Accelerating Data Transfers In Iterative MapReduce Framework

Bingjing Zhang
Department of Computer Science
Indiana University Bloomington
zhangbj@indiana.edu

Judy Qiu
Department of Computer Science
Indiana University Bloomington
xqiu@indiana.edu

***Abstract*—MapReduce has become popular in recent years due to its attractive programming interface with scalability and reliability in processing big data problems. Recently several iterative MapReduce frameworks including our Twister system have emerged to improve the performance on many important data mining applications. Utilizing local memory on each compute node to cache invariant data, we are able to accelerate MapReduce execution but we still find performance issues when transferring massive data between or during iterations. Taking K-means Clustering as an example, the centroids are required to be broadcasted to all the Map tasks each iteration and every local new centroid generated by each Map task must be transferred in the shuffling stage. Here we research several new methods to improve the performance of broadcasting and shuffling in iterative MapReduce. We introduce a new Multi-Chain broadcast method that reduces the broadcasting time by 60% when broadcasting 1 GB data to 125 nodes. Further we show a new local reduction method can reduce shuffling time to about 10% of the original time, where a K-means Clustering application runs 1000 Map tasks with 1 GB data output per Map task.**

KeyWords-Iterative MapReduce; Data-intensive applications; Data transfer; Broadcasting; Shuffling; Fat-Tree topology

I. INTRODUCTION

The rate of data generation has now exceeded the growth of computational power predicted by Moore's law. Computational challenges are related to mining and analysis of these massive data sources for the translation of large-scale data into knowledge-based innovation. However, many existing analysis tools are not capable of handling such big data sets. MapReduce frameworks have become popular in recent years for their scalability and fault tolerance in large data processing and simplicity in programming interface. Hadoop [1], an open source implementation following original Google's MapReduce [2] concept, has been widely used in industry and academia.

Intel's RMS (recognition, mining and synthesis) taxonomy [3] identifies iterative solvers and basic matrix primitives as the common computing kernels for computer vision, rendering, physical simulation, (financial) analysis and data mining applications. These observations suggest that iterative MapReduce will be a runtime important to a spectrum of eScience or eResearch applications as the kernel framework for large scale data processing.

However, classic Map and Reduce task pair cannot meet the requirement of executing iterative algorithms as inefficiency of repetitive disk access for fetching and merging data over iterations. Several new frameworks designed for iterative MapReduce are proposed to solve this problem, including Twister [4] and HaLoop [5].

Twister, developed by our group, is an iterative MapReduce framework. It categorizes data to be static and variable. Static data is loaded from disk and cached into memory in the configuration stage before the MapReduce job execution. Worker daemons are under the control of driver node and execute MapReduce jobs by spawning Map and Reduce task threads. The early version of Twister iterative MapReduce is targeted for optimizing data flow and reducing data transfer between iterations by caching invariant data in local memory or disk of compute nodes. The scheduling mechanism assigns tasks to the node where relevant invariant data is located.

In this paper, we propose three Map-collective methods: Multi-Chain, Scatter-Allgather-BKT (bucket) and Scatter-Allgather-MST (minimum spanning tree) to accelerate data transfers in Twister. They improve the performance of two key data transfer operations: broadcasting and shuffling. For broadcasting, we develop parallel methods based on the traditional deterministic broadcasting algorithms in MPI [7]. Multi-Chain creates multiple chains to transfer data in a pipeline style [8]. Scatter-Allgather-BKT and Scatter-Allgather-MST both follow the style of "divide, distribute and gather" [9]. The difference is that the former sets a barrier between Scatter and Allgather stage and uses bucket algorithm [10] for Allgather while the latter broadcasts each data piece scatted in a minimum spanning tree (MST) without waiting for a barrier. These Map-collective methods provide important capabilities of our new iterative MapReduce framework for date intensive applications. We show improved performance of transferring 8 Terabytes of intermediate data with 1000 Map tasks on 125 nodes enabling large-scale image classification.

A network topology commonly used in HPC or cloud environment is a Fat-Tree topology [12] where the cost of inter-switch connection is high. Instead of applying Multi-Chain, MST and BKT algorithm simplistically, we add the topology information into the algorithms.

Although Twister has already achieved better performance than Hadoop by leveraging in memory shuffling, the cost of shuffling is still high for applications like K-means Clustering [6] with intensive intermediate data transfer. We propose an extra step of local reduction before

shuffling to reduce data size by utilizing shared memory as Map tasks and Reduce tasks run on threads. Another optimization is to serialize data objects in memory efficiently. Instead of serializing data objects to byte arrays and then merge them into a byte stream at each iteration step, we provide new messaging interfaces and mechanisms to minimize the cost of data serialization. We also overlap data serialization and communication to reduce data broadcasting time.

We evaluate our new methods using a real application of image identification on 125 compute nodes of the PolarGrid [13] cluster at IU. K-means Clustering produces 1 GB data output per Map task with a total of 1000 Map tasks. We show that the new Multi-Chain method reduces the broadcasting time to 44% of the original broker based method and to 54% of standard MPI Scatter-Allgather-BKT algorithm. Shuffling operation with local reduction is about 10% of the original time and the total cost per iteration is reduced 50%.

The rest of paper is organized as follows. Section 2 discusses the basic Twister MapReduce framework and K-means Clustering algorithm. Section 3 presents the design of broadcasting algorithm. Section 4 presents how the new shuffling mechanism works. Section 5 shows the experiments and results. Section 6 gives related work And Section 7 the conclusion and discussion about future work.

II. BACKGROUNDS

Twister is a framework which can accelerate iterative MapReduce job execution by caching the invariant data into the local memory of the compute nodes. In this section, we provide an overview of Twister and discuss about the current problems. We also give the background knowledge of K-means Clustering algorithm with related applications and show how it works in Twister. Finally we give a brief discussion of the Fat-Tree topology in IU PolarGrid.

A. *Twister Iterative MapReduce Framework*

Twister has several components, a single driver to drive MapReduce jobs, and daemon nodes to handle requests from the driver and execute iterative MapReduce jobs (See Fig. 1). These components are connected through messaging brokers via a publish/subscribe mechanism. Currently Twister supports two different kinds of brokers. One is ActiveMQ [13], another is NaradaBrokering [14].

The Twister driver program allows users to configure an iterative MapReduce job with static data cached in Map tasks or Reduce tasks before the start of job execution and to drive the job iteratively with loop control. It can also send the variable data to work nodes at the beginning iteration and collected back at the end of the iteration. In this model, fault tolerance is provided through setting checkpoints between iterations.

On daemon nodes, during the configuration stage, Map and Reduce workers are created and static data is loaded from the local disk and cached into memory. Later in the execution stage, daemons start to execute MapReduce tasks with threads. Twister uses static scheduling for workers in order to leveraging the local data cache.

In current released version, there is no support for a distributed file system. Files and replicas are stored on local disks of each compute node and recorded in a partition file. The Max Flow algorithm [15] is used to decide the mapping between workers and the files in job execution. Since our current solution lacks reliability and scalability, we are moving toward using distributed file systems such as HDFS [16].

Early Twister work used messaging brokers to do data transfers and the first Twister prototypes use only one broker, which is sufficient for small messages transfers. However for large data transfers, it becomes a hot spot and stops Twister from scaling. Then we switched to use a network of brokers, but we still find the following issues:

- Unnecessary communication hops are added in data transfers. It is especially bad for big messages which usually need significant time to transfer from one point to another point.
- The Broker network doesn't provide optimal route to transfer data between a set of brokers and nodes. Every broker gets the message and forwards it directly.
- Reliability issues are found in maintaining brokers. Brokers are not always reliable in message transmission. Messages can get lost without notification and the broker could also fail. Then the potential failure points in the system increase as the number of brokers increase. This brings additional work to manage distributed brokers.

For these reasons, in current released version of Twister, we already transfer intermediate data in shuffling through TCP sockets. Though some broker based methods can improve broadcasting, they are far from the optimal, and so we no longer use brokers.

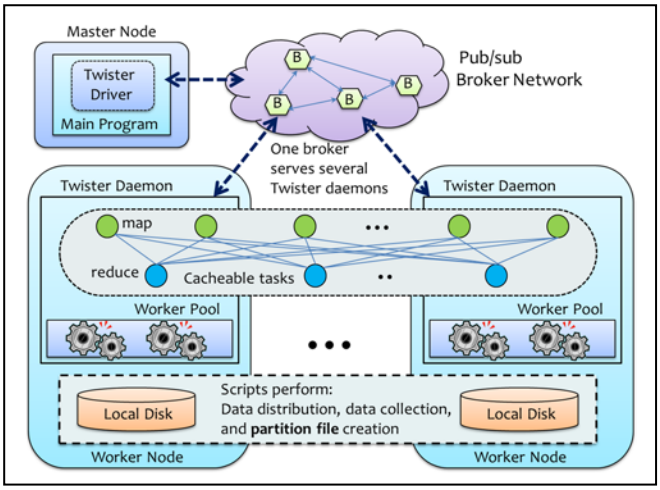

Figure 1. Twister achitecture

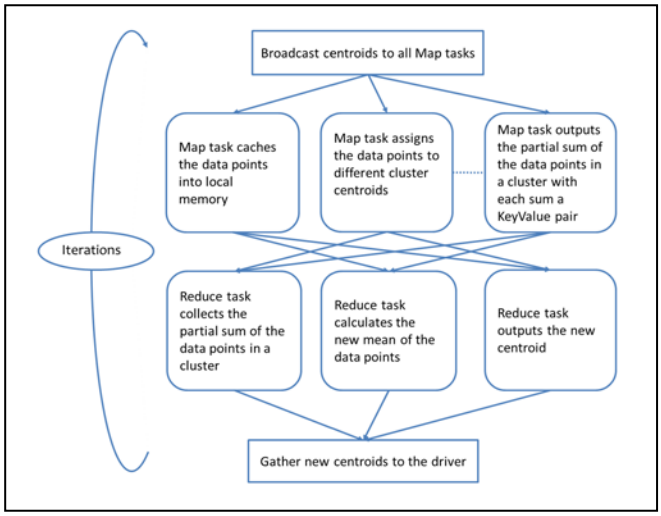

Figure 2. K-means Clustering execution flow in Twister

B. *K-means Clustering and Image Clustering Application*

K-means Clustering is an iterative algorithm to partition data points into a given number of clusters. At the beginning, a set of centroid points are randomly generated or randomly picked from the data points. To assign a point to a cluster, the algorithm lets each data point go through the whole list of centroid points to find the closest centroid by calculating the Euclidean distances. Once all the data points are assigned, the new centroids of clusters are recalculated by calculating the mean of all the coordination values of the points in each cluster. After several iterations, the positions of centroids reach a local optimization.

We observe that the positions of data points are static over iterations. So for Twister K-means Clustering, the data points are partitioned and each cached to a Map task. We broadcast centroid data to all the Map tasks. Then we let each Map task assign the data points it owns to the clusters and output the partial sum of coordination values of data points in each cluster. We use one reducer or multiple reducers to collect the partial sum of data points assigned to each cluster from the Map tasks and calculate the mean after getting the total sum. By combining these new centroids from Reduce tasks and the driver gets the update and goes to the next iteration (See Fig. 2).

In a real application, we use K-means Clustering algorithm to cluster images. In this application, each image is presented as a vector with 500 feature values so that it can be treated as a point with 500 dimensions. We use K-means clustering algorithm to partition images to clusters with each of which contains images with similar features.

However, there is difficulty to scale this application. As each point has high dimensions, the total size of centroids can be very large and go to MB or GB level. Because the time required for broadcasting and shuffling is proportional to the number of compute nodes and the data size of centroids, the cost of broadcasting and shuffling is extremely high.

C. *IU PolarGrid*

IU PolarGrid uses Fat-Tree topology to connect nodes. The nodes are split into sections of 42 nodes which are then tied together with 10 GigE into a Cisco Nexus core switch. For each section, nodes are connected with 1 GigE to an IBM System Networking Rack Switch G8000. So it is a 2-level Fat-Tree structure with first level 10 GigE connection and second level 1 GigE connection (See Fig. 3).

This kind of topology can easily cause contention when there are many inter-switch communication pairs. It is not only because inter-switch communication has more delay than intra-switch communication, but also because a 10 GigE connection limits the number of parallel communication pairs across switches. Assuming that every 1 GigE link to each node is fully utilized, a 10 GigE connection can only support 10 parallel communication pairs across rack switches in maximum. Otherwise the inter-switch communication pairs could affect each other in performance. As a result, reducing the times of inter-switch communication is the first thing to be considered in the design of efficient collective communication algorithms on this fat-tree topology,

For computing capacity, each compute node in PolarGrid uses a 4-core 8-thread Intel Xeon CPU E5410 2.33 GHz processor. The L2 cache size per core is 12 MB. The total memory per node is 16 GB.

III. BROADCASTING TRANSFERS

To solve the performance problems of broadcasting, we investigated several approaches. Initially we tried to use multiple brokers to replace original single broker only solution to improve broadcasting speed. However those methods still have performance issue because message routes between brokers or between brokers and clients are far from optimal. Thus we moved to other methods which uses TCP sockets directly. The broker systems can only achieve good performance if they are optimized for the structured communication patterns we need; this could be an interesting research area.

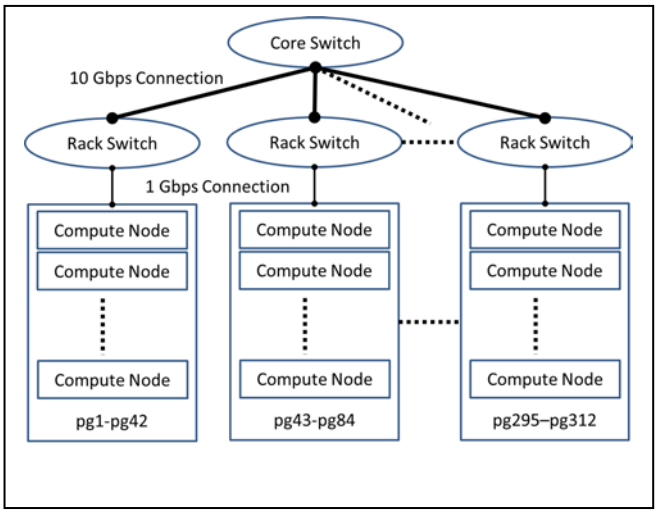

Figure 3. Fat-Tree Topology in IU PolarGrid

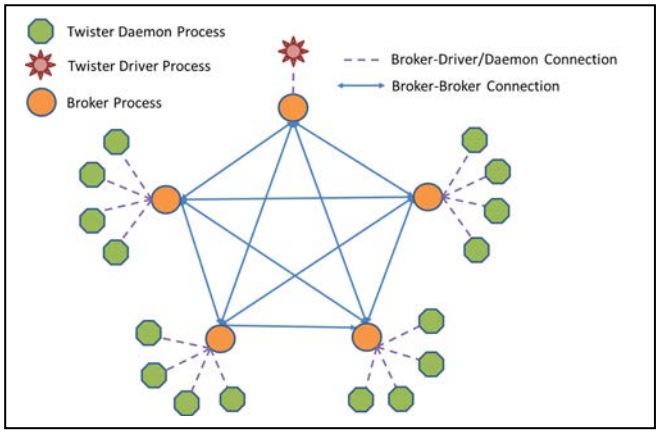

Figure 4. Full mesh broker network topology with 5 brokers and 16 daemon nodes

With utilizing bucket algorithm and minimum-spanning-tree algorithm, we develop two methods: Scatter-AllGather-BKT and Scatter-AllGather-MST. Then to fully utilize the bandwidth per link, we develop Multi-Chain method based on pipelined broadcasting. We also embed topology awareness in algorithm design and make the time used on message serialization overlap with the time used on broadcasting. To illustrate the performance model, we use p as the number of daemon nodes (each node is controlled by one daemon process), b as the number of brokers, k as the number of data chunks, n as the data size, α as communication startup time and β as data transfer time per unit.

A. *Broker-Based Methods*

Two methods are used to remedy the one broker only method. One is a full mesh broker network for tree based broadcasting and another is a set of brokers for broadcasting in "scatter and allgather".

In full mesh broker network, every broker connects with the rest of brokers (See Fig. 4). Each broker serves several Twister daemons which are evenly distributed except one broker serves Twister driver exclusively. By this way, we can maintain the reachability of connections between every two Twister components and do broadcasting in a two level tree structure. Once the exclusive broker gets the data sent from the driver, it forwards the data to each of the rest of brokers. Then these brokers continue forwarding the data to the clients it connects to. The performance improvement is gained from the second level where each broker can do data forwarding in parallel. Network contention can be avoided if the nodes served by one broker are in the same switch. The performance model can be established as follow:

$$T_{FullMeshNetwork}(p,b,n) \approx (b + p/b)(\alpha + n\beta) \quad (1)$$

When $\partial T/\partial b = 0$, we can get $b_{opt} \approx \sqrt{p}$. Though it is much better than naïve broadcasting in performance, it is still very slow. Considering 1GB broadcasting on 100 nodes using 1Gbps links, this method needs 10 brokers and use about 200 seconds to finish! Besides we found that messages could be lost in this algorithm..

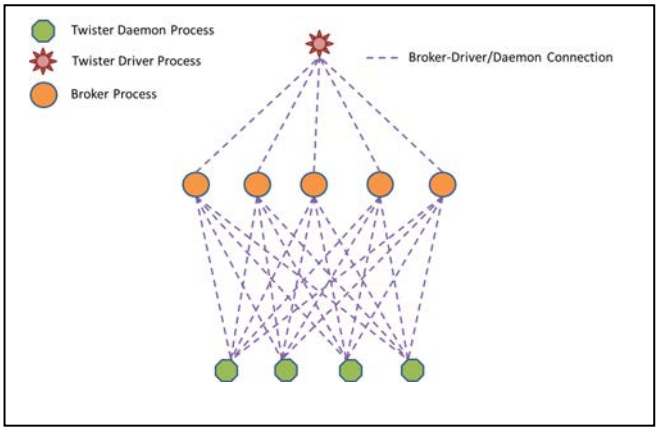

Figure 5. Unconnected broker array with 5 brokers and 4 daemon nodes

Another broker topology is an unconnected broker array. In this method, each broker connects to all Twister components and each client also connects to all the brokers and tries to balance the workload of data sending among broker connections (See Fig. 5). Broadcasting is done in a style of "scatter and allgather". In Scatter phase, data is split and sent to brokers first. Then in Allgather phase, each broker broadcasts the data chunk it owns to all the client processes. Ideally, assuming there is no contention, we can derive:

$$T_{Scatter-Allgather-Broker}(p,b,n) \approx (b + p)(\alpha + n\beta/b) = (b + p)\alpha + n\beta + pn\beta/b \quad (2)$$

Since α can be ignored in large data transferring, we conclude that $b_{opt} \approx p$. However, this is not verified in experiments. The problem is that we cannot control the routes of messages in broker based communication and congestion can happen on some links. As a result, Allgather overlaps Scatter and they affect each other and the performance degrades and is variable. There is also a reliability issue in deploying this kind of topology in large scale. Because the number of connections each node can support is limited, the number of connections on each node cannot grow as p grows. As p goes to a large number, failures can happen on some broker connections. Despite of seeing these shortcomings, we still see the method achieves better performance than the full mesh broker network.

B. *Scatter-Allgather Methods*

Well known successes with MPI collective communication algorithms makes us turn to use TCP socket connection directly in order to control the message routes in broadcasting ourselves. As Scatter-Allgather-Broker method, the following two methods are also called Scatter-Allgather methods because they all follow the principle of "divide, distribute and gather". The difference is that in Allgather phase, one uses bucket algorithm but the other uses minimum-spanning-tree algorithm. We call them Scatter-Allgather-BKT and Scatter-Allgather-MST separately.

Scatter-Allgather-BKT algorithm is an algorithm used in MPI for long vectors broadcasting. It first scatters the data to all the nodes. To do this, it can use MST algorithm or a

straightforward algorithm. Then in Allgather phase, it views the nodes as a chain. At each step, each node sends data to its right neighbor. By taking advantage of the fact that messages traversing a link in opposite direction do not conflict, we can do Allgather in parallel without any network contention (See Fig. 6). The performance model can be established as follow:

$$T_{Scatter-Allgather-BKT}(p,n) \approx p(\alpha + n\beta/p) + (p-1)(\alpha + n\beta/p) = (2p-1)(\alpha + n\beta/p) \quad (3)$$

Since we can control every step in the algorithm, we set a barrier between Scatter and Allgather to prevent them from affecting each other. We also make the communication in Allgather be topology-aware, i.e. nodes in the same rack are close to each other in the chain and each data only travel on any inter-switch link once. This makes its performance much better than broker based methods. But in experiments, we see the performance is still slightly slower than the theoretical value. Since it is impossible to enable all the nodes to do Allgather at the same global time through sending control messages from the driver, some links have more load than the others and thus it causes network contention.

An alternative method is Scatter-Allgather-MST which uses MST algorithm in Allgather phase Since MST is good at broadcasting small messages [10], we scatter small data chunks to nodes and let each node broadcast the data chunk in a MST (See Fig. 7).

To reduce the conflict, each node builds its own MST with itself as root. In each MST, each node in the tree is assigned with a rank for calculating the sending topology in the tree. The root node is always assigned with rank 0. Assuming the real ID of the root node is i $(0 \le i \le p-1)$, then rank j in the MST is the node with $(i+j)\ mod\ p$. So every node has a different rank in each tree. Due to the difficulty of mapping a set of MSTs to the physical topology, the contention still exists in this algorithm, but the total workload on each node in data sending is balanced and the link contention between trees is also reduced by rearranging ranks for each tree.

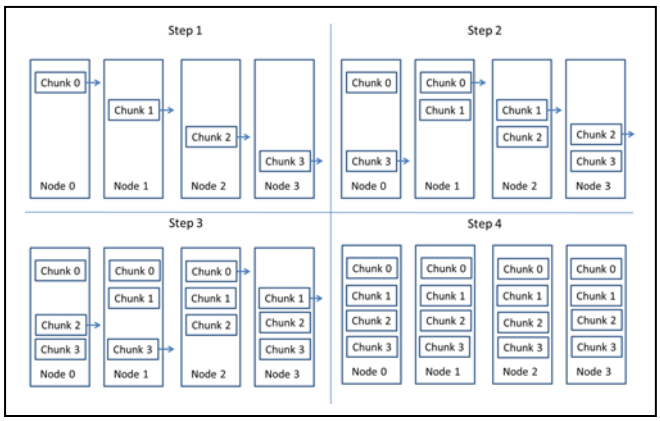

Figure 6. Bucket algorithm in Allgather of Scatter-Allgather-BKT

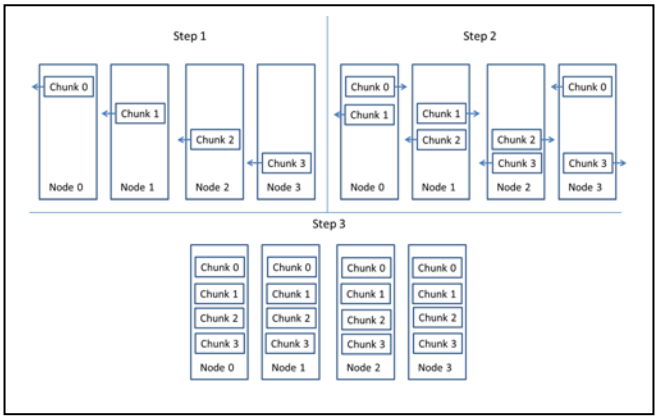

Figure 7. MST algorithm in Allgather of Scatter-Allgather-MST

C. *Multi-Chain*

Here we present the Multi-Chain method, an algorithm based on pipelined broadcasting. In this method, compute nodes in Fat-Tree topology are treated as a linear array and data is forwarded from one node to its neighbor chunk by chunk. The performance is gained by dividing the data into many small chunks and overlapping the transmission stages. For example, one would send the first chunk of the data to the next node. Then, while the second node sends the first piece to the third node, one would send the second piece to the second node, and so forth [8]. Furthermore, to better utilize the bandwidth and multi-core, we create multiple chains. For IU PolarGrid, we create 8 chains with each thread managing a chain.

Since in Fat-Tree topology each node only has two links, which is less than the number of links per node in Mesh/Torus [17] topology, chain broadcasting can maximize the utilization of the links per node. We also make the chain be topology-aware by putting nodes within the same rack close in the chain. If nodes are not evenly distributed among switches, assuming $N_{R1} > N_{R2} > N_{R3} > \cdots$, then we put the nodes in R_1 at the beginning of the chain, then nodes in R_2 at the second, and then nodes in R_3 ….

In the ideal case, if every transfer can be overlapped seamlessly, the theoretical performance is as follow:

$$T_{Pipeline}(p,k,n) = p(\alpha + n\beta/k) + (k-1)(\alpha + n\beta/k) \quad (4)$$

Because n is large in our transfers, when $\partial T_{1D}/\partial k = 0$, $k_{opt} = \sqrt{(p-1)n\beta/\alpha}$ [8]. However, the speed of data transfers on each link could not be always at the same speed in practice so that network congestion could happen at some time or place in the network when you keep forwarding the data into the pipeline. So we add barriers inside of the execution flow to coordinate the pipeline. The data is partitioned to chunks. Each chunk is broadcasted in a pipeline called “small pipeline” and the whole data is also broadcasted in a pipeline called “big pipeline”.

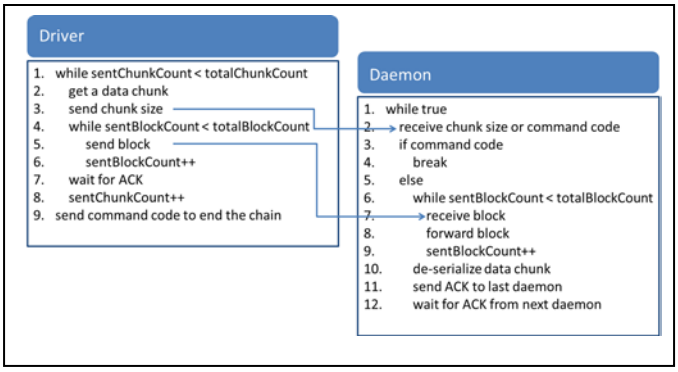

Figure 8. The algorithm steps per chain

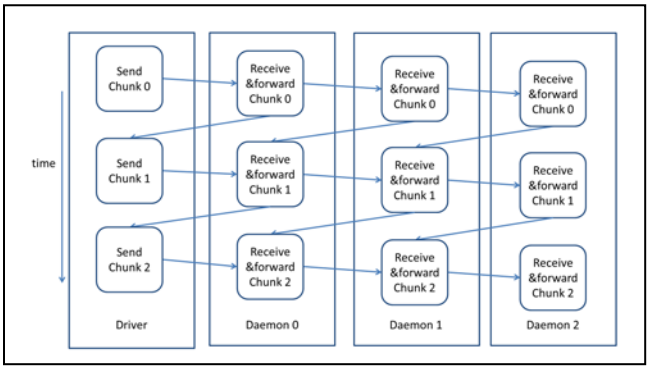

Figure 9. The execution flow per chain

In the small pipeline, every node receives partial data chunk called a block) and forwards it to the next node directly. No barrier is required to coordinate the behavior of this pipeline. Chunk 0 sent from Driver is pipelined in this way. But before sending Chunk 1, Driver needs to wait for an ACK sent from Daemon 0. For Daemon 0, once it finishes the action of receiving and forwarding Chunk 0, it sends an ACK to Driver to let it send Chunk 1 and wait for an ACK from Daemon 1 to see if it gets Chunk 0 just forwarded by Daemon 0. Once these two conditions are met, another small pipeline starts to transfer Chunk 1. This is the coordination in big pipeline. We present the algorithm steps of this flow in Fig. 8 and execution flow in Fig. 9. In the experiments, we set chunk size to 4 MB and block size to 8192 bytes as the optimal choices.

D. *Auxillary Steps in Broadcasting*

To start and end broadcasting, auxiliary processes are required, including topology learning, data serialization, serialization and transferring overlapping, and mechanisms to provide resiliency.

Currently automatic topology detection is not implemented. We put the topology information in a property file and let each node read it before starting broadcasting.

Broadcast data are abstracted and presented as Key-Value pairs in memory and they are serialized before sending and de-serialized after receiving. We find the serialization time of a single big object is extremely long so that so that we enable users to divide it into a set of small objects. Using small data objects also enable us to parallelize the serialization and let it be overlapped with broadcasting to improve the overall performance. In Multi-Chain, we use a producer/consumer model and in Scatter-Allgather methods, we let each thread serialize a data object and send it.

We also adapt several strategies to make the whole process fault tolerant. For failures in each node-to-node sending step, we do retry first otherwise jump to the other destinations. At the end of the broadcasting, the driver waits and checks if all the nodes have received all the data blocks. If driver doesn't get all the ACK within a time window, it restarts the process of broadcasting.

IV. SHUFFLING TRANSFERS

During the shuffling phase, the <Key, Value> pairs generated from Map tasks are regrouped by keys and processed as <Key, <Value>> by a Reduce task. In original MapReduce framework, this operation heavily uses the distributed file system and causes repetitive merges and disk access. As this could be very inefficient, in Twister, we leverage memory to do shuffling operation by directly transferring intermediate data through the network from the memory of the node where the Map task lives to the memory of the node where the Reduce task locates, so that the whole process is different from the one in original MapReduce.

But when the scale goes large, the performance degrades drastically. For example, in K-means Clustering, the data required to be transferred in shuffling is about apn bytes, a is the number Map task threads per node, p is the number of the node, and n is the data size of centroids generated by each Map task. So even the data of centroids is small, it can generate large intermediate data and cause the inefficiency of transferring large amount of data. So we try to reduce the intermediate data size to minimum by using local reduction across Map tasks.

To support local reduction, we provide related interface to help user to define the operation for local reduction. We also optimize the interface for serialization to reduce its cost.

A. *Memory-leveraging Shuffling in Twister*

Instead of disk-based repetitive merge in MapReduce frameworks like Hadoop, the current Twister does shuffling in memory. Due to the poor reliability and scalability of the brokers, we turn to use direct TCP transfers instead of relying on brokers to send intermediate data.

In Twister, each Map task is located in a daemon process and executed by a thread. Once a Key-Value pair is output from a Map task, it is hashed according to the key and regrouped according to the destination, i.e., the location of the Reduce task which is selected to process this key. The reduce task selection can be redefined by the user but the default implementation is based on the key's hash code and modulo operation. When a Map task finishes, it sends out all the Key-Value pairs it collects. There are two different kinds of routes. If the data size is small, e.g. less than 1MB, they are sent through the broker network. Otherwise, a small

control message which contains the metadata information of the real data is sent through brokers to the daemon process where the Reduce task resides. Then it processes the message and fetches the real data by using direct TCP transfers.

Since the intermediate data is large in shuffling, the program enters the second route in most cases. A thread pool is used at the receiver side to schedule the data retrieving activities to prevent it from crashing in heavy workload. The data received from the remote daemons are de-serialized and regrouped in a hash map based on the key. Once the data of a key from all the Map tasks are available, the daemon process starts the Reduce without delay. So the shuffling and reduce stages are coupled together and executed in a pipeline style.

B. Local Reduction

The mechanism currently used in Twister is efficient compared with original disk-based shuffling mechanism. However, the essence of the problem is that the data transferred in the shuffling stage is really large and the number of links is limited. Some solutions try to use Weighted Shuffle Scheduling (WSS) [18] to balance the data transfers by making the number of transferring flow to be proportional to the data size. But for K-means Clustering, this is not helpful because the data size generated per Map task is same and then there is no space for optimization.

By observing the computation flow of K-means Clustering, we find that each Key-Value pair in intermediate data is a partial sum of the coordination values of data points in a cluster. Since addition is an operation with both commutative and associative properties, for any two values belonging to the same key, we can operate them and merge them to a single Key-Value pair. This doesn't change the final result. This property can be illustrated by the following formula:

$$f(kv_1, \cdots, kv_i, \cdots, kv_j, \cdots, kv_n) = f(kv_1, \cdots, (kv_i \oplus kv_j), \cdots, kv_n) = f(kv_1, \cdots, (kv_j \oplus kv_i), \cdots, kv_n) \; \forall \, i, j, 1 \leq i, j \leq n \quad (5)$$

Here $\oplus$ is the operation defined on any two KeyValue pairs, f is the Reduce function and n is the number of Key-Value pairs belonging to the same key. In K-means Clustering, it is the addition of two partial sums of coordination values of data points. In other applications, we can also find this property. In Word Count [2], the intermediate data is the partial count of a word. We can merge two Key-Value pairs together to a single Key-Value pair with the value as the sum of two count values. And $\oplus$ can be operations other than addition, such as multiplication and max/min value selection, or just simple combination of the two values.

With this property and the fact that Map tasks works as threads in Twister daemon processes, we do local reduction in the memory of daemon processes shared by Map tasks. Once a Map task is finished, it doesn't send data out immediately but caches the data to a shared memory pool. When the key conflict happens, the program invokes user defined operation to merge two Key-Value pairs into one. A barrier is set so that the data in the pools are not transferred until all the Map tasks are finished. By exchanging communication time with computation time, the data required to be transferred can be significantly reduced.

C. New Interface Design

To support shuffling and local reduction, we provide new interfaces to let user define the Key and Value objects. We abstract data presentation through general interfaces Key and Value extended from TwisterSerializable Java interface.

Originally we serialize each Key-Value pair into a byte array and merge them together. However, it is very inefficient in shuffling stage when a large number of Key-Value pairs are required to be serialized and merged into a single byte array because the byte streams have to be created repeatedly to serialize each Key-Value pair. Now the new interface is provided to delegate TwisterMessage object to do serialization. With TwisterMessage object, user can use its APIs to directly serialize multiple Key-Value pairs into a single byte stream managed by it.

```
public interface TwisterSerializable {

    public void
        fromTwisterMessage(TwisterMessage
        message) throws SerializationException;
    public void toTwisterMessage(TwisterMessage
        message) throws SerializationException;
}
```

Based on TwisterSerializable, the interfaces Key and Value are defined. In the interface Key, an API named isMergeableInShuffle is defined to check if the current Key-Value pair can be merged in shuffling. At the same time, an API mergeInShuffle is defined in Value. It can take a Value object and merge its contents to the current Value object.

```
public interface Key extends
TwisterSerializable {

   public boolean equals(Object key);
   public int hashCode();
   public boolean isMergeableInShuffle();
}

public interface Value extends
TwisterSerializable {

   public boolean mergeInShuffle(Value value);
}
```

V. EXPREIMENTS

We do experiments on IU PolarGrid to evaluate the performance of the new methods we propose and compare the pros and cons of them. We do micro-benchmarking on broadcasting and shuffling, and full application benchmarking on K-means Clustering. The results show that Multi-Chain and Scatter-Allgather-BKT are two good choices for broadcasting and Shuffling with local reduction can outperform the original shuffling significantly.

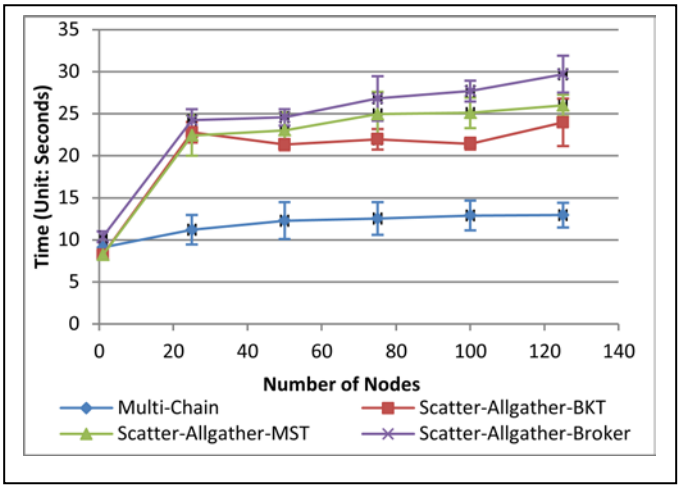

Figure 10. 1 GB data broadcasting

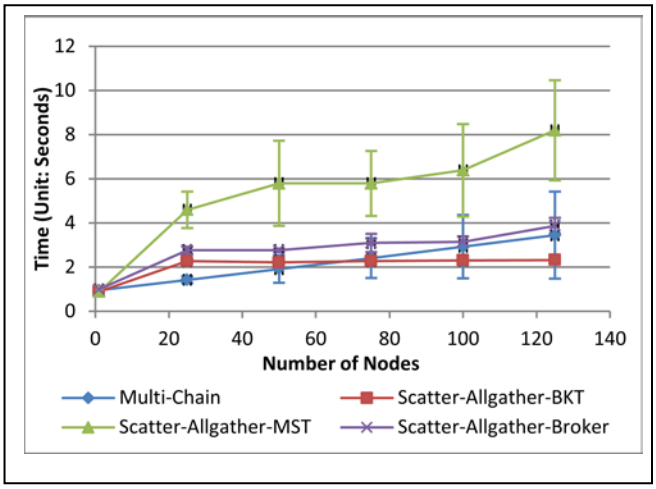

Figure 11. 100 MB data broadcasting

A. *Broadcasting*

Four broadcasting methods are tested in IU PolarGrid: Scatter-Allgather-BKT/MST/Broker, and Multi-Chain. The first three are all the methods following the principle of "divide, distribute and gather" and the final one is a pipeline based method described above.

The original one-broker and mesh-network solutions are not included in performance evaluation because they are not only extreme slow in theoretical performance but also easily cause failures in real experiment. The action of sending 1 GB data to a broker often fails. Due to the constraint of 1Gbps connection, one-broker method uses about 1250 seconds and mesh-network method uses 240 seconds on 125 nodes from a theoretical analysis.

For time evaluation, we measure the whole broadcasting process which starts from serializing message, then sending data, and ends with getting all the ACK messages from the receivers. We test the performance of broadcasting from a small scale to a moderate large scale. The range include 1 node, 25 nodes with 1 switch, 50 nodes under 2 switches, 75 nodes with 3 switches, 100 nodes with 4 switches, and 125 nodes with 5 switches. We also test on different data size, including 100 MB and 1GB. Each test is done 10 times. The performance results are given in Fig. 10 and Fig. 11.

We use different data chunking settings on different algorithms. For Multi-Chain, each chunk is about 4MB. That means for 1 GB data, there is about 250 chunks and for 100 MB data, there is about 25 chunks. For Scatter-Allgather-BKT/MST/Broker, we set the number of chunks equal to the number of nodes. But for 1 node test, because of the absence of Allgather stage, we can set the number of chunks comparable to the settings in Multi-Chain.

On 1 GB data broadcast, the performance results show that Multi-Chain has the best performance for all node counts. For Scatter-Allgather-BKT/MST, the former is not only better than the latter in performance but also more stable with lower deviation. For Scatter-Allgather-Broker, it only works well on small scale. When the scale goes large, the performance drops drastically because of the network contention caused by its route selection in Allgather stage. Multi-Chain outperforms it by a factor 2.3.

However, on 100 MB data broadcast, the chart tells a different story. The performance of Multi-Chain is variable at large node count. Though it can achieve the highest performance, its average performance is slower than Scatter-Allgather-BKT. The jitters probably come from the stragglers in the chain or the nondeterministic behaviors in the network. At the same time, Scatter-Allgather-MST is even worse than Scatter-Allgather-Broker in performance due to the contention on the receivers.

We also present Cumulative Distribution Function (CDF) chart of completion times of data chunks received in 1 GB data broadcast on 125 nodes (See Fig. 12) to give a closer look of the performance. The results show that Multi-Chain and Scatter-Allgather-BKT have higher data receiving rate than the rest two. We also see a large delay in MST method at receiving the final few data chunks. This is probably due to the contention on receivers for receiving messages from different MSTs.

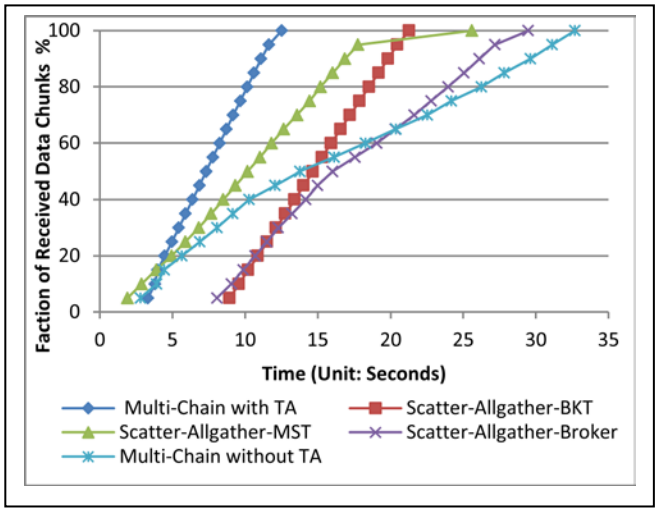

Figure 12. CDF of the received data chunks on 125 nodes under different broadcast methods (TA: Topology-Awareness)

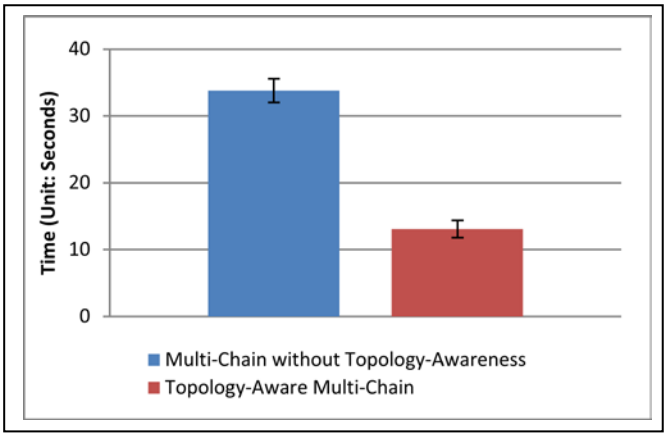

Figure 13. Broadccasting time comparison of Multi-Chain with or without Topology-Awareness

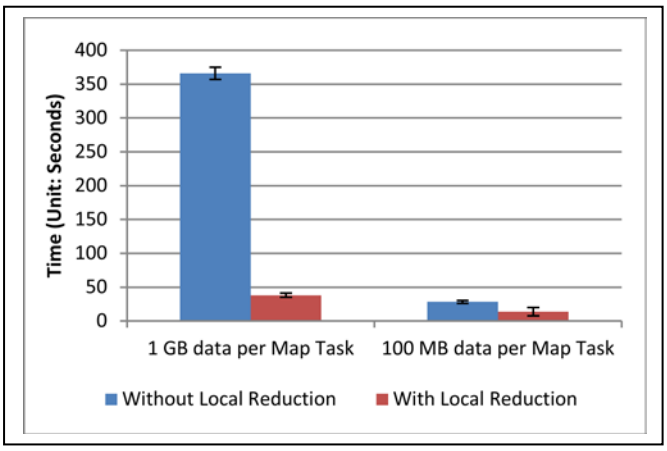

Figure 14. Shuffling time comparison with or without local reduction on 125 nodes with 1000 Map tasks

Besides, we show the importance of topology in our algorithm design by comparing the topology–aware Multi-Chain and the chain without topology-awareness on 125 nodes. By executing each method 30 times, we show that inter-switch transfers have significant impact compared with intra-switch transfers. The one with topology-awareness shows a factor of 2.8 performance increase shown in Fig. 13. Further, in Fig. 12, we also show that it has the lowest data chunk receiving rate.

B. Shuffling and K-means Clustering Application

We test shuffling with local reduction by using K-means Clustering. In order to show the clear difference of shuffling with local reduction and shuffling without local reduction, we focus on the case of 125 nodes.

To benchmark the performance of shuffling, we lower the number of data points each Map task processes to one point per task in order to shorten the total execution time. To simulate the image clustering application, we set each point to 500 dimensions. This application is called K-means Clustering benchmarking application. We measure the time from the start of shuffling to the end of Reduce. Time costs on Reduce tasks are included but can be ignored because they are very small compared with the data transfer time.

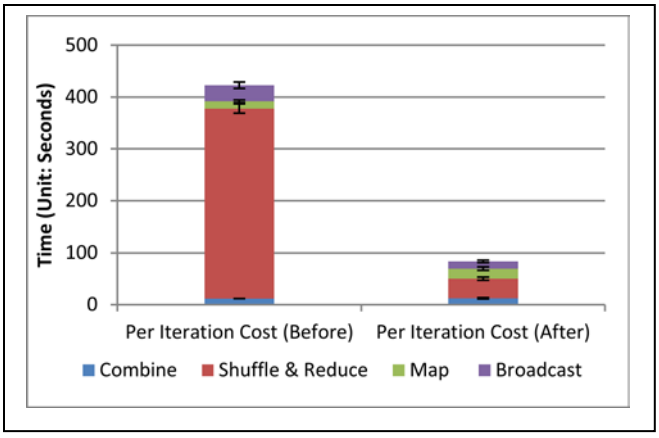

Figure 15. Comparison of time cost per iteration on 125 nodes with K-means clustering benchmarking application on 250K 500D centroids (1 GB data), and 1000 data points

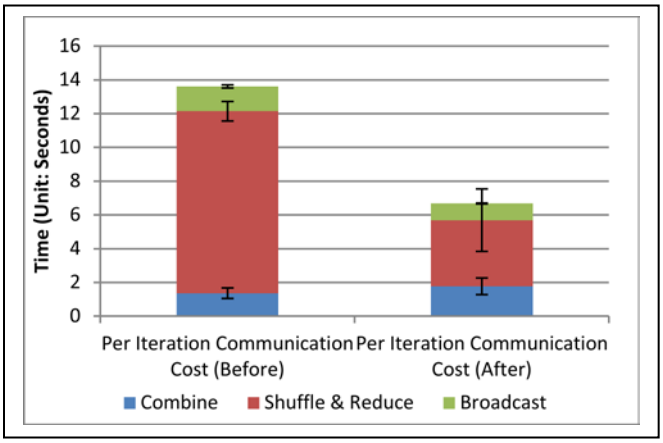

Figure 16. Comparison of communication time cost per iteration on 80 nodes with real image clustering application on 10K 500D centroids and 1 million data points

Fig. 14 shows the time difference on shuffling with or without local reduction on 125 nodes with 1000 Map tasks. We see that when the data size output per Map task is 1 GB, the total time used on shuffling with local reduction is one tenth of the original algorithm as the data is reduced to one tenth of the original. For 100 MB data per Map task, the time difference of shuffling is just a factor of two.

We also benchmark the total time cost per iteration by using the benchmarking application above to see how much the performance improvement is gained in by changing from the old data transfer methods to the new ones. We test with 1 GB centroids data. For the old methods we use broker based method for broadcasting and shuffling without local reduction. The new methods use Multi-Chain broadcasting and shuffling with local reduction. We find that the time cost per iteration is reduced to 20% of the original time (See Fig. 15).

We also test K-means Clustering with real image clustering data: 10K centroids and 1 million data points. We test it on a smaller scale with 80 nodes. The old methods still use Scatter-Allgather-Broker method for broadcasting and shuffling without local reduction. The new methods use

Scatter-Allgather-BKT for broadcasting because of its good performance on broadcasting small data and shuffling with local reduction. We show that the communication cost per iteration of the new method is half that of the old (See Fig. 16).

VI. Related Work

Collective communication algorithms are well studied in MPI runtime. The communication operations are divided into two parts, data redistribution operations including Broadcast, Scatter, Gather, Allgather, and data consolidation operations including Reduce, Reduce-scatter, Allreduce. Each operation has several different algorithms based on message size and network topology such as linear array, mesh and hypercube [10]. Basic algorithms are pipeline broadcast method [8], minimum-spanning tree method, bidirectional exchange algorithm, and bucket algorithm [10]. Since these algorithms have different advantages, algorithm combination is widely used to improve the communication performance [10]. And some solution also provides auto algorithm selection [19].

However, many solutions have a focus which is different from our work as they study small data transfers up to megabytes level [10][20]. The data type is typically vectors and arrays whereas we are considering objects. Many algorithms such as Allgather algorithms have the assumption that each node has the same amount of data [9][10] but this is not common in MapReduce computation model. As a result, though shuffling can be viewed as Reduce-scatter operation, its algorithm cannot be applied directly on shuffling because the data amount generated by each Map task is uneven in most MapReduce applications.

Furthermore, many past effort work on static topology such as linear array and mesh. But for Fat-Tree topology, algorithms have to be topology-aware in order to handle the differences on speed of heterogeneous links. Several papers discuss this and developed topology-aware broadcast, scatter/gather operations to map the logical communication topology to real network topology. Not only the algorithm itself is optimized, but also auxiliary services such as topology detection [21][22] and fault detection and recovery [23] are also added to these solutions to improve the performance and resiliency.

In MapReduce domain, there are several solutions to improve the performance of data transfers. Orchestra [18] is such a global control service and architecture to manage intra and inter-transfer activities on Spark [24]. It not only provides control, scheduling and monitoring on data transfers, but also provides optimization on broadcasting and shuffling. For broadcasting, it uses an optimized BitTorrent [25] like protocol called Cornet, augmented by topology detection. Though this method achieves similar performance as our Multi-Chain method, it is still unclear about its internal design and what kind of communication graph formed in data transfers. For shuffling, it uses weighted shuffle Scheduling (WSS) to set the weight of the flow to be proportional to the data size. But as what we discussed above, it is not helpful to applications such as K-means Clustering.

There are also other solutions to improve shuffling performance. One is Hadoop-A [26] that provides a pipeline to overlap the shuffle, merge and reduce phases, which is very similar to the mechanism in Twister. But it uses an alternative Infiniband RDMA [27] based protocol to leverage RDMA inter-connects for fast data shuffling. However, due to the limitation of test environment, we didn't see how it performs on 100+ nodes. Another is MATE-EC2 [28], a MapReduce like framework working on EC2 [29] and S3 [30] with alternate APIs. For shuffling it uses local reduction and global reduction. The mechanism is similar to what we did in Twister but it focuses on EC2 cloud environment so that the design and implementation are totally different.

For iterative MapReduce, there are also other solutions such as iMapReduce [31] iHadoop [32] to optimize to the data transfers between iterations by doing iteration asynchronously. That means there is no barrier between any two iterations. However, this doesn't work for applications which need broadcast data in every iteration because all the outputs from Reduce tasks are needed for every Map task.

VII. Conclusion

In this paper, we improved the performance of data transfers in Twister iterative MapReduce framework. We removed original broker-based methods and implemented 3 topology-aware algorithms including Multi-Chain, and Scatter-Allgather-BKT/MST. Among them, Multi-Chain method can improve broadcasting performance by 60% of original broker based broadcast method and 50% of Scatter-Allgather-BKT, a standard broadcasting algorithm for large vectors in MPI. We also improve shuffling performance to 10% of the original time by using local reduction.

There are number of directions in future work. We try to apply Twister with new data transfer methods to other iterative applications [33]. We are also transplanting our algorithms to Infiniband and test the performance gain from different methods. The initial observation suggests that the optimal choice could be different from the situation on Ethernet. Further we try to extend the methods to a complete service by building topology and link speed detection service and utilizing auxiliary service such as ZooKeeper [34] to provide coordination and fault detection.

Acknowledgement

This work is supported by National Science Foundation CAREER Award on "Programming Environments and Runtime for Data Enabled Science".

References

[1] Apache Hadoop. http://hadoop.apache.org.

[2] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. Sixth Symp. on Operating System Design and Implementation, pp. 137–150, December 2004.

[3] Dubey, Pradeep. A Platform 2015 Model: Recognition, Mining and Synthesis Moves Computers to the Era of Tera. Compute-Intensive, Highly Parallel Applications and Uses. Volume 09 Issue 02. ISSN 1535-864X. February 2005.

[4] J.Ekanayake et al., Twister: A Runtime for iterative MapReduce, in Proceedings of the First International Workshop on MapReduce and

its Applications of ACM HPDC 2010 conference June 20-25, 2010. 2010, ACM: Chicago, Illinois.

[5] Yingyi Bu, Bill Howe, Magdalena Balazinska, and Michael D. Ernst. Haloop: Efficient Iterative Data Processing on Large Clusters. Proceedings of the VLDB Endowment, 3, September 2010.

[6] J. B. MacQueen, Some Methods for Classification and Analysis of MultiVariate Observations, in Proc. of the fifth Berkeley Symposium on Mathematical Statistics and Probability. vol. 1, L. M. L. Cam and J. Neyman, Eds., ed: University of California Press, 1967.

[7] MPI Forum, "MPI: A Message Passing Interface," in Proceedings of Supercomputing, 1993.

[8] Watts J, van de Geijn R. A pipelined broadcast for multidimensional meshes. Parallel Processing Letters, 1995, vol.5, pp. 281–292.

[9] Nikhil Jain, Yogish Sabharwal, Optimal Bucket Algorithms for Large MPI Collectives on Torus Interconnects, ICS '10 Proceedings of the 24th ACM International Conference on Supercomputing, 2010

[10] E. Chan, M. Heimlich, A. Purkayastha, and R. A. van de Geijn. Collective communication: theory, practice, and experience. Concurrency and Computation: Practice and Experience, 2007, vol 19, pp. 1749–1783.

[11] Charles E. Leiserson, Fat-trees: universal networks for hardware efficient supercomputing, IEEE Transactions on Computers, vol. 34 , no. 10, Oct. 1985, pp. 892-901.

[12] PolarGrid. http://polargrid.org/polargrid.

[13] ActiveMQ. http://activemq.apache.org/

[14] S. Pallickara, G. Fox, NaradaBrokering: A Distributed Middleware Framework and Architecture for Enabling Durable Peer to-Peer Grids, Middleware 2003, 2003.

[15] Ford L.R. Jr., Fulkerson D.R., Maximal Flow through a Network, Canadian Journal of Mathematics , 1956, pp.399-404.

[16] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, The Hadoop Distributed File System. IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST), 2010

[17] S. Kumar, Y. Sabharwal, R. Garg, P. Heidelberger, Optimization of All-to-all Communication on the Blue Gene/L Supercomputer, 37th International Conference on Parallel Processing, 2008

[18] Mosharaf Chowdhury et al. Managing Data Transfers in Computer Clusters with Orchestra, Proceedings of the ACM SIGCOMM 2011 conference, 2011

[19] H. Mamadou T. Nanri, and K. Murakami. A Robust Dynamic Optimization for MPI AlltoAll Operation, IPDPS'09 Proceedings of IEEE International Symposium on Parallel & Distributed Processing, 2009

[20] P. Balaji, A. Chan, R. Thakur, W. Gropp, and E. Lusk. Toward message passing for a million processes: Characterizing MPI on a massive scale Blue Gene/P. Computer Science - Research and Development, vol. 24, pp. 11-19, 2009.

[21] Krishna Kandalla et al. Designing topology-Aware Collective Communication Algorithms for Large Scale InfiniBand Clusters: Case Studies with Scatter and Gather, IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010

[22] H. Subramoni et al. Design and Evaluation of Network Topology-/Speed-Aware Broadcast Algorithms for InfiniBand Clusters, Proceedings of the 2011 IEEE International Conference on Cluster Computing, 2011

[23] M. Koop, P. Shamis, I. Rabinovitz, and D. K. Panda. Designing High Performance and Resilient Message Passing on Infiniband. In Proceedings of Workshop on Communication Architecture for Clusters, 2010.

[24] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster Computing with Working Sets. In HotCloud, 2010.

[25] BitTorrent. http://www.bittorrent.com.

[26] Yangdong Wang et al. Hadoop Acceleration Through Network Levitated Merge, International Conference for High Performance Computing, Networking, Storage and Analysis (SC'11), 2011

[27] Infiniband Trade Association. http://www.infinibandta.org.

[28] T. Bicer, D. Chiu, and G. Agrawal. MATE-EC2: A Middleware for Processing Data with AWS, Proceedings of the 2011 ACM international workshop on Many task computing on grids and supercomputers, 2011

[29] EC2. http://aws.amazon.com/ec2/.

[30] S3. http://aws.amazon.com/s3/.

[31] Y. Zhang, Q. Gao, L. Gao, and C. Wang. imapreduce: A distributed computing framework for iterative computation. In DataCloud '11, 2011.

[32] E. Elnikety, T. Elsayed, and H. Ramadan. iHadoop: Asynchronous Iterations for MapReduce, Proceedings of the 3rd IEE International conference on Cloud Computing Technology and Science (CloudCom), 2011

[33] B. Zhang et al. Applying Twister to Scientific Applications, Proceedings of the 2nd IEE International conference on Cloud Computing Technology and Science (CloudCom), 2010

[34] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed, ZooKeeper: wait-free coordination for internet-scale systems, in USENIXATC'10: Proceedings of the 2010 USENIX conference on USENIX annual technical conference, 2010, pp. 11–11.