SSS: An Implementation of Key-value Store based MapReduce Framework

Hirotaka Ogawa, Hidemoto Nakada, Ryousei Takano, and Tomohiro Kudoh
Information Technology Research Institute
National Institute of Advanced Industrial Science and Technology (AIST)
Akihabara Dai Bldg 11th Floor, 1-18-13 Sotokanda,
Chiyoda-ku, Tokyo 101–0021, Japan
Email: h-ogawa@aist.go.jp

***Abstract*—MapReduce has been very successful in implementing large-scale data-intensive applications. Because of its simple programming model, MapReduce has also begun being utilized as a programming tool for more general distributed and parallel applications, e.g., HPC applications. However, its applicability is limited due to relatively inefficient runtime performance and hence insufficient support for flexible workflow. In particular, the performance problem is not negligible in iterative MapReduce applications. On the other hand, today, HPC community is going to be able to utilize very fast and energy-efficient Solid State Drives (SSDs) with 10 Gbit/sec-class read/write performance. This fact leads us to the possibility to develop "High-Performance MapReduce", so called. From this perspective, we have been developing a new MapReduce framework called "SSS" based on distributed key-value store (KVS). In this paper, we first discuss the limitations of existing MapReduce implementations and present the design and implementation of SSS. Although our implementation of SSS is still in a prototype stage, we conduct two benchmarks for comparing the performance of SSS and Hadoop. The results indicate that SSS performs 1-10 times faster than Hadoop.**

Keywords: MapReduce, Cloud Technologies, Key-value Store

I. Introduction

The practical needs of efficient execution of large-scale data-intensive applications propel the research and development of Data-Intensive Scalable Computing (DISC) systems, which manage, process, and store massive datasets in a distributed manner. MapReduce[1], Hadoop MapReduce[2], Dryad[3], and Sphere[4] are well-known as such DISC systems.

While MapReduce programming model essentially realizes global and integrated data-parallel operations to very large amount of key-value data, existing implementations mainly focus on performing single-step MapReduce with better fault-tolerance. Thereby, all input/output data are stored as a set of text files or serialized structured data in the shared and distributed file systems, such as Google File System (GFS)[5], Hadoop Distributed File System (HDFS)[6], and Sector[4].

There are two issues in these existing implementations. First, while each map and reduce program handles key-value data, the distributed file systems essentially handle large data files, which are eventually divided into chunks. In other words, there is a semantic gap between the MapReduce data model and the input/output data stored in the data store. Second, iterative MapReduce applications, which create new map and reduce tasks in each iteration, have to read and write any data in the data store repetitively. Furthermore, the intermediate data that one map task creates cannot be reused from other map/reduce tasks. These two issues are not so problematic for single-step MapReduce run, however, they incur a considerable overhead especially for iterative applications of MapReduce. Therefore, we can say that existing implementations limit the range of applications that MapReduce can be utilized.

There have been several work in realizing faster iterative MapReduce computations, e.g., Twister[7] provides a distributed in-memory MapReduce runtime, and Spark[8] redesigns MapReduce framework by using simple abstractions, such as resilient distributed datasets (RDDs), broadcast variables, and accumulators.

On the other hand, today, HPC community is going to be able to utilize very fast and energy-efficient Solid State Drives (SSDs). Especially, high-end SSDs connected with PCI-Express interface, such as Fusion-io's ioDrive^{TM}[9] duo, have 10 Gbit/sec-class read/write performance, equivalent to 10x of high-end HDDs and 1/10 of the main memory. This fact leads us to the possibility to develop more straightforward but efficient implementation of MapReduce, which can bridge the gap between MapReduce data model and input/output data model, and can sustain not only single-step workloads but also more flexible workloads, including iterative ones. From this perspective, we have been developing a new MapReduce prototype system called "SSS", which is based on distributed key-value store (KVS).

SSS completely substitutes distributed KVS for the distributed file systems such as GFS/HDFS. Furthermore, SSS utilizes distributed KVS for storing the intermediate key-value data, as well as the inputs of map tasks and the outputs of reduce tasks. SSS offers several advantages over existing MapReduce implementations:

1) As mentioned earlier, we can bridge the gap between MapReduce data model and storage systems and handle key-value data more intuitively.
2) We can eliminate shuffle & sort phase which may occupy most of the execution time of MapReduce depending on workloads. Once all map tasks have finished storing intermediate key-value data to the distributed

978-0-7695-4302-4/10 $26.00 © 2010 IEEE
DOI 10.1109/CloudCom.2010.89

KVS, all intermediate key-value data have already been grouped by intermediate keys.

3) Map and reduce tasks can be almost equivalent operations on distributed KVS as we will describe in Section II. This makes the implementation itself simple and enables any combination of multiple maps and reduces in a single workload.

SSS prototype is implemented in Java and provides a programming interface almost same as Hadoop MapReduce. And, at this moment, SSS uses an existing key-value store implementation, called Tokyo Tyrant[10], in order to realize a distributed key-value store. Tokyo Tyrant is known as one of the fastest key-value database implementations for single node.

Although our implementation of SSS is still in a prototype stage, we show that SSS can outperform Hadoop MapReduce by 1-10 times in Word Count and an iterative composite benchmark.

This paper is organized as follows: Section II brief introduces MapReduce, and Section III describes the conceptual overview of our key-value based MapReduce. Section IV describes detailed implementation of SSS and discussions, and Section V presents early results.

II. MapReduce

MapReduce is a distributed computing model which takes a list of input key-value pairs, and produces a list of output key-value pairs. MapReduce computation consists of two user defined functions: map and reduce. Map function takes an input key-value pair and produces zero or more intermediate key-value pairs. MapReduce runtime groups together all intermediate values associated with the same intermediate key and hands them over to the reduce function. Reduce function takes an intermediate key and a list of values associated with the key and produces zero or more results.

Conceptually, the map and reduce functions have associated types:

- map: (k1, v1) → (k2, list(v2))
- reduce: (k2, list(v2)) → list(v3)

This can be easily generalized as below:

- map: (k1, singleton-list(v1)) → (k2, list(v2))
- reduce: (k2, list(v2)) → (k2, list(v3))

Therefore, map and reduce functions have almost same associated types.

Google and Hadoop MapReduce implementations[1][2] focus on utilizing distributed file systems, namely Google File System (GFS)[5] and Hadoop Distributed File System (HDFS)[6].

Figure 1 shows the overview of MapReduce execution on these systems. Map tasks are distributed across multiple servers by automatically partitioning the input data info a set of M splits. Map tasks processes each input splits in parallel and generates the intermediate key-value pairs. The intermediate key-value pairs are sorted by the key and partitioned into R regions by the partitioning function (e.g., hash(key) mod R).

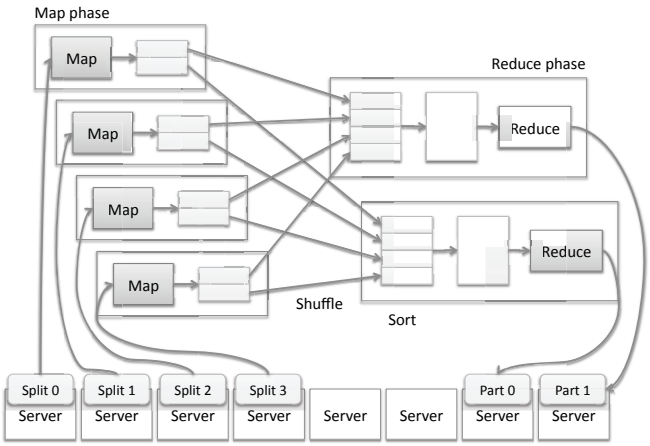

Fig. 1. MapReduce Execution

Reduce tasks are also distributed across multiple servers and process each partitioned region and generate R output files.

Both of these runtimes consist of a single master and multiple workers. While the master manages job requests from clients and dynamically schedules map/reduce tasks into idle workers, each workers handles an assinged map/reduce task. In addition, both implementations utilize the distributed file systems in their runtimes. These file systems employ the local disks of the worker nodes, which can be used to coallocate data and computation. Therefore, their approach provides the automatic load-balancing based on dynamic scheduling of map/reduce tasks, as well as the assignment of the initial map tasks based on the data locality.

However, there are two common issues in these implementations. First, there is a semantic gap between the MapReduce data model and the input/output data stored in the distributed file systems. Each map and reduce tasks handles key-value data, on the other hand, the distributed file systems only provide the feature to access large data files. Second, iterative MapReduce applications, which create new map and reduce tasks in each iteration, have to read and write any data in the distributed file systems repetitively. In addition, the intermediate data that a map task creates cannot be reused from any other map/reduce tasks.

These two issues are not so problematic for single-step MapReduce run, where existing implementations target to, but are serious for more flexible MapReduce applications, in particular iterative applications. Because they incur a considerable overhead for reading and writing large amount of data in the distributed file systems repetitively. Therefore, existing implementations may limit the range of applications that MapReduce can be utilized. But, potentially, MapReduce can be utilized as a programming tool for broader range of HPC applications including iterative numerical algorithms, if these defects are resolved.

III. Proposed Key-value Store based MapReduce

In order to resolve the above mentioned issues and effectively exploit the I/O performance of 10 Gbit/sec-class local SSDs, we have been designing and implementing a new MapReduce prototype system called "SSS", which is based on distributed key-value store (KVS).

SSS completely substitutes distributed KVS for the distributed file systems such as GFS/HDFS. SSS utilizes distributed KVS for not only storing the inputs of map tasks and the output of reduce tasks, but also storing the intermediate key-value data.

In this section, we will outline the conceptual design of SSS.

A. Key-value Store

Key-value store is known as a kind of database that holds data as a pair of key and value. Traditional database systems, such as RDBMS, can have multiple values for a piece of data as columns, but they lack the scalability to handle a very large number of data. On the other hand, key-value pairs can be divided by the ranges of the keys or the hash values of the keys, and distributed across multiple key-value store servers easily. Thereby, key-value store can accomplish "scale-out" and contribute significantly to read/write performance, capacity, and fault-tolerance.

While several implementations of distributed KVS already have been proposed, we have built our own distributed KVS based on an existing single node implementation, which can store key-value data to disks persistently. At this moment, we employ an existing implementation, but in future, we will to specialize our key-value store to data-intensive computation and exploit read/write performance of flash memory storage.

There are two key design requirements for our KVS: dynamic partitioning and supporting duplicated keys.

- Dynamic partitioning
  Our KVS requires a mechanism to dynamically partition the key-value data across the set of storage nodes. We partition the keys based on consistent hashing[11].
- Supporting duplicated keys
  Our KVS requires a mechanism to handle multiple key-value pairs with the same key. Because, map tasks usually emit multiple key-value pairs with the same key, e.g., counting the occurrence for the same word in multiple documents. We introduce the local key just for identifying each key-value pair with a key.

To satisfy these requirements, our key-value store handles triples (namely, *key*, *local-key*, and *value*) as key-value data, and distributes them across multiple storage nodes by using consistent hashing on *key*s of the triples, as Figure 2 shows. Clients make put/get requests to the target storage node which is decided from the hash value of the *key*, and each storage node handles requests and loads/stores triples.

B. SSS MapReduce

Building on top the distributed KVS mentioned above, we can greatly simplify the architecture of MapReduce framework. That is, map/reduce tasks can store their input/output data, including the intermediate key-value data, in the distributed KVS. And, they input and output only the list of the keys of the key-value data.

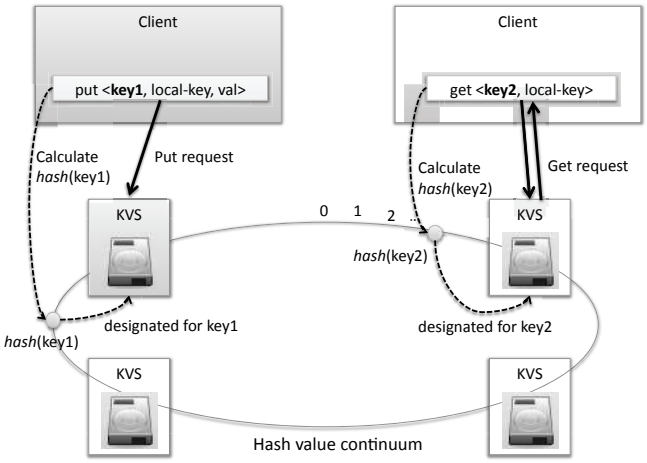

Fig. 2. Our Distributed Key-value Store

As we show in Figure 3, the outlined process of SSS MapReduce is as follows:

- Map
  Map tasks take each input key and load the key-value data from the key-value store in parallel, and generates the intermediate key-value data. Generated intermediate data are also stored in the key-value store, and only their keys are passed to reduce tasks.
- Shuffle & Sort
  Shuffle & Sort phase is not required. Key-value store provides the feature to look up key-value data from a given key, that means, it can automatically and internally sort and group the intermediate key-value data by their own keys.
- Reduce
  As well as Map tasks, reduce tasks take each intermediate key and load the key-value data in parallel, and generates the output key-value data. Generated output data are also stored in the key-value store, and only their keys are passed to clients.

IV. Implementation

SSS implementation consists of mainly two parts: key-value store, and MapReduce runtime.

A. SSS Key-Value Store

For key-value store, SSS employs an existing key-value store implementation, called Tokyo Tyrant[10]. Tokyo Tyrant is known as one of the fastest key-value database implementations for a single node. Prior to the implementation, we performed precise benchmarks for comparing the performance of 3 different key-value store implementations, namely Tokyo Tyrant which uses a database specialized to key-value data[12], MemcacheDB[13] which uses BerkeleyDB[14], and

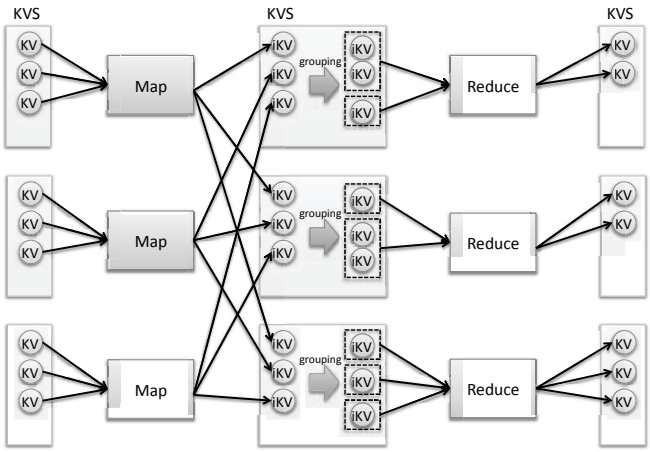

Fig. 3. KVS-centric MapReduce

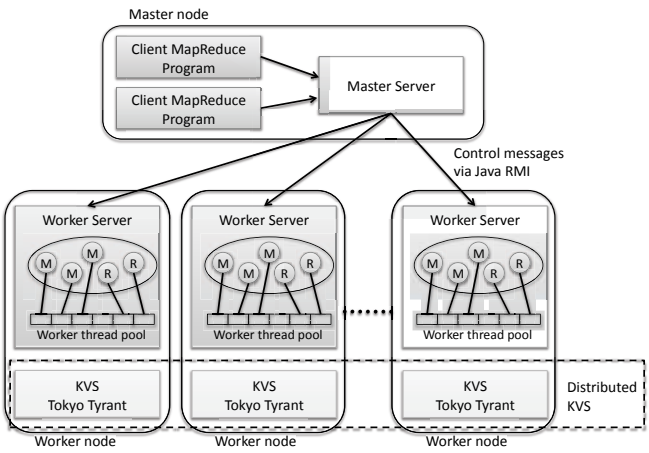

Fig. 4. SSS MapReduce Runtime

chunkd[15] which uses local file system. For the sake of brevity, we cannot cover the detailed results in this paper, but Tokyo Tyrant has better scalability for multiple threads and lower latency for reading/writing key-value data than other two.

To realize our distributed key-value store, we configure Tokyo Tyrant on each storage node, distribute key-value data across them by using consistent hashing on the *key*. Every client can make put/get/put_list/get_list requests to the target Tokyo Tyrant node which is determined by the hash value of the *key*. And, as we described at III-A, we need to handle triples as key-value data. Therefore, we extend the key format of Tokyo Tyrant to be able to include the *key* itself with additional information as below:

- *session-id* indicates the application and the phase of computation, by which this key-value is used.
- *local-key*: indicates the *local-key* to handle duplicated keys.
- *hash(key)*: indicates the hash value of the key. It is useful to partition key-value sets in a single node into several chunks based on this hash value. For example, to process the key-value data by multiple threads, we can partition them by the MSB of the hash value and perform map/reduce tasks for each partition in each thread.

For client-side library, we employ a Java binding of Tokyo Tyrant called jtokyotyrant[16]. Each client takes a list of storage nodes from the master node, and locally calculates the hash values of the keys and decides the target storage node based on consistent-hashing.

B. SSS MapReduce Runtime

SSS MapReduce runtime is built on top of distributed key-value store above mentioned. Each worker node serves both as a storage node of KVS and a worker server of MapReduce runtime. That is, each worker node is responsible for handling and executing map/reduce tasks for the key-value data owned by itself. While all input/output data of map/reduce tasks are provided by distributed KVS, other commands and control data transfers to the worker servers are performed by SSS MapReduce runtime.

As Figure 4 shows, SSS MapReduce runtime comprises of two main entities, (1) master server that manages the entire MapReduce computations, and (2) worker server running on every worker node.

Initializing MapReduce runtime, master server invokes a worker server on each worker node, which establishes a connection with master server to receive commands and control data. Each worker server, on its own responsibility, handles map/reduce tasks assigned to the target worker node, maintains worker thread pool to execute map/reduce tasks properly, and notifies the status to master server. Master server provides a programming interface to the clients, which is almost compatible with Hadoop MapReduce, and sends commands and control data to the worker servers running on each worker nodes.

To execute a map task, each worker server consumes a URI list, a mapper class name, a partitioner class name, and a combiner class name, provided by the master server. A URI list specifies a set of key-value data which is stored in the local KVS and processed in the map task, and each URI includes information, e.g., a host name, *session-id*, and *hash(key)*. Worker server allocates a worker thread from the worker thread pool to process an assigned URI, and each worker thread reads key-value data from the local KVS based on the information included in the URI. Then, the worker thread performs *map* operation (specified by the mapper class name) on the target key-value data and generates the intermediate key-value data. The intermediate key-value data are divided into partitions by using the partitioner class, where the number of partitions is equal to the number of worker nodes. And, if the combiner class name is given, *combine* operation is performed on the key-value data in each partition. After that, the worker thread stores the partitioned intermediate key-value data to the distributed KVS.

Internally, these processes are performed using pipelining technique so as to limit the memory usage and hide the

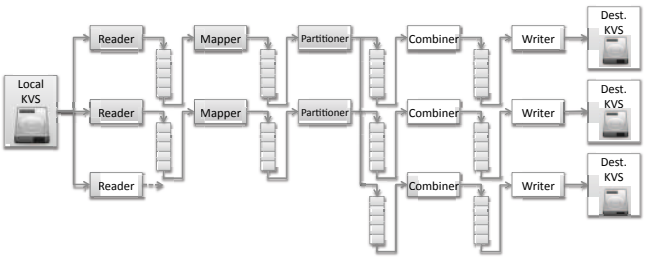

Fig. 5. Pipeline of a worker thread for a map operation

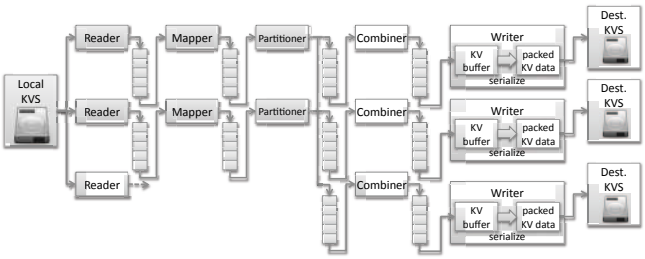

Fig. 6. Pipeline of a worker thread for a map operation (packed)

latency of KVS. Figure 5 shows the internal architecture of the worker thread. Each worker comprises of one or more reader, mapper, partitioner, combiner, and writer threads. All these threads cooperate with each other through queues from left to right, and in right end, writer threads store partitions of the intermediate key-value data successively.

After storing the intermediate key-value data, the worker server returns a state and a URI list which specifies the key-value data.

To execute a reduce task, each worker server consumes a URI list and a reducer class name, and processes them almost same as a map task.

Worker servers are also responsible for handling utility command requests, such as inspecting the state of the server and threads, profiling the key-value store usage, and so on.

C. Packed-SSS Runtime

We also realize a variant of SSS MapReduce runtime, called Packed-SSS (Figure 6). In this runtime, each writer stores the intermediate key-value data that the preceding combiner generates, into a memory buffer. After that, each writer converts them into a single large key-value data and stores it to the target key-value store.

Packed-SSS makes the number of the intermediate key-value data considerably small, which is actually equal to the product of the number of map worker threads by that of reduce worker nodes, and minimizes the overhead of accessing key-value stores. However, it almost requires twice as much as memory for packing all of the intermediate key-value data. Therefore, it may limit the size of target applications.

D. Discussions

We know of several advantages and disadvantages in implementing MapReduce relying on the key-value store.

1) Advantages:

- Map/reduce tasks are assigned to the storage node based on the data locality. When a key is given, consistent hashing allows us to determine uniquely the storage node where a set of key-value data with the key are stored. On the other hand, Google and Hadoop implementations take into account only the distribution of the input data for map tasks.
- Shuffle & sort phase is omitted, because the intermediate data are already sorted and grouped in the key-value store. Unlike SSS, existing implementations require sorting the intermediate keys in each worker server for *map*, passing them to worker servers for *reduce*, and then grouping them by the intermediate keys.
- Map and reduce tasks can be almost equivalent operations on distributed KVS. This makes the implementation itself simple, and enables any combination of multiple maps and reduces in a single workload. Especially, iterative algorithms such as data clustering, machine learning, and computer vision, are expected to be easily supported.
- Tokyo Tyrant, the key-value store we employed, provides aggressive caching mechanism. Thereby, we can reuse memory-cached key-value data across multiple map/reduce operations. At this moment, the cache interference problem still exists because of the lack of the mechanism to control which data should be cached. However, we plan to extend our framework to provide a user-directed method to specify key-value data cacheable or non-cacheable. This approach is more straightforward and transparent to users, compared to facilitating a kind of hierarchical data store which Twister[7] and Spark[8] provide.

2) Disadvantages:

- The key-value store requires random accesses to local disks, while the distributed file system only requires sequential accesses to the file chunks. In general, random accesses are much slower than sequential accesses. Using flash memory storage, however, random read is fast enough. Random write is also fast enough in high-end products, such as ioDrive™.
- The granularity of dynamic partitioning based on *key* is relatively small, therefore the degradation of the runtime performance will become an issue. However, when processing very large number of small files in existing implementations, the overhead of the task management may become huge, similar to ours.

  As we described at IV-C, we have also implemented a variant of SSS (called packed-SSS), which packs multiple key-value data into a single key-value data before storing key-value store. This makes the number of the intermediate key-value data small considerably. In Section V, we will show benchmark results including this version.
- There is no file system interface to our key-value store, unlike existing distributed file systems. However, we think that it is relatively easy to realize FUSE

interface to our key-value store, in the same way as CouchDB-FUSE[17] provides a FUSE interface to Apache CouchDB.

On the other hand, SSS cannot utilize key-value data stored in existing distributed file systems. This problem can also be resolved by providing KVS interface adaptors to file systems, however, we put higher priorities on the performance of the underlying storage systems than their availability.

- SSS doesn't have enough considerations to fault-tolerance. As for the underlying storage systems, HDFS and existing cluster file systems provide fault-tolerance to block losses with node failure, while SSS doesn't provide such features at this moment. We will realize fault-tolerance features referred to highly-available KVS implementations, e.g., Dynamo[18] and Cassandra[19].

  As for the runtime, in Hadoop, when one of mappers and reducers fails, it can re-execute independently of other workers. In SSS, however, when one of map tasks fails, the intermediate key-value data in distributed KVS will be inconsistent. We plan to support rollback and re-execution mechanisms based on *session-id* in the near future.

V. Evaluation

Although our implementation of SSS is still in a prototype stage, we performed two benchmarks, Word Count and an iterative composite benchmark, and presents early results. These benchmarks are not real-life iterative and/or multiple-step applications and not sufficient to prove the effectiveness of our approach. We plan to conduct more comprehensive application benchmarks including iterative K-means clustering, that have to be left for future investigations.

For these benchmarks, we used an experimental cluster which consists of a master node and 16 worker nodes with 10Gbit Ethernet and ioDrive^{TM}Duo, as shown in Table I. We use Apache Hadoop version 0.20.2, SSS, and packed-SSS for comparison purpose. Packed-SSS is an optimized version of SSS, which packs multiple key-value data into a single key-value data before storing key-value store. In packed-SSS, map tasks can considerably decrease the number of the intermediate key-value data created, on the other hand, reduce tasks require more memory to merge keys than original SSS.

TABLE I
Benchmarking Environment

Number of nodes	17 (*)
CPU	Intel(R) Xeon(R) W5590 3.33GHz
Number of CPU per node	2
Number of Cores per CPU	4
Memory per node	48GB
Operating System	CentOS 5.5 x86_64
Storage	Fusion-io ioDrive Duo 320GB
Network Interface	Mellanox ConnextX-II 10G Adapter
Network Switch	Cisco Nexus 5010

(*) one node is reserved for the master server.

Both Hadoop and SSS use JDK 1.6.0_20 (64bit), and SSS and packed-SSS use Tokyo Tyrant 1.1.41 and Tokyo Cabinet 1.4.46. To avoid the effect of unintended replications, the replica count of Hadoop HDFS is configured to be 1.

A. Word Count

This benchmark counts the number of occurrences of each word appeared in given multiple text files. We prepared 128 text files, and 4 text files are assigned to each worker node and processed in parallel. Each text file is 100MiB, therefore the total size of all files is 12.5GiB and each worker node has 800MiB text data. For SSS and packed-SSS, each files are stored as a key-value data (that is, the key is file name, the value is file content), and for Hadoop, each files are stored as an HDFS file. In fairness, we set the block size of HDFS to 256MB, to store each file in a single HDFS DataNode and process it by the local Mapper without fail.

Prior to the benchmark, we measured the times to transfer files from a disk of master node to SSS distributed KVS and HDFS. The result is that SSS takes 148.4sec (equivalent to 86.3MB/sec) and Hadoop takes 178.0sec (71.9MB/sec) to transfer, and hence SSS is 17% faster than Hadoop. In our environment, the disk of master node is actually provided from an iSCSI disk array connected via Gigabit Ethernet, and its read/write bandwidth must be bounded to approx. 1Gbps. Therefore this result is not so bad compared to the baseline performance. However, both transferring methods are not well-optimized and performed sequentially and further optimization is left for future work.

Figure 7 shows the execution times of Hadoop, SSS, and packed-SSS (sequentially 5 times executed). As you see, SSS is almost equivalent to or faster than Hadoop, and packed-SSS is almost 3.3 times faster than Hadoop.

In fact, this benchmark is quite disadvantageous to SSS, because the number of the intermediate key-value data is extremely large. Benchmark data includes 1,586,167,232 words (uniquely 2,687,884 words included), therefore SSS creates almost 1.5G intermediate key-value data during running map tasks. To reduce the number of intermediate key-value data, we can utilize a combiner for combining multiple occurrences of each word. But, we need to avoid using a combiner in order to compare our system with Hadoop in *equal* conditions, e.g., the numbers of running combiners should be same, the numbers of combined key-value data should be equivalent, and so on.

Other than combiners, we made a lot of efforts to employ dynamic optimization techniques, such as storing multiple key-value data in a batch, but unfortunately this was the ceiling performance at the time this camera-ready version was written.

On the other hand, the result of packed-SSS seems to be very hopeful. For every file, packed-SSS's map task groups key-value data by the destination KVS and converts each grouped key-value data into a single key-value data. Therefore, only 16 key-value data are created for each target file. That is, packed-SSS creates only 2,048 key-value data during running map tasks. However, packed-SSS requires lots of memory to pack the intermediate key-value data in map tasks, and to extract and merge the intermediate key-value data in reduce tasks.

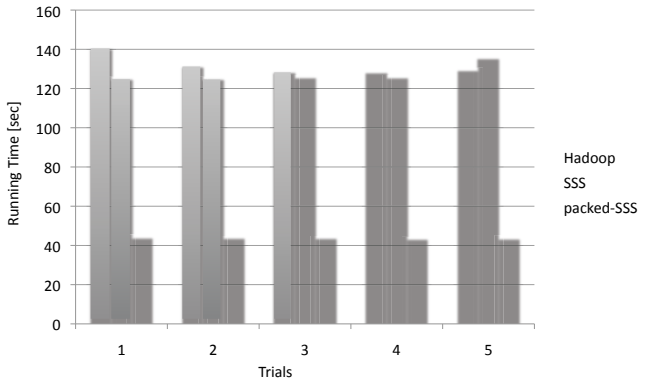

Fig. 7. Running Times of Word Count

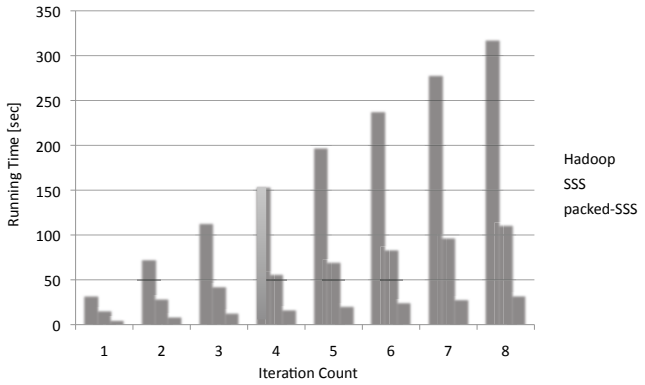

Fig. 8. Running Times of Iterative Identity Map and Reduce

B. Iterative Identity Map and Reduce

This benchmark applies identity map and identity reduce iteratively. Both of identity map and identity reduce generate a set of key-value data same as the input set of key-value. Therefore, the number and amount of key-value data are never changed after applying identity map and reduce repetitively. We prepared 524,288 key-value data for each worker node, i.e., 8,388,608 key-value data in total. Each data has 6 bytes key and 8 bytes value, and total size of all data is almost 128MiB, roughly 100 times smaller than that of Word Count.

Figure 8 show the execution times of Hadoop, SSS, and packed-SSS respectively. In this benchmark, SSS is 2.9 times faster than Hadoop, and packed-SSS is 10 times faster than Hadoop. Also, running times are linearly increased according to the iteration counts in all three implementations.

This result indicates that SSS has a big advantage in handling large number of very small key-value data over Hadoop. That means, SSS can outperform Hadoop especially for fined-grained applications.

VI. Related Work

We know of several related work in implementing iterative MapReduce efficiently, namely Twister[7], MRAP[20], Spark[8], and DryadLINQ[21], [22].

Twister[7] provides a distributed in-memory MapReduce runtime. In Twister, map tasks can read input data from both of the preloaded memory cache and the local disks of the worker nodes. Therefore, Twister allows users to select pre-loading relatively smaller static data or reading larger variable data from the local disks. Coupling very fast SSDs and memory-caching mechanism provided by key-value stores, we think, we can relieve users from trouble such as selecting access methods depending on the data property.

MRAP[20] extends MapReduce to eliminate the multiple scans and reduce the cost of pre-processing input data from the distributed file system. Their motivation is similar to ours, as we described in Section I and II. In MRAP, map tasks can read input data from both of the original files and the preprocessed files optimized to efficient access. In our implementation, all data are preprocessed to a set of key-value data for efficient data access, and they are stored in the distributed key-value store beforehand. However, we need to investigate that key-value format is suitable for HPC analytic applications which MRAP targets to.

Spark[8] redesigns MapReduce framework by using simple abstractions, such as resilient distributed datasets (RDDs), broadcast variables, and accumulators. In Spark, any distributed datasets are represented by RDDs. And users can perform global operations on RDDs and reuse RDDs in iterative applications.

Dryad and DryadLINQ[21], [22] offer a programming model for distributed computing, which enables to execute data-intensive applications including SQL and MapReduce in a data-parallel manner. DryadLINQ provides various static and dynamic optimization techniques e.g., pipelining and I/O reduction, but it still employs file-based communication mechanism to transfer data and cannot avoid incurring large overhead.

VII. Conclusion

MapReduce has been very successful in implementing large-scale data-intensive applications. Because of its simple programming model, MapReduce has also begun being utilized as a programming tool for more general distributed and parallel applications, including HPC applications. However, its applicability is limited due to relatively inefficient runtime performance and hence insufficient support for flexible workflow. In particular, the performance problem is not negligible in iterative MapReduce applications. On the other hand, today, HPC community is going to be able to utilize very fast and energy-efficient Solid State Drives (SSDs) with 10 Gbit/sec-class read/write performance. This fact leads us to the possibility to develop “High-Performance MapReduce”, so called. From this perspective, we have been developing a new MapReduce framework called “SSS” based on distributed key-value store.

In this paper, we discussed the limitations of existing MapReduce implementations and presented the design and detailed implementation of SSS. In SSS, we completely substitute distributed KVS for the distributed file systems, e.g.,

GFS and HDFS. Furthermore, we utilize distributed KVS for storing the intermediate key-value data, as well as the inputs of map tasks and the outputs of reduce tasks. Although our implementation of SSS is still in a prototype stage, we performed an evaluation using two benchmarks in order to compare the runtime performance of SSS, packed-SSS, and Hadoop. The results indicate that SSS and packed-SSS performs 1-10 times faster than Hadoop. We know that this evaluation is very limited, but it (at least partially) proves the effectiveness of our KVS-based design and implementation of MapReduce framework.

In future work, we will focus on following:

- Enhance the performance and completeness of our prototype system. Packed-SSS is enough faster than Hadoop, but it also requires much more memory than SSS. Therefore, we need to improve both of the performance and memory usage of SSS by our continuous effort.
- Provide fault-tolerance to our prototype system. For the underlying storage systems, HDFS and cluster file systems provide fault-tolerance to block losses with node failure, while SSS doesn't provide such features at this moment. We will introduce a replication mechanism to avoid such problems.
  For the runtime, in Hadoop, when one of mappers and reducers fails, it can re-execute independently of other workers. However, in our system, when one of map tasks fails, the intermediate key-value data in distributed KVS will be inconsistent. We will support rollback and re-execution mechanisms based on *session-id*.
- Perform more comprehensive benchmarks, in order to identify the characteristics of SSS and its feasibility to various classes of HPC and data-intensive applications. Especially, both benchmarks we conducted in Section V are not real-life iterative and/or multiple-step applications. To prove the effectiveness of our approach, we have to conduct more extensive benchmarks by employing realistic applications, such as iterative K-means clustering and iterative PageRank computation.
- Provide a higher-level programming tool, e.g., Sawzall[23], DryadLINQ[21], and R[24], in order to allow users to facilitate our framework for broader range of distributed data-intensive applications.

ACKNOWLEDGMENT

We would like to thank the members of the SSS project for their very substantial contributions to this work.

This work was partly funded by the New Energy and Industrial Technology Development Organization (NEDO) Green-IT project.

REFERENCES

[1] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[2] Apache Hadoop Project, "Hadoop," http://hadoop.apache.org/.

[3] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly, "Dryad: distributed data-parallel programs from sequential building blocks," in *EuroSys '07: Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*. New York, NY, USA: ACM, 2007, pp. 59–72.

[4] Y. Gu and R. L. Grossman, "Sector and Sphere: The Design and Implementation of a High Performance Data Cloud," *Philosophical Transactions: A Special Issue associated with the UK e-Science All Hands Meeting*, vol. 367, pp. 2429–2445, 2009.

[5] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The Google File System," in *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*. New York, NY, USA: ACM, 2003, pp. 29–43.

[6] D. Borthakur, "HDFS Architecture," http://hadoop.apache.org/hdfs/docs/current/hdfs_design.html.

[7] J. Ekanayake, H. Li, B. Zhang, T. Gunarathne, S.-H. Bae, J. Qiu, and G. Fox, "Twister: A Runtime for Iterative MapReduce," in *Proceedings of The First International Workshop on MapReduce and its Applications (MAPREDUCE'10)*, 2010.

[8] M. Zaharia, M. Chowdhury, M. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster Computing with Working Sets," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2010-53, May 2010. [Online]. Available: http://www.eecs.berkeley.edu/Pubs/TechRpts/2010/EECS-2010-53.html

[9] Fusion-io, "Fusion-io :: Products," http://www.fusionio.com/Products.aspx.

[10] FAL Labs, "Tokyo Tyrant: network interface of Tokyo Cabinet," http://fallabs.com/tokyotyrant/.

[11] D. Karger, E. Lehman, T. Leighton, R. Panigrahy, M. Levine, and D. Lewin, "Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the World Wide Web," in *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*, ser. STOC '97. New York, NY, USA: ACM, 1997, pp. 654–663. [Online]. Available: http://doi.acm.org/10.1145/258533.258660

[12] FAL Labs, "Tokyo Cabinet: a modern implementation of DBM," http://fallabs.com/tokyocabinet/index.html.

[13] S. Chu, "MemcacheDB," http://memcachedb.org/.

[14] Oracle Corporation, "Oracle Berkeley DB," http://www.oracle.com/technetwork/database/berkeleydb/overview/index.html.

[15] J. Garzik, "Hail Cloud Computing Wiki," http://hail.wiki.kernel.org/index.php/Main_Page.

[16] "Java Binding of Tokyo Tyrant," http://code.google.com/p/jtokyotyrant.

[17] couchdb-fuse: A FUSE interface to CouchDB, http://code.google.com/p/couchdb-fuse/.

[18] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: Amazon's Highly Available Key-Value Store," in *SOSP '07: Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*. New York, NY, USA: ACM, 2007, pp. 205–220.

[19] A. Lakshman and P. Malik, "Cassandra: A Decentralized Structured Storage System," *SIGOPS Operating Systems Review*, vol. 44, no. 2, pp. 35–40, 2010.

[20] S. Sehrish, G. Mackey, J. Wang, and J. Bent, "MRAP: A Novel MapReduce-based Framework to Support HPC Analytics Applications with Access Patterns," in *Proceedings of High Performance Distributed Computing (HPDC 2010)*, 2010.

[21] Y. Yu, M. Isard, D. Fetterly, M. Budiu, U. Erlingsson, P. K. Gunda, and J. Currey, "DryadLINQ: A System for General-Purpose Distributed Data-Parallel Computing Using a High-Level Language," in *OSDI'08: Proceedings of the 8th USENIX conference on Operating Systems Design and Implementation*. Berkeley, CA, USA: USENIX Association, 2008, pp. 1–14.

[22] M. Isard and Y. Yu, "Distributed Data-Parallel Computing Using a High-Level Programming Language," in *SIGMOD '09: Proceedings of the 35th SIGMOD International Conference on Management of Data*. New York, NY, USA: ACM, 2009, pp. 987–994.

[23] R. Pike, S. Dorward, R. Griesemer, and S. Quinlan, "Interpreting the data: Parallel analysis with Sawzall," *Scientific Programming*, vol. 13, no. 4, pp. 277–298, 2005.

[24] The R Project for Statistical Computing, http://www.r-project.org/.