Processing Massive Sized Graphs Using Sector/Sphere

Yunhong Gu, Li Lu
University of Illinois at Chicago
yunhong@lac.uic.edu
lilu@lac.uic.edu

Robert Grossman
University of Illinois at Chicago and
Open Data Group
grossman@labcomputing.org

Andy Yoo
Lawrence Livermore National
Laboratory
ayoo@llnl.gov

***Abstract*—Data intensive computing is having an increasing awareness among computer science researchers. As the data size increases even faster than Moore's Law, many traditional systems are failing to cope with the extreme large volumetric datasets. In this paper we use a real world graph processing application to demonstrate the challenges from the emerging data intensive computing and present a solution with a system called Sector/Sphere that we developed in the last several years. Sector provides scalable, fault-tolerant storage using commodity computers, while Sphere supports in-storage parallel data processing with a simplified programming interface. This paper describes the rationale behind Sector/Sphere and how to use it to effectively process massive sized graphs**

Keywords-Graph, Breadth First Search, Data Intensive Computing.

I. INTRODUCTION

During the past decade we have witnessed two trends in high performance computing (HPC). On the one hand, as the datasets increase exponentially, there are more and more data intensive applications (in contrast to compute intensive applications). On the other hand, due to advances in computing technologies and the significant drop in hardware prices, many HPC applications that used to rely on super computers can now run on inexpensive commodity clusters.

Although commodity computer clusters may have limitations on their performance and programmability due to distributed memory, low bandwidth (compared to system bus), and high communication latency between processors, they can achieve very good performance for data intensive applications when a certain level of data parallelism is present, which is common for very large datasets. Data parallelism allows data to be processed on the same node where it is stored, therefore it avoids the communication bottleneck/penalties in the cluster system. In fact, data locality is the fundamental principle that supports distributed data intensive applications over these commodity clusters.

Data locality requires an integrated compute and storage system. That is, in the system each storage node is also a compute node. On a commodity cluster, usually each node (i.e., one computer) uses its internal hard drives as storage. Today, a typical rack may contain 40 computers, 160 - 320 processor cores, 160TB storage, 640GB memory, and 1 - 10Gb/s node-node Ethernet connection. The cost for such a rack is approximately between $100,000 - $200,000 USD.

A software system that allows easy development of data intensive applications on the above hardware can pave the way to the popularization of HPC.

Google's MapReduce [5] is one of the most notable examples of using commodity clusters to process very large datasets. A typical application at Google is to use MapReduce to generate an inverted index from the raw web data. Now MapReduce and its open source implementation Hadoop [18] have been used in many other disciplines.

There are two basic principles in MapReduce. One is data locality. The other is transparency as it hides the data location, load balancing and fault tolerance from developers. The former addresses performance. The latter addresses usability (ease/simplicity of use), which distinguishes MapReduce from more traditional grid or data flow systems. Programming distributed systems used to be very complicated due to the handling of these issues (e.g., message passing, locating, load balancing, and fault tolerance).

Hadoop MapReduce is the most popular implementation of the MapReduce framework. In this paper, when we refer to MapReduce, it represents the Hadoop design and implementation of MapReduce.

Like other programming models, MapReduce has its limitations too. It supports certain applications well but performs poorly with others. In addition, the limitations do not only come from the design, but also from its implementations (e.g., Hadoop). For many HPC applications, MapReduce, especially the Reduce operation, lacks flexibility and introduces negative impacts on performance and programmability. In addition, MapReduce only considers data locality on the single input dataset, but not multiple inputs and the output. While output location does not matter for a single round of MapReduce processing, it can have significant impact on multiple related MapReduce operations such as those iterative or combinative ones.

We have already designed and developed a similar framework called Sector/Sphere [7, 8, 23] that is comparable to MapReduce. Sector is the distributed file system, while Sphere is the parallel in-storage data processing framework that can be used to process data stored in Sector. Instead of MapReduce,

978-1-4244-9705-8/10/$26.00 ©2010 IEEE

Sphere applies arbitrary user-defined functions (UDF) on data segments in parallel, rather than fixed Map and Reduce operations. It also introduces a more efficient and flexible data exchanging pattern. Moreover, the framework does not only consider input data locality, but also output data locality. Overall, the new framework significantly improves the data processing performance and supports a much greater range of applications. Finally, it worth noting that writing a UDF is as simple as writing a Map or a Reduce function.

In this paper, we examine those critical issues in the MapReduce and Sphere UDF model that have significant impact on the performance and programmability of data intensive HPC applications. We have carefully chosen a specific and yet representative benchmark application for experimental studies: graph breadth-first search (BFS). Graph BFS is a very simple concept but contains a rich set of data processing and exchanging patterns that present non-trivial challenges to the framework. The performance study was conducted using graphs representing prevalent large real-world graphs.

The contribution of this paper includes:

- We analyze the critical components of parallel data processing framework with simplified interfaces such as MapReduce and Sphere UDF. we show that the UDF framework not only generalizes MapRecuce but also introduces arbitrary user defined functions and output locality optimization. The new framework significantly improves the flexibility and performance.
- We provide a high performance solution to searching massive size graphs using the framework. This solution is comparable to the best performance published so far but it is much easier to program and requires only inexpensive hardware.

The rest of the paper is organized in the following sections. Section 2 introduces this new parallel data processing framework and its comparison to MapReduce. Section 3 explains the graph BFS problem and describes the Sphere solution to the problem. Section 5 describes the detailed results from experimental studies. Section 6 briefly reviews related work. Section 7 summarizes the paper.

II. The Sphere Parallel Data Processing Framework

A. Overview

The Sphere programming framework consists of three components: data storage, data processing, and data exchanging pattern. The framework is depicted in Figure 1.

Data Storage

A distributed file system is required to support data access across nodes on distributed clusters. Sphere uses the Sector distributed file system as its data storage. Sector provides a uniform namespace with directory service across all nodes in the system. Sector uses replication to provide reliability.

In addition to regular files, Sector also supports in-memory objects. These objects are treated in the same way as regular files (in the sense of locating and access), except that they can be directly accessed by Sphere on the same node. In-memory object is an optimization to store the data structure in memory for repeated accessing.

Data Processing

The Sphere software daemon (called SPE, or Sphere Processing Engine) accepts one dataset as input, applies an arbitrary user-defined function (UDF) to each segment in the dataset, and generates a new dataset as the output.

Here the dataset refers to one or multiple files/directories that contain user data. Sphere splits the input dataset into multiple segments. Each segment may be a record, a group of records (in fixed or various number), or a file. The same UDF is applied to each data segment independently.

In comparison, MapReduce supports two specific data processing operations: Map and Reduce. Map can exist independently but Reduce can only run after a Map operation. Both Map and Reduce can be expressed by UDFs.

The Reduce operation presents certain limitations on programmability and performance. MapReduce requires a sort before the reduce operation. It is not necessary in many cases (e.g., a hash could work as well in order to group records with the same key). Sort is not required by a Sphere UDF, although the UDF can implement sort when necessary.

Data Exchanging

The results of each UDF can be written into a new set of Sector files. Sphere allows records in the result of each UDF to be sent to different files according to a user defined key in the UDF. The key is not part of the data record, but it is purely defined to indicate the destination file. As a consequence, results from multiple UDFs can be written into a common file.

In the Sphere terminology, we use the term "bucket" to represent a file in the output dataset, because the process is similar to a hashing process, where data is hashed into different buckets.

The bucket writers accept results from multiple SPEs and write into a single file on the local disk. A special routine can be put in the bucket to process the incoming results before writing them to the disk. This processing routine is called a bucket combiner.

The physical location of each bucket writer can be specified in order to provide better data locality in further processing. While MapReduce considers the location of the input data, it ignores the location of the output. We will discuss the output locality in Section 2.2.

MapReduce supports a similar data exchanging pattern. The Reducers reads the Map results, which are partitioned according to the keys. Although this "pull" model is logically equivalent to the "push" model that Sphere uses, in implementation the pull model performs more poorly because it requires writing twice.

The output locality support is an even more important feature that is missing in MapReduce. This is necessary to support iterative or combinative operations that process multiple input datasets.

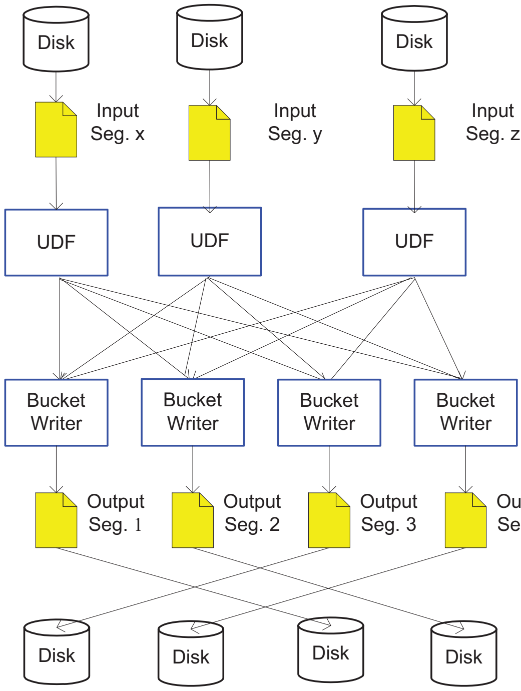

Figure 1. Sphere UDF/SPE and Bucket Output

Table 1 summarizes the core differences between the MapReduce and the Sphere frameworks.

Table 1. Comparison between the Hadoop MapReduce and the Sphere Frameworks

	Hadoop MapReduce	Sphere
Storage	Disk data	Disk data and in-memory objects
Processing	Map and Reduce, Reduce can only exist after Map	Arbitrary user defined functions
Data Exchanging	Reducers "pull" result from previous mappers	UDFs push results to various bucket files
Data Locality	Input data is assigned to the nearest Mapper (input locality)	Input data is assigned to the nearest UDF (input locality). Output data can be sent to specific locations to coordinate with other processing (output locality).

B. *Data Locality*

Data locality is the most important principle to consider when processing very large datasets across distributed clusters.

According to the data locality principle, for each data segment in the input dataset, Sphere always tries to locate an SPE on the node that contains the data; if this cannot be satisfied (e.g., an SPE would be idle while there are still more data segments to be processed), an available SPE that is nearest to the data is located.

Sphere is integrated with the Sector distributed file system. Sphere can learn the physical location of each data segment via Sector, and it can specify a particular physical node to write to, if enough space is available. In addition, Sector also supports the topology of the hardware system in order to measure the distance (e.g., number of hops) between any two nodes.

Under this basic data locality principle, Sphere schedules SPEs as follows:

- For a spare SPE, find if there is a local data segment to process; if yes, assign a local data segment to the SPE.
- If there is no local data to process, assign the SPE a data segment from the nearest node.
- If there are multiple files that satisfy the above data locality rules, choose the one that serves the least number of SPEs, because accessing the same file at the same time from many SPEs can cause a bottleneck.
- If there is one file that contains significantly more data segments that are not yet processed, this file should be processed first even if it breaks the first two rules, because otherwise at the end this could be the only file left to be finished and would significantly delay the whole processing.

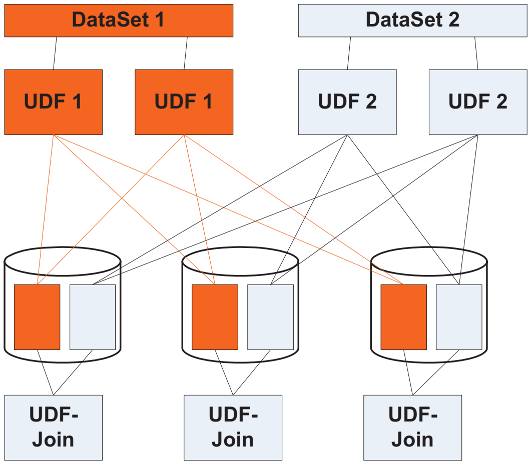

Figure 2. Join Operation with Sphere

In addition, a UDF may need to process data from multiple files, equivalent to accepting multiple inputs in the programming model. Performance can be significantly improved if the corresponding data segments from multiple input datasets are located together on the same nodes.

While the original inputs may be located on different nodes, the Sphere result datasets can be placed according to the data locality rule in order to optimize future processing. Sector/Sphere allows developers to specify the physical location of bucket files. This is why we need output locality.

For example, Sphere can support efficient "join" operations by performing iteration UDFs on multiple datasets independently but sending corresponding results to buckets on the same node (Figure 2). This effectively covers Map-Reduce-Merge [16], a MapReduce expansion that supports database operators.

As MapReduce does not prevent a Map function from reading multiple datasets, the "join" performance can be significantly lower when reading data from different locations. Note that the input datasets may not be necessarily matched sequentially. An iteration process is often required to scan the data first.

C. *Load Balancing and Fault Tolerance*

In most cases, the number of data segments is much greater than the number of SPEs, thus the system is naturally balanced: SPEs that process data faster (either due to better hardware or because of the input data) will process more data segments. The scheduling algorithm described in Section 2.2 guarantees that the data segments to be finished at the final stage of the processing are still uniformly scattered.

Jobs that failed on one SPE will be restarted on another, if the failure is node specific (e.g., due to hardware failure) rather than caused by data or code. If the error is caused by the input data or the code logic in the UDF, the data segment will not be processed again since it will fail again. If the same data segment has failed on multiple SPEs, it will not be executed again either, since it is likely that there is a logical error in UDF that was not checked.

When an SPE sends output to multiple buckets, fault tolerance may be rather complicated. Because a single failed data segment may "infect" more than one bucket, the origins of the data records in the buckets must be recoverable and traceable. However, maintaining the tracing information requires additional storage and processing time. (In MapReduce, the map result is written to independent intermediate files.)

An alternative is to not provide fault tolerance in this situation but leave it to the applications. For example, applications can split a job into multiple sub-tasks by splitting the input data. Only the failed subtasks need to be restarted, thus it reduces the impact of this type of failure.

In this paper, we focus on the programmability and performance related issues and will not include these aspects (load balancing and fault tolerance) in the experimental studies.

D. *Sphere vs. Hadoop MapReduce*

There are many available MapReduce implementations according to the principle described in [5], but with various implementation strategies. In this section we compare our own Sphere implementation and the Hadoop MapReduce implementation, which is the most popular MapReduce implementation.

Table 2 lists the steps of a Hadoop MapReduce process and the corresponding steps in the Sphere implementation.

1) Sphere uses record offset index files (that indicate the position of each record in the input file) to locate the position of each record; Hadoop requires run time parsing.

2) The Map operation can be expressed by a UDF in Sphere. Sphere does not explicitly use <key, value> pair. Instead, it treats all data as binary format and leaves UDF to handle the data format. Hadoop requires an explicit key, even if it is pseudo and for the purpose of partitioning only.

3) Sphere assigns a bucket ID to each record so that the resulting records can be sent to the proper buckets. This is equivalent to the partition function in Hadoop.

4) Hadoop has a "Combine" step to merge the same or relevant records. This can be done in the Sphere UDF.

5) Because Sphere pushes results to buckets directly, there is a bucket writer on each node to accept results from multiple UDFs. A bucket combiner can process the incoming records before writing them to the disk. This does not apply to Hadoop.

6) In Hadoop, the Reduce operation reads data from the previous Map operations, sorts the records, and performs a user defined operation on a group of records with the same key. This can be done equivalently in Sphere by running a UDF on the bucket files. Sort is not required in the Sphere UDF but it can be included if necessary.

Table 2. Comparing the Steps in MapReduce and Sphere

Sphere	**Hadoop MapReduce**
Record Offset Index	Parser / Input Reader
UDF	Map
included in UDF, assigning bucket ID to each record	Partition
<can be included in UDF>	Combine
Bucket Writer / Combiner	-
<can be included in UDF>	Compare
UDF	Reduce

Our Sphere implementation is written in C++. For application developers, they only need to specify the input dataset, the UDF (either system provided or user developed according to an interface specification), and the output directory. The output dataset can be the input of the next round of computation if necessary. Users can also specify parameters to the UDF. Other finer granularity controls are supported, such as how to split the input dataset, what is the processing unit (single record, file) of the UDF, etc.

III. Graph BFS and Its Sphere Solution

A. Graph Breadth-First Search

The graphs are commonly used in many HPC applications. Graphs of particular interest are so-called scale-free graphs [24] that include those prevalent graphs such as web graphs, citation graphs, and friendship networks, Applying various graph analysis techniques on these graphs can reveal valuable information. For example, in social network analysis, graphs in which vertices represent "person" or "organization" and edges represent relationships such as "work for" are analyzed to find groups of socially close individuals or those people who have strong influence on others.

The graph data structure can be used in many HPC applications. For example, in social network analysis, vertices can represent "person" or "organization", while edges can represent relationships such as "work for".

In this paper we choose the breadth first search (BFS) algorithm in graph analysis as the benchmark application for our parallel data processing engine. BFS is one of the fundamental approaches to many types of graph queries.

There are two types of BFS. Unidirectional BFS (Uni-BFS) starts from one given vertex, searching all its neighbors, then for every neighbor, search the neighbors' neighbors, until the search reaches the other given vertex. Figure 3 depicts the above process. In Figure 3, the search starts from vertex a and ends at vertex b.

For a given vertex, we use Level n to represent the collection of all of its n-th level of neighbors. The vertex itself is level 0. If b is on the n-th level of neighbors, then the shortest path between a and b is n.

Bidirectional BFS (Bi-BFS) starts search from vertex a and vertex b at the same time until a common vertex is reached on their search paths.

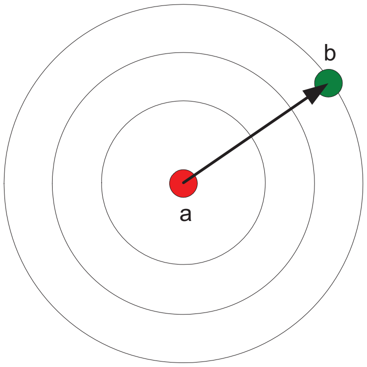

Figure 3. Unidirectional Breadth First Search

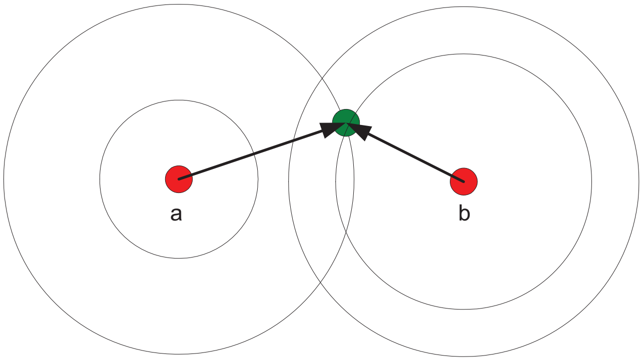

Figure 4. Bidirectional Breadth First Search

The graph BFS problem is a good benchmark application because it is simple and representative. With a simple problem, we can easily observe the internal mechanisms of the processing framework without the side effects of many variables in a more complex application.

In addition, it presents a more challenging problem than several other benchmark applications that are typically used, e.g., Terasort [5, 7, 8] and wordcount (inverted index) [5]. Both Terasort and wordcount require a single round of computation only (i.e., a Map operation followed by a Reduce operation, or two UDFs in Sphere). Graph BFS, however, requires iterative operations.

Although MapReduce implementations such as Hadoop support repeated execution of Map and Reduce procedures, it is not straightforward to program the Graph BFS problem in an efficient way, because:

1) In the BFS problem, information needs to be shared by all rounds of computation in order to avoid visiting a vertex more than once. Hadoop does not provide any mechanism to share information between different rounds of computation except for files, therefore a data structure has to be reconstructed every time.

2) The output of the current round of computation (level n neighbors) becomes the starting search set of the next round (level n+1 neighbors). However, the set can be very large and can consist of millions of vertices. Passing it as a parameter is very inefficient. Therefore, we have two input datasets now: the original graph data and the current level of vertices. Although the MapReduce model does not prevent multiple inputs, the data locality becomes a huge challenge since the two inputs can be located on different nodes and performance can be seriously damaged. (See discussion in Section 2.2.)

3) The data distribution is highly unbalanced. Real-world scale-free graphs usually follow a power law degree distribution with a small number of vertices connecting to most edges. Balancing the computation out of the unbalanced data presents yet another challenge.

Iterative processing is common in scientific computing and HPC areas. For example, many data mining algorithms, such as K-Means, require iterative processing on the input data.

Many have relied on conventional relational database systems to store and process large graphs mainly because they offer inexpensive large data management capability and ease of use. Although it is relatively straightforward to write graph algorithms including BFS in SQL, the resulting graph algorithms suffer extremely poor performance, since these relational database systems, designed primarily for transaction processing, cannot handle the excessive memory accesses and high communication volume involved in running large graph algorithms well. Such poor performance was evident even on a high-end parallel SQL machine [17].

Recently, researchers started investigating the use of dataflow systems, exemplified by Lexis-Nexis Data Analytic Supercomputer (DAS) [22], Microsoft Dryad [10], and Hadoop Pig [20], for running large graph algorithms. The dataflow model combined with well-designed underlying data management substrate enables a high degree of data parallelism. However, the dataflow model is inadequate for running graph algorithms, since it lacks control flow that is essential to support the aforementioned iterative processing of graph components.

B. The Sphere UDFs for Graph BFS

In this section we describe the Sphere implementation and the UDFs used to perform graph BFS.

The input graph represented by an adjacency list format and each edge appears twice in each vertex' list.

The dataset is split into m segments (segment 1 to segment m) according to vertex IDs. A single vertex' adjacency list only appears in a single segment. That is, the adjacency list for any given vertex will not be split into two segments.

The distribution of the vertex ID may vary and it will affect the processing performance. For example, we can have each segment containing approximately the same number of vertices or the same number of edges. The data segmentation can be done by a preprocessing that scans or samples the whole dataset.

Graph BFS is an iterative processing and each step produces the current level of neighbors (see Figure 3 and Figure 4). These partial results are written into Sector files as well. We use level.i.j to represent the j-th segment of the i-th level of neighbors. Each level of neighbors is split in a way so that all vertices in level.i.j must only appear in segment j of the original graph data. In fact, level.i.j is one bucket file generated by the UDF.

For Uni-BFS, two UDFs are applied repeatedly until the destination vertex is found or the search cannot reach another vertex. Recall that the UDF is applied to every input data segment.

UDF-UNI-1: Input data segment *j*, current level *i*.

For all vertices in level.*i*.*j*, find the next level of neighbors in segment *j* according to the adjacency list. Assign each vertex a key x according to the initial data segmentation so that the vertices in the result can be sent to proper buckets (level.*i*+1.x); however, repeated vertices in the result are discarded. The visited vertices are removed from the input data segment.

The results of the above UDF generates level *i*+1 (leve.*i*+1.1 .. level.*i*+1.*m*).

UDF-UNI-2: input bucket file level.*i*+1.*j*, current level *i*

Check if the destination vertex is in level.i+1.j. If found, the process can stop.

Note that UDF-UNI-2 is only a logical step. In the implementation, the checking of destination vertex can be done in UDF-UNI-1, which can perform the check at the same time of the scan. It can also be done in the bucket combiner function, which can check if the incoming results match the destination vertex.

In Bi-BFS, we need two sets of level files: level.*i*.*j*.forward and level.*i*.*j*.backward, which are used to represent the current expansion set in two directions, respectively. Similarly to Uni-BFS, two UDFs are applied repeatedly until a common vertex is found.

UDF-BI-1: input data segment *j*, current level *i*.

For all vertices in level.*i*.*j*.forward and level.*i*.*j*.backward, find the next level of neighbors, assign each vertex in the result a key according to the initial data segmentation, except for repeated vertices, which will be discarded. Visited vertices are removed from the input data.

Note that two copies of the input graph are required, and vertices are removed in each copy separately per direction. This can be done using a separate index file/data structure, instead of modifying the original file.

The results of UDF-BI-1 generate level *i*+1.

UDF-BI-2: input bucket file level.*x*.*j*.forward and level.*y*.*j*.backward, where *x*, *y* = 1..*i*+1. current level *i*

Compare level.forward and level.backward, if a common vertex is found, the process can stop.

C. Related Sphere Optimizations

In this Section we revisit some of the design and implementation issues described in Section 2.2 and 2.4 and explain how they are applied to the graph BFS problem.

The processes in Section 3.2 can be expressed by MapReduce as well. For example, the Uni-BFS problem can be solved with a loop of Map and Reduce operations. For each level, the Map operation works similarly to UDF-UNI-1, but writes the results to local disks, instead of sending them out to multiple buckets. In the Reduce operation, all the intermediate results will be read into each Reducer to perform an operation similar to UDF-UNI-2.

However, MapReduce (or the Hadoop implementation) lacks the following functionalities that would otherwise make the MapReduce significantly faster. On the other hand, these are easily addressed in Sphere.

First, Sphere supports in-memory objects. This can be used to store certain data structures between rounds of computations in memory. For example, in graph BFS, a data structure used to store index of graph adjacency list and mark visited vertices can be stored in memory. The structure is constantly changing and writing it to disk and reconstructing it every round is very time consuming. In fact, small input datasets that requires multiple rounds of processing can also be loaded into memory.

Support of in-memory objects is a common optimization in databases and data warehouses and in fact it can be critical for performance [13]. Hadoop does not support this yet, while data analytics systems based on Hadoop, such as Casandra [21], have to implement their own memory cache for data access optimization.

Second, Sphere allows users to specify the location of the output files (buckets). Users can access Sector files transparently to the physical location, but when necessary, they can specify location information for certain optimizations. For graph BFS, a UDF requires reading data from both the adjacency list and the level data. Reading one of them from a different node breaks the data locality rule and will hurt performance.

By specifying the location of each segment of the output files (level.i), the corresponding parts of the adjacency list and the level data will be put together on the same node. Consequentially, the UDF can read both inputs from local disk.

This also depends on the "push" model as Sphere can send the UDF result directly to the specified location. Since MapReduce does not support "push", the bucket location optimization does not apply to MapReduce.

D. *Implementation*

Using Sphere is very simple. We implemented the Uni-BFS program with a total of 440 lines of C++ code (300 lines of UDF and 140 lines of Sphere client application) and the Bi-BFS program with a total of 800 lines of code (600 lines of UDF and 200 lines of Sphere client application).

IV. EXPERIMENTAL STUDY

A. *The Input Data*

The graph datasets used for our experimental study are the original PubMed data [17] and a synthesized graph based on PubMed called PubMedEx.

PubMed consists of 28 million vertices and 542 million edges. The PubMed data is about 6GB.

PubMedEx replicates PubMed 183 times, assigns unique vertex IDs for each copy, and creates 20% additional random edges between vertices of different copies. PubMedEx consists of 5 billion vertices and 118 billion edges. The PubMedEx dataset is about 1.3TB in size.

Both datasets are preprocessed into the adjacency list in binary format.

B. *Testbed Configuration*

The experiments are performed on the Open Cloud Testbed [19], consisting of 8 racks (256 servers).

Each server has dual 4-core Intel Xeon E5410 2.33GHz, 16GB physical memory, 4X1TB RAID-0 disk, and 1Gb/s Ethernet connection. The inter-rack bandwidth is 10Gb/s.

C. *Performance Experiments*

We run both Uni-BFS and Bi-BFS on PubMed using 20 servers to check the performance impact of each component in the Sphere framework.

Table 3: Average Time Cost (seconds) on PubMed using 20 Servers

Length	Count	Percent	Avg Time Uni-BFS	Avg Time Bi-BFS
2	28	10.8	21	25
3	85	32.7	26	29
4	88	33.8	38	33
5	34	13.1	70	42
6	13	5	69	42
7	7	2.7	88	51
8	5	1.9	84	54
Total	260	Avg Time	40	33

Table 3 lists the average time cost (in seconds) for both Uni-BFS and Bi-BFS on PubMed using 20 servers. In Table 3, "Length" means the number of hops (distance between the input pair of vertices), "Count" means the number of pairs in the group, "Percent" is the percentage (Count/Total).

It is clear that more hops between two vertices usually require longer time to search, although the connectivity of a particular vertex along the path also plays an important role. As the number of hops increases, Bi-BFS requires less time than Uni-BFS since it searches from both ends, but the advantage is less obvious when the number of hops is small, and it can even take more time due to additional overhead.

The BFS algorithm has been widely implemented or supported by database, data warehouse, and dataflow systems. In a previous report on the performance of the LexisNexis Data Analytics Supercomputer (DAS), a commercial data flow engine, it took DAS 120 seconds to execute uni-BFS and 56 seconds to execute Bi-BFS, on the same dataset, with 20 slightly different servers.

DAS is a highly efficient data flow engine and it is already significantly faster than database and data warehouse systems. Performance comparison was given between DAS, Oracle and Netezza (a commercial data warehouse system), and DAS was faster than the other two by about two orders of magnitude.

In addition, the Sphere graph performance is close to MSSG, a system that was specially designed and optimized for graph data storage and processing. When caching all data in memory, MSSG was able to finish uni-BFS of the same PubMed graph between 10 - 50 seconds, for path length 4 - 6, respectively, on 16 servers with faster Infiniband connections [9].

We were unable to compare Sphere against DAS and MSSG directly due to software availability. It is also worth noting that writing applications using Sphere is in general much simpler than using data flow systems because Sphere provides transparent data location, load balancing, and fault tolerance.

In order to examine the performance impact of various components in Sphere, we turn on/off certain related components to compare the performance against the above setup, which we use as the base value. The results are listed in Table 4.

Table 4: Performance Impact of Various Components in Sphere

Components Change	Time Cost Change
Without in-memory object	117%
Without bucket location optimization	146%
With bucket combiner	106%
With bucket fault tolerance	110%
Data segmentation by the same number of vertices	118%

When we use disk file only instead of using in-memory objects to store repeated accessed data structure, the processing time increases to 117%. The bucket location optimization provides the most significant performance boost. When writing the bucket files at random locations, each UDF reads (with a high probability) the bucket data from other nodes and the processing time increases to 146%. As mentioned above, the bucket combiner in this application does not help, but slightly increases the processing time to 106%. Enabling fault tolerance on bucket files (by recording the original input data segment of all incoming data records), the processing time is increased by 10%.

Data segmentation also has an impact on the performance as it affects the load balancing on different UDFs. If we change the segmentation from the same number of edges per segment to the same number of vertices per segment, the processing time is increased by 18%.

Although these values are application specific, they still give us a deep insight on how these design and implementation issues can affect Sphere and other MapReduce style frameworks. To the best of our knowledge, there have been few other similar experimental studies.

D. Scalability Experiments

We run the BI-BFS program on a much larger scale, using PubMedEx with 60 servers. The result can be found in Table 5.

Table 5: The Average Time Cost (seconds) on PubMedEx using 60 Servers

Length	Count	Percent	Avg Time
2	11	4.2	56
3	1	0.4	82
4	60	23.2	79
5	141	54.2	197
6	45	17.3	144
7	2	0.7	201
Total	260	Avg Time	156

Note that the search time is not proportional to the data size as only part of the graph on the search path is scanned. However, considering PubMedEx is 180 times bigger and the number of servers is only three times of that used in Table 3, the performance is very good.

We further examine the scalability performance of Sphere by running 100 Bi-BFS queries on PubMedEx using 19, 41, 59, and 83 servers, respectively. These servers are located on 1, 2, 3, and 4 racks respectively in each group. The result is listed in Table 6.

Table 6: The Average Time Cost (seconds) on PubMedEx on 19, 41, 61, 83 Servers

Group	3	4	5	6	7	
Count	1	24	58	16	1	
Servers#						**AVG**
19	112	257	274	327	152	275
41	153	174	174	165	280	174
59	184	150	157	140	124	153
83	214	145	146	138	192	147

In this experiment, there is only one query that falls in group 3 and group 7, so we can omit these two results due to the small sample size. Based on the results of group 4, 5, and 6, we can see a clear pattern that using more servers reduces the processing time. However, as the number of servers increases, the performance acceleration decreases, because the impact of unbalanced data on the search path becomes more and more significant.

E. Concurrency Experiments

In the real world, users may submit a large number of concurrent requests at any time, such as the Google search engine. We updated the BFS program to a more realistic scenario by searching multiple pairs of vertices at the same time.

We insert a tag for each vertex in the bucket files (level.i.j) to identify which pair's search path it is on. Neighbors of different vertices are scanned in one UDF.

The following result in Table 7 is obtained using PubMedEx and 83 servers. We run 5, 10, 20 and 50 Bi-BFS queries each time. These are the same queries used in the experiments in Section 4.4. As we run more queries together, the average time cost per query decreases, meaning that we can effectively support concurrent queries in real world scenarios.

Table 7: Average Time Cost (seconds) for Multiple Concurrent Queries on PubMedEx using 83 Servers

Queries#	5	10	20	50
Time per Run	324	373	469	805
Avg per Query	64.8	37.3	23.5	16.1

We can further expand our solution to more problems in graph search. For example, we can update the code in Section 4.5 to search the distance between any pair of vertices in a group. We can also develop a more advanced version of this program to process queries coming continuously by adding a level parameter to each pair so that in each round of the computation, a UDF may scan different levels on the search paths of different query pairs. These solutions would be more domain specific and beyond the scope of this paper. Nevertheless, Sphere is very effective for these HPC and large scale data analytics applications.

V. Experimental Study

We have been working on the design and implementation of a generic simplified parallel data processing framework in the last several years. In previous work, we have explained the Sector distributed file system and the early version of Sphere system [7, 8]. We have compared the performance of the previous versions of Sector/Sphere and Hadoop using the Terasort and MalStone benchmark. Sphere is about 2 - 20 times faster in these benchmarks [1, 7, 8]. In this paper, we formalize the Sphere framework and introduce new concepts such as output locality. We also introduce the graph BFS benchmark that is more challenging to the framework than TeraSort and MalStone. Graph BFS requires iterative processing and for each step it reads two input datasets. Our result is comparable to previously published results using DAS [17] and MSSG [9], which outperformed traditional database and data warehouse systems by two orders of magnitude.

To date, there has been a great deal of work investigating using MapReduce in various HPC applications, but little has been done to analyze and expand the MapReduce framework itself. Map-Reduce-Merge [16] is one of the few projects to expand the framework to suit more applications. Map-Reduce-Merge adds a "merge" phase that basically joins multiple reduce outputs, which is a common operation in databases. In Sphere, this Merge phase can also be represented by a UDF. Another UDF with buckets output can be used to program how to iterate and match between multiple input datasets (see Section 2.2).

It is worth noting that Google has developed a new system called Pregel [25] to process their graph data structures, which apparently have addressed many shortcomings of the MapReduce framework, but little details of Pregel has published so far.

In the (not-strictly-defined) cloud computing concept, systems like Sphere and MapReduce are commonly categorized as the compute cloud in the PaaS (Platform as a Service), in addition to storage cloud (e.g., Sector, Google File System [6]) and data cloud (e.g., BigTable [4]). The other two cloud system categories are IaaS (Infrastructure as a Service) and SaaS (Software as a Service).

DataCutter [2] and Active Data Repository (ADR) [12] 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 provide relatively low level data flow service. They do not have a high level programming model that supports inexplicit data exchanging (such as the Reduce operation), load balancing, and fault tolerance. Developers need to program explicit data flows to connect/schedule the filters. This can be very complicated especially when load balancing and fault tolerance are considered. In contrast, a Sphere application treats the whole system as a single entity and data movement, load balancing, and fault tolerance are transparent to users.

Another related group of systems are grid schedulers such as Condor [15], Swift [14], Dryad [10], etc. These grid scheduling systems allow grid users to schedule multiple tasks by defining their relationships, commonly with directed acyclic graphs (DAG). The tasks are usually loosely coupled or even completely independent. In contrast, Sphere and MapReduce are data driven. A Sphere job is a single task but sub-tasks are created according to data segmentation. In contrast, grid scheduling is task driven.

Other systems that consider data placement and transfer cost as a factor in batch task schedulers include BAD-FS [3] and Stork [11]. These systems are remotely comparable to Sphere but they have separated storage and computing sub-systems.

Because Sphere and MapReduce are designed to process very large datasets, many data analytics applications and systems have been built on top of them or similar principles. In fact, certain optimizations in Sphere also appear in many database and data warehouse systems. However, databases do not handle the size of data that Sphere and MapReduce can handle.

VI. Conclusions and Future Word

We have presented Sphere, a parallel data processing framework that generalizes the MapReduce programming model, and a micro-level analysis on its performance using the graph BFS problem as the benchmark application. We have shown that, despite its popularity, MapReduce is not necessarily the only choice as a simple way to program distributed clusters. Sphere introduces three major improvements:

- Spheres uses arbitrary UDFs to replace Map and Reduce and it eliminates the limitations of Reduce.
- The Sphere data exchanging model is more efficient and flexible than the "pull" model between Map and Reduce.
- Sphere introduces output locality that effectively address the data locality for combinative and iterative operations.

Our experimental study has shown that the design and implementation of Sector/Sphere provide an efficient and effective support to data intensive applications, including the iterative processing that MapReduce implementations (e.g., Hadoop) cannot handle well.

In addition, the graph BFS problem can be a good benchmark on distributed systems that are designed to support data intensive computing. Compared to other benchmarks such as Terasort, graph BFS presents other non-trivial challenges on data distribution, exchanging, and locating.

We plan to implement two additional function layers on top of Sphere. The first layer will provide database operators such as projection, join, selection, product, etc. The second layer will provide a higher level of data analytics API.

Sphere has been released as open source software and is available together with Sector from http://sector.sf.net.

ACKNOWLEDGMENT

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.

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] Michael D. Beynon, Tahsin Kurc, Umit Catalyurek, Chialin Chang, Alan Sussman, and Joel Saltz, 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] Yunhong Gu, Robert Grossman, Sector and Sphere: The Design and Implementation of a High Performance Data Cloud, Theme Issue of the Philosophical Transactions of the Royal Society A: Crossing Boundaries: Computational Science, E-Science and Global E-Infrastructure, 28 June 2009, vol. 367, no. 1897, page 2429-2445.

[8] Yunhong Gu and Robert Grossman, Lessons Learned From a Year's Worth of Benchmarks of Large Data Clouds, Workshop on Many-task Computing on Grids and Supercomputers (MTAGS), co-located with SC09, Portland, OR. Nov. 2009.

[9] Timothy D. R. Hartley, Ümit V. Çatalyürek, Füsun Özgüner, Andy Yoo, Scott Kohn, Keith W. Henderson: MSSG: A Framework for Massive-Scale Semantic Graphs. CLUSTER 2006.

[10] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, Dennis Fetterly, Dryad: Distributed Data-Parallel Programs from Sequential Building Blocks, Proceedings of the 2007 Eurosys Conference.

[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] Hasso Plattner: A Common Database Approach for OLTP and OLAP Using an In-Memory Column Database, Proceedings of the 35th SIGMOD International Conference on Management of Data, Providence, Rhode Island, 2009

[14] 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.

[15] 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.

[16] Hung-Chih Yang, Ali Dasdan, Ruey-Lung Hsiao, and D. Stott Parker, Map-Reduce-Merge: Simplified Relational Data Processing on Large Clusters, in Proc. of ACM SIGMOD, pp. 1029–1040, 2007.

[17] Andy Yoo, Ian Kaplan: Evaluating use of data flow systems for large graph analysis. SC-MTAGS 2009

[18] Hadoop, http://hadoop.apache.org/

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

[20] Hadoop Pig, http://hadoop.apache.org/pig/

[21] Cassandra, http://cassandra.apache.org/.

[22] Lexis-Nexis Data Analytic Supercomputer, http://www.lexisnexis.com/government/solutions/intelligence/supercomputer.aspx

[23] Sector/Sphere, http://sector.sourceforge.net

[24] Barabasi AL, Albert R., Emergence of scaling in random networks. Science. 1999 Oct 15;286(5439):509-12.

[25] Grzegorz Malewicz, Matthew H. Austern, Aart J. C. Bik, James C. Dehnert, Ilan Horn, Naty Leiser, Grzegorz Czajkowski, Pregel: a system for large-scale graph processing, Sigmod '10.

[26] 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.