimental studies, Sector/Sphere is about 2 – 4 times faster than Hadoop. In this paper, we have explored some of the possible reasons for Sector's superior performance.

9. Acknowledgements

The Sector/Sphere software system is funded in part by the National Science Foundation through Grants OCI-0430781, CNS-0420847, ITR-0325013 and ACI-0325013.

10. REFERENCES

[1] Collin Bennett, Robert Grossman, and Jonathan Seidman, Open Cloud Consortium Technical Report TR-09-01, MalStone: A Benchmark for Data Intensive Computing, Apr. 2009.

[2] Beynon, Michael D. and Kurc, Tahsin and Catalyurek, Umit and Chang, Chialin and Sussman, Alan and Saltz, Joel, Distributed processing of very large datasets with DataCutter, Journal of Parallel Computing, Vol. 27, 2001. Pages 1457 - 1478.

[3] J. Bent, D. Thain, A. Arpaci-Dusseau, and R. Arpaci-Dusseau, "Explicit control in a batch-aware distributed file system," in Proceedings of the First USENIX/ACM Conference on Networked Systems Design and Implementation, March 2004.

[4] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber, Bigtable: A Distributed Storage System for Structured Data, OSDI'06: Seventh Symposium on Operating System Design and Implementation, Seattle, WA, November, 2006.

[5] Jeffrey Dean and Sanjay Ghemawat, MapReduce: Simplified Data Processing on Large Clusters, OSDI'04: Sixth Symposium on Operating System Design and Implementation, San Francisco, CA, December, 2004.

[6] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung, The Google File System, pub. 19th ACM Symposium on Operating Systems Principles, Lake George, NY, October, 2003.

[7] Albert Greenberg, James R. Hamilton, Navendu Jain, Srikanth Kandula, Changhoon Kim, Parantap Lahiri, David A. Maltz, Parveen Patel, and Sudipta Sengupta, VL2: A Scalable and Flexible Data Center Network, SIGCOMM 2009.

[8] Yunhong Gu and Robert Grossman, Exploring Data Parallelism and Locality in Wide Area Networks, Workshop on Many-task Computing on Grids and Supercomputers (MTAGS), co-located with SC08, Austin, TX. Nov. 2008

[9] Yunhong Gu, Robert Grossman, UDT: UDP-based data transfer for high-speed networks, Computer Networks (Elsevier), Volume 51, Issue 7. May 2007.

[10] Yunhong Gu, Robert L. Grossman, Alex Szalay and Ani Thakar, Distributing the Sloan Digital Sky Survey Using UDT and Sector, Proceedings of e-Science 2006.

[11] Tevfik Kosar and Miron Livny, Stork: Making Data Placement a First Class Citizen in the Grid, in Proceedings of 24th IEEE International Conference on Distributed Computing Systems (ICDCS 2004), Tokyo, Japan, March 2004.

[12] T. Kurc, Umit Catalyurek, C. Chang, A. Sussman, and J. Salz. Exploration and visualization of very large datasets with the Active Data Repository. Technical Report CS-TR4208, University of Maryland, 2001.

[13] I. Raicu, Z. Zhang, M. Wilde, I. Foster, P. Beckman, K. Iskra, and B. Clifford, Toward Loosely Coupled Programming on Petascale Systems, Proceedings of the 20th ACM/IEEE Conference on Supercomputing.

[14] Douglas Thain, Todd Tannenbaum, and Miron Livny, "Distributed Computing in Practice: The Condor Experience" *Concurrency and Computation: Practice and Experience*, Vol. 17, No. 2-4, pages 323-356, February-April, 2005.

[15] Hadoop, hadoop.apache.org/core, Retrieved in Oct. 2009.

[16] The Open Cloud Testbed, http://www.opencloudconsortium.org.

[17] Ioan Raicu, Ian Foster, Yong Zhao, Many-Task Computing for Grids and Supercomputers, Workshop on Many-task Computing on Grids and Supercomputers (MTAGS), co-located with SC08, Austin, TX. Nov. 2008