Exploring Data Parallelism and Locality in Wide Area Networks

Yunhong Gu and Robert Grossman[1]
University of Illinois at Chicago
{ygu3, grossman}@uic.edu

Abstract

Cloud computing has demonstrated that processing very large datasets over commodity clusters can be done simply given the right programming structure. Work to date, for example MapReduce and Hadoop, has focused on systems within a data center. In this paper, we present Sphere, a cloud computing system that targets distributed data-intensive applications over wide area networks. Sphere uses a data-parallel computing model that views the processing of distributed datasets as applying a group of operators to each element in the datasets. As a cloud computing system, application developers can use the Sphere API to write very simple code to process distributed datasets in parallel, while the details, including but not limited to, data locations, server heterogeneity, load balancing, and fault tolerance, are transparent to developers. Unlike MapReduce or Hadoop, Sphere supports distributed data processing on a global scale by exploiting data parallelism and locality in systems over wide area networks.

1. Introduction

Today, clusters of commodity workstations and high performance networks are ubiquitous. Scientific instruments routinely produce terabytes or even petabytes of data every year. As a result, large distributed datasets are quite common. Recently, a new computing paradigm called cloud computing has been used to provide a simple programming interface for large scale data processing using clusters of commodity computers. The most well known cloud computing system is Google's GFS/MapReduce/BigTable stack [5, 4, 2] and its open source implementation Hadoop [15]. The approach taken by cloud computing is to provide a very simple distributed programming interface by limiting the type of operations supported. The MapReduce paradigm used in these systems is fundamentally a simplified data parallel and master/worker pattern.

In order to utilize this simplified computing paradigm, the computation tasks applied on each element of the dataset are usually relatively independent. Such paradigm is also called Multi-task computing (MTC) [11].

To date, Hadoop is one of the most successful systems in this category. However, as far as we know, all of these systems are set up on racks of clusters within a single data center. There are many applications, especially scientific data processing, that naturally generate, store, and process data in geographically distributed locations. In this paper, we present a new compute cloud called Sphere that supports data parallelism, exploits data locality when possible, and is effective in loosely coupled systems connected by high speed wide area networks.

Sphere is built on top of the Sector storage cloud that has been used to store and distribute terabyte-sized datasets (e.g., astronomy data) using clusters connected by 10Gb/s wide area networks [8]. Sector provides the storage services, while Sphere provides the computing services for high performance distributed applications.

Applications developed using the Sphere client API do not need to locate and move data explicitly, nor do they need to locate computing resources, explicitly provide load balancing, or provide explicit support for fault tolerance. The distributed parallelization is done implicitly by Sphere: Sphere automatically locates data and computing resources to run the required processing function in parallel, while Sector provides a uniform data access interface.

Sphere uses a stream processing paradigm [10] to process large datasets. In the stream processing paradigm, each element (e.g., a record) in the input dataset is processed independently by a processing function (i.e., the same processing function is used for

[1] Robert Grossman is also a Partner at the Open Data Group.

978-1-4244-2872-4/08/$25.00 ©2008 IEEE

each element in the input dataset). Fundamentally, Sphere uses distributed computers to process a large dataset in parallel, achieving the same results as a loop in a serial program. For example, consider the following loop in a serial program. Note that the term "stream" used in "stream processing paradigm" means a large static dataset. It does NOT refer to the infinite or very large streaming data usually from a live source such as sensors.

```
for (int i = 0; i < 100000000; ++ i)
  process(data[i]);
```

In the stream process paradigm used by Sphere, this loop will be replaced by:

```
sphere.run(data, process);
```

The majority of the processing time for many data intensive applications is spent in loops like these; developers typically spend a lot of their time parallelizing these types of loops (e.g., with PVM or MPI). Parallelizing these loops in distributed environments presents additional challenges. Sphere provides a simple way for application developers to express these loops, and then Sphere parallelizes and distributes the required computations automatically.

We believe that Sphere can help to simplify the data processing in high speed wide area networks, especially in e-science collaborations such as astronomy and bioinformatics, where the data is generated, stored, and processed in multiple, geographically distributed locations. To the best of our knowledge, Sphere is the first system that provides this simplified data processing support over loosely coupled high speed wide area networks. This is the main contribution of our paper.

In the rest of this paper, we will describe the design, implementation, and evaluation of the Sphere system. We briefly explain the Sector file system in Section 2 and then describe how Sphere is designed in Section 3. In Section 4 we use experimental studies to evaluate the performance of Sphere. The paper is concluded in Section 5.

2. Sector File System

Sphere runs on top of a distributed file system called Sector, so that it can locate files and move the files when necessary.

Figure 1 shows the system architecture of Sector. The Security server maintains user accounts, passwords, and privileges on each of the files or directories. It also maintains a list of IP addresses of the allowed slaves, so that illicit computers cannot join the system or send messages to interrupt the system.

The master server maintains the metadata of the files stored in the system, controls the running of all slaves, and responds to users' requests. The master communicates with the security server to verify the slaves and the clients/users.

The slaves are the nodes that actually store files and process the data upon request. The slaves are usually racks of computers that are located in one or more data centers. The topology of the slave system (e.g., the 2-level data center/rack hierarchy) can be manually specified by a configuration file to the master server. High speed networks are expected to connect all nodes and sites.

A client logs in to the master server and requests access to data files or runs an application. The master checks the credibility of the client and its privileges on the requested files (by asking the security server). If granted, the master will choose one or more slaves to process the client's request. The slaves and the client will set up data connections on rendezvous.

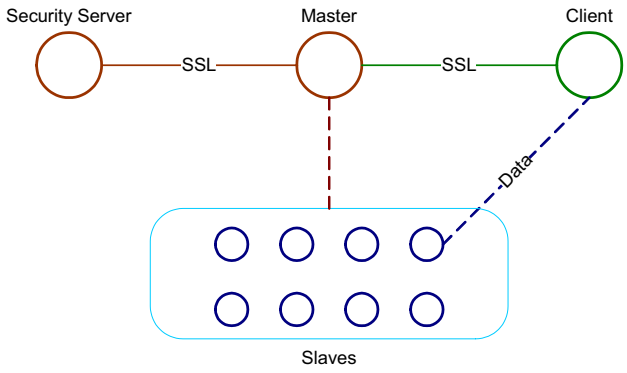

Figure 1. Sector/Sphere system diagram.

Sector provides functionality similar to that of a distributed file system. In addition, the master checks the number of copies of each file periodically. If the number is below a threshold (e.g., 3), then the master will choose a slave to make a new copy of the file. The new location of the file copy is based on the topology of the slave network. When a client requests a file, the master can choose a slave (that contains a copy of the file) that is close to the client and is not busy with other services. When the slaves are across wide area networks, Sector can act like a content distribution network or a file sharing system.

3. Sphere

3.1 Overview

To introduce Sphere, consider the following example application. Assume we have 1 billion astronomical images of the universe from the Sloan Digital Sky Survey and the goal is to find brown dwarfs (a stellar object) in these images. Suppose the average size of an image is 1MB so that the total data size is 1TB. The SDSS dataset is stored in N files, named SDSS1.dat, …, SDSSn.dat, each consisting one or more images.

In order to access an image randomly in the dataset, we built a record index for each file. The record index indicates the start and end positions (i.e., offset and size) of each record (in this case, an image) in the data file. The record indexes are named by adding a ".idx" postfix to the data file name: SDSS1.dat.idx, …, SDSSn.dat.idx.

To use Sphere, the user writes a function "findBrownDwarf" to find brown dwarfs from each image. In this function, the input is an image and its size, while the output indicates the brown dwarfs in "result" of size "rsize". This function is put into a dynamic library file that is stored on Sector servers.

findBrownDwarf(char* image, int isize, char* result, int rsize);

A standard serial program might look like this:

```
for each file F in (SDSS datasets)
   for each image I in F
      findBrownDwarf(I, …);
```

Using the Sphere client API, the corresponding pseudo code looks like this:

```
SphereStream sdss;
sdss.init("sdss files");
SphereProcess myproc;
myproc->run(sdss,"findBrownDwarf");
myproc->read(result);
```

In the code fragment above, "sdss" is a Sector stream data structure that stores the metadata of the SDSS files. The application can initialize the stream by giving it a list of file names. Sphere automatically retrieves the metadata from the Sector network. The last three lines will simply start the job and wait for the result using a small number of Sphere APIs. There is no need for users to explicitly move data, nor do they need to know any information about the Sector servers except for the location of one server to which the client can connect.

3.2 The Computing Paradigm

We begin by explaining the key abstractions used in Sphere. A dataset consists of one or more physical files. A stream is an abstraction in Sphere and it represents part or the entire dataset as the input/output of a Sphere computation. Note that "stream" in this context does not refer to streaming data from a live source. A Sphere stream is split into multiple data segments and is processed by Sphere Processing Engines (SPEs) on Sphere servers. An SPE processes each data segment by elements, which can be a data record, a group of records, or a complete physical file.

Figure 2 shows the computing paradigm of Sphere. An application communicates with a Sphere client by exchanging parameters and results. The Sphere client first collects information about an input stream, including the total size and the total number of records. Second, the Sphere client locates the required service providers, or SPEs, by looking for the dynamic library files with the same name as the processing functions in the code. The dynamic library files are also stored in Sector. Based on this information, the client splits the input stream into data segments. Usually the number of data segments is much larger than the number of SPEs.

At the beginning of the job, each SPE is assigned one data segment and starts to process it. Once the processing is finished and the result is collected, the Sphere client assigns the SPE a new data segment to process.

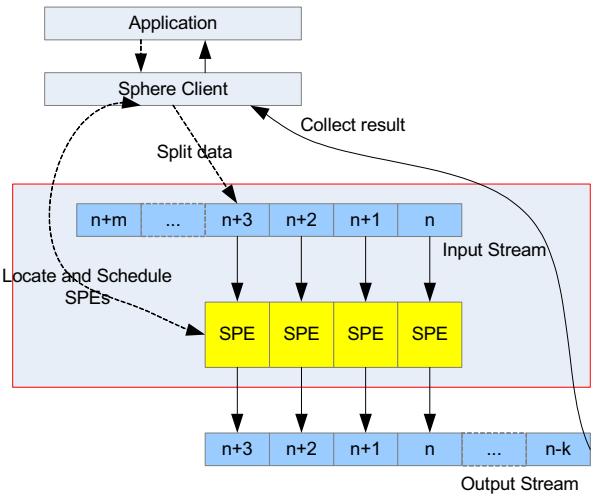

Figure 2. The computing paradigm of Sphere.

The result stream (output data) is either sent back to the client or written into Sector files. In the latter case, Sphere can dump the results of each SPE to a local (Sector server) node, or transfer them to a list of specified destinations ("shuffling"). In the "shuffling" process, each destination will combine the results from

multiple SPEs. Whether the data is sent back, written to local disk, or shuffled to a specific destination depends on how the output stream is defined.

Both the input and the output of the Sphere process are Sector Streams. Therefore, the output stream can be further processed by a new Sphere process. That is, the process described in Figure 2 can be repeated multiple times to conduct more complicated computations. Each processing is one "stage" in Sphere. Figure 3 shows a two-stage computation that sorts a large dataset in a distributed manner.

In Figure 3, the first stage shuffles the input data into multiple buckets. The shuffling function scans the complete stream and places each element in a proper bucket. For example, if the data to be sorted is integers, the shuffling function can place all data less than T0 in bucket B0, data between T0 and T1 in bucket B1, and so on.

In stage 2, each bucket (data segment) is sorted by an SPE. After stage 2, the whole dataset (stream) is sorted, because all elements in a certain bucket are smaller than those in any buckets further in the stream.

Note that in this stage, the SPE processes the whole data segment together (sorting), rather than processing each record individually. In Sphere, an SPE can process a single record, multiple records, the whole segment, or a complete file at one time. This choice is specified in the configuration of the input stream.

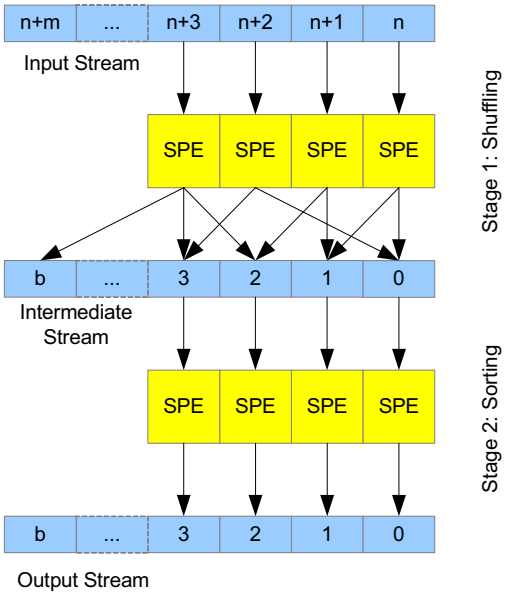

Figure 3: Sorting large distributed datasets with Sphere

To summarize, Sphere supports applications in which the data segments (records, group of records, or files) can be processed independently and the results can be either stored independently or merged. The process can be repeated as the output of a previous stage can be the input of the next stage.

3.3 Sphere Server

An SPE is the major Sphere service and is started by a Sphere server in response to a request from a Sphere client. Each SPE is based on a streaming processing function (operator) defined by the application developer. The stream processing function is implemented as a dynamic library and is stored on the server local disk and is managed by the Sector server. The library file can be either distributed by system administrators or uploaded by users who have write access to the servers that are allowed to run the particular job. In contrast to data files, Sector does not create duplicates of library files.

A shortcoming of the current Sphere implementation is that library files are not distributed automatically. We are developing a tool that will automatically distribute the library file according to the user and security profile, as well as system compatibility.

Once the server accepts the client's request, it starts an SPE and binds it to the local stream processing function. The SPE runs in a loop that consists of the following four steps:

First, the SPE accepts a new data segment from the client containing the file name, offset, number of rows to be processed, and additional parameters, etc.

Next, the SPE reads the data segment and its record index from the local disk or from a remote Sector server node.

For each element (single data record, group of data records, or entire data file), the stream processing function processes the element and writes the result to a temporary buffer. In addition, the SPE periodically sends acknowledgments to the client about the progress of the processing.

When the data segment is completely processed, the SPE sends an acknowledgment to the client and writes the results to the appropriate destinations.

If there are no more data segments to be processed, the client closes the connection to the SPE, and the SPE is released. The SPE may also timeout if the client is interrupted.

3.4 Sphere Client

The Sphere client provides a set of APIs that developers can use to write distributed applications. The client can initialize itself by connecting to any running Sector server and during the lifetime of the client will cache the address for more server nodes so

that if the connection to the primary server is lost, the client will automatically connect to another available server. Once initialized, the client is responsible for initializing Sector streams, locating and scheduling SPEs, and collecting results.

The client splits the input stream into multiple data segments, so that each can be processed independently by an SPE. The processing status and the result of the computation, when the result is returned instead of being persisted to the local disk, are returned by the SPE. A data structure called Data Segment Index tracks the status of these segments (e.g., whether the segment has been processed) and holds the results.

The client is responsible for orchestrating the complete running of each Sphere process. One of the design principles of the Sector/Sphere system is to leave most decision making to the client and keep the simplest server core. In Sphere, the client is responsible for the control and scheduling of the program execution. The assumption is that the client knows more about the particular application than the server.

Note that we developed the Sphere client as the runtime library to support Sphere computing. The related functionalities described in this paper are handled by the Sector/Sphere client and are transparent to applications. Developers only use the client API in their applications.

3.5 Scheduling

The client first locates the data files in the input stream from Sector. If the input stream is the output stream of a previous stage, then this information is already within the Sector stream structure and no further segmentation is needed.

Both the total data size and the total number of records are calculated in order to split the data into segments. This is based on the metadata of the data files retrieved from Sector.

Next, the client locates any Sector servers that contain the required stream processing functions. These servers are candidates that can start SPEs. Each server may start multiple instances of different services (e.g., file access and SPE are both Sector services). The maximum number of services is specified in the server configuration file. If the quota of a Sector server has not been reached, then the Sphere selects the Sector server and start one or more SPEs.

The client tries to uniformly distribute the input stream to the available SPEs by calculating the average data size per SPE. However, in consideration of the physical memory available per SPE and the data communication overhead per transaction, Sphere limits the data segment size between size boundaries Smin and Smax (the default values are 8MB and 128MB, respectively, but user defined values are supported). In addition, the scheduler rounds the segment size to a whole number of records since a record cannot be split. The scheduler also requires that a data segment only contains records from a single data file.

As a special case, the application may request that each data file be processed as a single segment. This would be the case, for example, if an existing application were designed to only process files. This is also the way the scheduler works when there is no record index associated with the data files.

Once the input stream is segmented, the client assigns each segment to an SPE. The following rules are applied:

Each data segment is assigned to an SPE on the same node if there is one available.

Segments from the same file are not processed at the same time unless, otherwise, an SPE will be idle.

If there are still idle SPEs available after rule 1 and rule 2 are applied, assign them incomplete data segments in the same order as they are in the input stream.

The first rule tries to run a SPE on the same node as where the data resides (in other words, to exploit locality). This reduces the network traffic and yields better throughput. The second rule allows better data access concurrency because SPEs can read data from multiple files independently at the same time.

As mentioned in Section 3.3, SPEs periodically provide feedback about the progress of the processing. If an SPE does not provide any feedback about the progress of the processing before a timeout occurs, then the client discards the SPE. The segment being processed by the discarded SPE is assigned to another SPE, if one is available, or placed back into the pool of unassigned segments. This is the mechanism that Sphere uses to provide fault tolerance. Currently, Sphere does not use any check pointing in an SPE; when the processing of a data segment fails, it is completely re-processed by another SPE.

In the current implementation, the process feedback period from an SPE is 0.1 second and the timeout value is the maximum between 1 second and 4 times the round trip time (RTT) between the client and the server.

Different SPEs can require different times to process data segments. There are several reasons for this, including: Sector/Sphere servers may not be dedicated; the servers may have different hardware configurations (Sector systems are typically heterogeneous); and different data segments may require different processing times. Near the end of the

computation, when there are idle SPEs but incomplete data segments, each idle SPE is assigned one of the incomplete segments. That is, the remaining segments are run on more than one SPE and the client collects results from whichever SPE finishes first. In this way, Sphere avoids waiting for the slow SPEs while the faster ones are idle.

If errors occur during the processing of a data segment due to problems with the input data or bugs in user defined functions, the data segment will not be processed by any other SPE. Instead, an error report is sent back to the client, so that the application can take the appropriate action.

In most cases, the number of data segments is significantly greater than the number of SPEs. For example, hundreds of machines might be used to process terabytes of data. As a consequence, the system is naturally load balanced because all SPEs are kept busy during the majority of the runtime. Imbalances only occur towards the end of the computation when there are fewer and fewer data segments to process, causing some SPEs to be idle.

3.6 Outputs

When SPE processing produces small amounts of data, the application can directly get back the results. An application can choose either to read the results of the segments as the SPE completes the processing or in the order of the original file segments being processed. Obviously the former case is more efficient. Results can be stored temporarily in the Data Segment Index and reordered there when necessary.

However, when the data produced by the SPEs is large, a bottleneck may occur when these large result buffers are all written back to a single client. In this case, applications can gain considerable speed up by using Sector to store the results in Sector files for future use. Recall that Sector provides high performance peer-to-peer storage specifically designed to store large distributed files efficiently.

Sphere has two modes for file output: regular mode and shuffle mode. With the regular mode, the output files are saved to the local disk. In the shuffle mode, in which the output from all SPEs can be mixed together into multiple destination files, the Sphere client specifies the appropriate Sector nodes to write the output files.

More flexible output means that more applications can be supported by Sphere, but this also requires more complicated data movement between Sphere servers. For example, in the shuffling stage of the distributed sort (See Section 4.4 for more details), each server needs to write output to all the other servers.

One important technique we use to optimize output processing in Sphere is to cache data connections so that the connection setup and the consequent slow start phase in the congestion control algorithm is significantly reduced. For example, if a data connection is set up to transfer data between two nodes, the connection is not closed after the data transfer is completed (unless a limit on the number of open connections is exceeded). If a new data transfer between the same two nodes is required, the cached connection is reused rather than a new connection being setting up. Connection setup causes significant delay and network congestion (and hence low throughput), because transport protocols have a slow start phase that aggressively probes the available bandwidth. This problem is especially acute in high speed wide area networks.

4. Experimental Studies

We have developed an open source version of Sector/Sphere that is available on SourceForge [16]. We conducted experimental studies using this implementation on the TeraFlow network testbed [17] and the Open Cloud Testbed [18]. In Section 4.1 to 4.3, we examine the performance related to SPE function processing time, data size, and fault tolerance, respectively. In Section 4.4, we compare Sphere and Hadoop using the TeraSort benchmark.

4.1 Impact of SPE Processing Time

In the first three experiments, we choose 10 servers on the TeraFlow testbed. These servers have dual AMD Opteron 2.4GHz or 3.0GHz, 2-4GB RAM, 1.5-5.5TB disk and are connected by 10Gb/s wide area networks. Of the 10 machines, 2 are in Chicago, IL, 4 are in Greenbelt, MD, 2 are in Pasadena, CA, and 2 are in Tokyo, Japan.

The processing function we use to test Sphere is a Null function that actually does no real computations, but instead uses a loop to consume a specified amount of CPU time. The Null function can be applied to any data, since it simply ignores the input. A parameter can be used to specify the length of the busy waiting. The actual wait is determined by sampling from a uniform distribution that is centered on a specified value, and varies between -10% and +10% of the specified value as a simple way to simulate processing time variations in real applications.

In this experiment, we increased the duration of the busy loop in the Null function and used a very small dataset so that the application is CPU intensive rather than data intensive. Specifically, we used a data file of

8MB and provided each Sector server with a local copy. The file contains 1MB records and each record is 8 bytes long.

We used four different durations for the per-record processing times (0, 10µs, 100µs, and 1ms, respectively) and ran experiments using 1 to 10 SPEs. The performance increase factor (shown on the y-axis) is defined as the throughput of multiple SPEs over that of a single SPE.

Figure 4 illustrates the impact of the per-record processing time on the performance of Sphere. When the per-record processing time is 1ms, the performance stably increases to almost 9 for 10 SPEs (16m47.686s for 1 SPE vs. 1m59.918s for 10 SPEs). This is the performance we expected because the data IO overhead is minimal and the computation is distributed to multiple SPEs. The overhead caused by SPEs is also negligible.

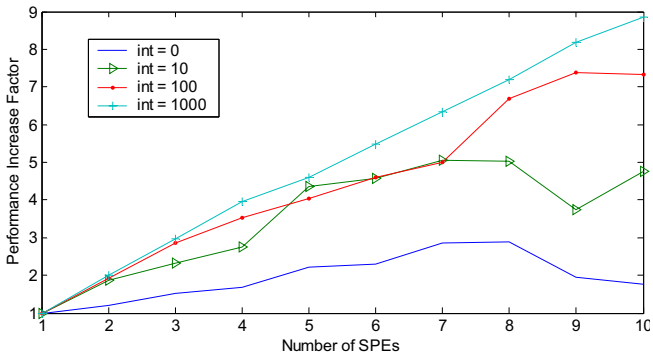

Figure 4: Performance speedups vs. element processing time.

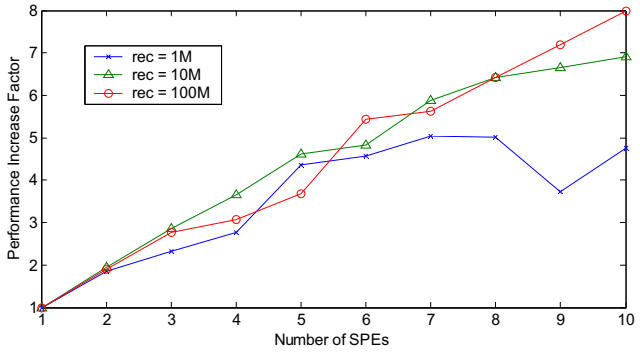

Figure 5: Performance speedups vs. number of elements

However, the Sphere's overhead becomes more significant as the per-record processing time decreases and the number of SPE increases. In Figure 4, the performance stops improving as the number of SPE passes 7 for per-record processing times of 0 and 10 microseconds. When the overhead per SPE is greater than the computation task assigned to the SPE, the overall performance will decrease. This is why Sphere imposes a lower bound for the size of a data segment.

4.2 Impact of Data Size

The size of a data segment has a similar impact on the performance as that of the per-record processing time, because as long as the total processing time (equal to the product of the number of records and the per-record processing time) is significantly greater than the per SPE overhead, Sphere will benefit from the higher parallelism by using more SPEs.

We used the same experiment settings of Section 4.1 but increased the total number of data records to 10M and 100M. We set the per-record processing time to 10µs. Figure 5 shows the results of the experiments. With 1M data records, the performance stops increasing after 7 SPEs and decreases at 9 SPEs. On the other hand, with 10M and 100M data records, the performance continues to increase with the addition of each new SPE.

The small number of data segments in the experiments causes the oscillations in the performance curve. Because the data size is small and only a small number of data segments are needed (there is only one data segment for each SPE), once an SPE finishes processing its first segment, there are no more data segments for it to process. For this reason, the system load becomes less balanced towards the end of the computation. In real applications, we expect that the number of data segments is significantly greater than the number of SPEs.

4.3 Fault Tolerance

Finally, we tested the fault tolerance characteristics of the system when one of the SPEs fails. This experiment has the same set-up as the experiment described in Section 4.2. Specifically, the per-record processing time is 10µs and the data size is 800MB (100M records).

In this experiment, we stopped one of the SPEs before it completed processing its segment, which forced Sphere to find a new SPE to continue the interrupted task.

Figure 6 shows the result. We see that Sphere handles the SPE failure well and that the performance is similar to the performance of a normal execution with one SPE less. A larger dataset would have better load balancing and the performance curve would be smoother.

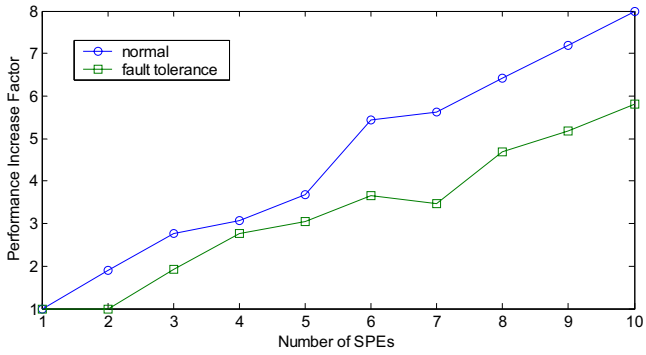

Figure 6: Execution time vs. number of lost SPEs.

4.4 The TeraSort Benchmark

We implemented the TeraSort benchmark [6, 13] in Sphere and compared its performance against Hadoop in various settings. The TeraSort application sorts a large dataset of 100-byte records with 10-byte key. In the benchmark configuration, the data is distributed on all participating nodes and each node stores 10GB data. Most of the distributed sorting algorithm uses the 2-stage strategy described in Section 3.2: the data is hashed into multiple buckets and each bucket is then sorted locally.

In the Sphere implementation, the data is hashed into 2k buckets by using the first k-bit of the 10-byte key as the hash key. The local sorting method is the Introsort algorithm used by C++ STL [9]. The hashing method can result in buckets with different sizes if the data is not uniformly distributed. To compensate, one can use a "sampling" stage that samples the input data and decides how to hash the data into similar size buckets.

TeraSort is a good benchmark to examine the performance of Sphere because it is both data intensive (10GB data each node) and compute intensive (nlogn complexity for sorting). In addition, it needs to move almost all the data between participating nodes. With S nodes, approximately (S-1)/S * 10GB data is moved during the sorting. As described in Section 3.6, Sphere pushes the data to different buckets in the hash stage, whereas Hadoop pulls the data from each Map result file into the Reducer at the beginning of the Reduce phase. In Sphere, the intermediate data is written into K buckets directly, whereas in Hadoop, there will be S * K temporary files.

The appendix contains the 52 lines of Sphere code required to implement TeraSort (due to page limitations, we removed the comment lines and error messages). The "hash" and the local "sort" SPE functions require an additional 36 and 100 lines, respectively. In contrast, because sorting is a built-in functionality in Hadoop, the Hadoop TeraSort program, which totals 161 lines of Java code, is simply a framework to organize the input, output, and execution parameters.

The performance value listed in Table 1 was achieved using the Open Cloud Testbed [18]. Currently the testbed consists of 4 racks. Each rack has 32 nodes, including 1 NFS server, 1 head node, and 30 compute/slave nodes. The head node is a Dell 1950, dual dual-core Xeon 3.0GHz, 16GB RAM. The compute nodes are Dell 1435s, single dual core AMD Opteron 2.0GHz, 4GB RAM, and 1TB single disk. The 4 racks are located in JHU (Baltimore), StarLight (Chicago), UIC (Chicago), and Calit2 (San Diego).

The nodes on each rack are connected by two Cisco 3750E switches, but only 1GE is enabled at this time (maximum 2GE can be enabled in/out each node). The inter-rack bandwidth is 10GE, supported by CiscoWave deployed over National Lambda Rail.

In summary, both Sector and Hadoop are deployed over the 120-node (240-core) wide area system. The master of Sector, or the name node/job tracker of Hadoop, is installed on one or more of the 4 head nodes.

Table 1 lists the performance (time in seconds; data generation time is excluded.) of the Terasort benchmark of both Sphere and Hadoop. Note that it is normal for the longer processing time for more nodes because the total amount of data also increases proportionally.

In this experiment, we sort 300GB, 600GB, 900GB, and 1.2TB data over 30, 60, 90, and 120 nodes, respectively. In the last case, the 1.2TB data is distributed on four racks located in four data centers across the U.S. All 120 nodes participated in the sorting process and almost 1.2TB data will be moved within the testbed.

Table 1. Terasort Benchmark for Sphere and Hadoop

	Sphere	**Hadoop 3 replica**	**Hadoop 1 replica**
UIC	1265	2889	2252
UIC + StarLight	1361	2896	2617
UIC+StarLight + CalIT2	1430	4341	3069
UIC+StarLight + CalIT2+JHU	1526	6675	3702

Because Sector duplicates data files periodically, while Hadoop writes multiple copies at real time, it is a fair comparison when we configure Hadoop to generate 1 copy only (same number of data copy as

Sector during data processing). In the second column, Hadoop is configured to generate 3 replicas for each file; in the third column, Hadoop is configured to generate 1 replica only.

The results show that Sphere is about twice as fast as Hadoop. Moreover, Sphere scales better as the number or racks increases (1526/1265 vs. 3702/2252).

Initial analysis shows that the superior performance demonstrated by Sphere benefited from multiple factors, including optimized data flow management, data transfer protocol (Sphere uses UDT [7]), and C++ implementation (vs. Java for Hadoop). We will investigate more details in future work.

5. Conclusions

We have presented Sphere, a distributed programming framework that supports simplified data parallel applications. Sphere implements stream processing paradigm over clusters of computers, either within single data center or across a wide area.

Like MapReduce, Sphere tries to process the data on the node where it is located, while it hides data locations, data movement (when necessary), and fault tolerance. Unlike MapReduce, Sphere's stream processing paradigm is simpler, more straightforward, and reaches better performance in certain cases.

Compared to traditional computing, Sphere is more data oriented, while Grid scheduling systems such as Swift [12] and DAGMan [14] are more task oriented. Finally, Sphere also differs from systems that process infinite streaming data from live sources, such as GATES [3] and DataCutter [1].

6. Acknowledgment

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

7. References

[1] M. D. Beynon, R. Ferreira, T. Kurc, A. Sussman, and J. Saltz. DataCutter: Middleware for filtering very large scientific datasets on archival storage systems. Mass Storage Systems Conference, College Park, MD, March 2000.

[2] 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, Seattle, WA, November, 2006.

[3] Liang Chen, K. Reddy, and Gagan Agrawal. GATES: A Grid-Based Middleware for Processing Distributed Data Streams. 13th IEEE International Symposium on High Performance Distributed Computing (HPDC) 2004, Honolulu, Hawaii.

[4] 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.

[5] 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.

[6] Jim Gray. Sort benchmark home page. http://research.microsoft.com/barc/SortBenchmark/.

[7] Yunhong Gu, Robert L. Grossman, *UDT: UDP-based Data Transfer for High-Speed Wide Area Networks*. Computer Networks (Elsevier). Volume 51, Issue 7. May 2007.

[8] 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.

[9] D. R. Musser, "Introspective Sorting and Selection Algorithms", Software Practice and Experience 27(8):983, 1997.

[10] John D. Owens, David Luebke, Naga Govindara Ju, Mark Harris, Jens Kruger, Aaron E. Lefohn, and Timothy J.Purcell, A Survey of General-Purpose Computation on Graphics Hardware, Eurographics 2005, pages 21-51, 2005.

[11] Ioan Raicu, Zhao Zhang, Mike Wilde, Ian Foster, Pete Beckman, Kamil Iskra, Ben Clifford. "Toward Loosely Coupled Programming on Petascale Systems", to appear at IEEE/ACM Supercomputing 2008.

[12] Y. Zhao, M. Hategan, B. Clifford, I. Foster, G. von Laszewski, I. Raicu, T. Stef-Praun, and M. Wilde, Swift: Fast, Reliable, Loosely Coupled Parallel Computation IEEE International Workshop on Scientific Workflows 2007.

[13] Jim Wyllie. Spsort: How to sort a terabyte quickly. http://alme1.almaden.ibm.com/cs/spsort.pdf.

[14] The Condor DAGMan (Directed Acyclic Graph Manager), http://www.cs.wisc.edu/condor/dagman, 2007.

[15] Hadoop, http://hadoop.apache.org/core.

[16] Sector, http://sector.sf.net.

[17] The Teraflow Testbed, http://www.teraflowtestbed.net

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

8. Appendix: C++ Code for TeraSort with Sphere

```
#include "dcclient.h"
using namespace cb;   // cb is Sector/Sphere name space

int main(int argc, char** argv) {
   Sector::init(argv[1], atoi(argv[2]));   // argv[1,2]: server IP, port

   vector<string> files;
   files.insert(files.end(), "rand1.dat");
   files.insert(files.end(), "rand2.dat");
   Stream s;
   if (s.init(files) < 0)
      return -1;

   Stream temp;       // intermediate results, buckets for hashed data
   temp.m_strName = "stream_sort_bucket";
   temp.init(64);     // = 2^6 buckets

   Process* myproc = Sector::createJob();
   int n = 6;         // hash parameter, using the first 6-bits
   if (myproc->run(s, temp, "sorthash", 1, (char*)&n, sizeof(int)) < 0)
      return -1;

   Result* res;
   while (true)
      if (-1 == myproc->read(res)) {
         if (myproc->checkProgress() == -1)      // failed, quit
            return -1;
         if (myproc->checkProgress() == 100)     // progress 100%, stop
            break;
         continue;
      }

   Stream output;
   output.init(0);

   if (myproc->run(temp, output, "sort", 0) < 0)
      return -1;

   while (true)
      if (-1 == myproc->read(res)) {
         if (myproc->checkProgress() == -1)
            return -1;
         if (myproc->checkProgress() == 100)
            break;
         continue;
      }

   myproc->close();
   Sector::releaseJob(myproc);
   Sector::close();
   return 0;
}
```