Accelerating Data Transfers In Twister Iterative MapReduce Framework

Bingjing Zhang

Indiana University Bloomington

*Abstract*—MapReduce becomes popular in recent years for its scalability and reliability in processing big data problems and its easy programming interface. Later, by the demands of processing iterative algorithms in MapReduce, several iterative MapReduce frameworks emerge. In previous research works, we presented our Twister Iterative MapReduce Framework. By leveraging local memory on each computing node to cache invariant data, we are able to accelerate iterative MapReduce execution instead of simple invocations of MapReduce jobs. However, many other issues also come. We find performance issues when transferring massive data intra or inter iteration. For example, in K-means Clustering the centroids are required to be broadcasted to all the Map tasks and it has to be done over iterations. Besides, local new centroids generated by each Map task are also required to be transferred in shuffling stage. As a result, the time required for broadcasting and shuffling is proportional to the number of computing nodes and the size of centroids. This could cost more than 50% job execution time and stop the application from scaling. To solve this problem, we implement several methods to improve the performance of broadcasting and shuffling. We show that the new Multi-Chain method reduces the broadcasting time to 40% of the original broker based method when broadcasting 1 GB data to 125 nodes. And we also show that the new mechanism with local reduction reduces shuffling time to about 25% of the original time when running K-means Clustering on over 100 nodes with 1 GB data of centroids.

KeyWords-Iterative MapReduce, Data-intensive applications, Data transfer, broadcasting, shuffling, Fat-Tree topology

# Introduction

MapReduce frameworks become popular in recent years for their scalability and reliability in large data processing and simplicity in programming interface. Especially Hadoop, an open source implementation following original Google’s Map-Reduce concept, have been widely used in industry and academia.

However, single Map and Reduce task pair cannot meet the requirement of executing iterative algorithms which usually have several MapReduce iterations. Repetitive disk access for fetching and merging data over iterations made the execution inefficient. Several new frameworks designed for iterative MapReduce are proposed to solve this problem, e.g. Twister[], HaLoop[].

Past effort on iterative MapReduce was on optimizing data flow. The main contribution was to reduce the times of data transfers between iterations by caching invariant data in local memory or disk of computing nodes. New scheduling mechanism was also designed to assign tasks to the node where relevant invariant data is located.

Twister, developed by our group, is such an iterative MapReduce framework. It categorizes data to static data and variable data. Static data is loaded from disk and cached into memory in configuration stage which is before MapReduce executions. Worker daemons are under the control of driver node and execute MapReduce jobs by spawning Map and Redcue task threads.

But we find that Twister could still suffer from variant data transfers. Many iterative algorithms contain shared data which are required to be broadcasted to all the computing nodes. For example, the centroids in K-means Clustering are required to be broadcasted to all the nodes. Thinking about 100MB data of centroids and 100 computing nodes, it is very inefficient if we naively send the data to the nodes one by one. Besides, since in Map stage each map task generates its local version of centroids, assuming there are 800 map tasks, then the data required to be transferred in Shuffling stage is about MB, i.e. 80GB in total! Because K-means Clustering algorithm needs several iterations to the end, most of job execution time is spent on data transfers but not on computation.

In this paper, we try to improve the performance of data transfers in Twister. We focus on two kinds of data transfer operations. One is broadcasting, another is shuffling. We use K-means Clustering as our test application. For broadcasting, we borrow the ideas from the traditional deterministic algorithms and try to apply them to Twister. We propose three new methods, Multi-chain, All-to-All-BKT and All-to-All-MST. Multi-chain creates several data transfer chains to transfer data in a pipeline style. All-to-All-BKT is an algorithm used in MPI for broadcasting large data where the data is scattered first and then each data piece is broadcasted to all the nodes by BKT algorithm. All-to-All-MST differs from All-to-All-BKT with changing BKT-Allgather to Minimum-Spanning-Tree algorithm and removes the barrier between Scatter and Allgather.

However, former research on these methods is based on the assumption that the computing nodes are fully connected and no contention in routing. But the network topology commonly used in cluster or cloud environment is Fat-Tree topology where the cost of inter-switch connection is high. Instead of copying original algorithm design to Twister directly, we make the algorithm be topology-aware.

Since data object in memory can be very large in transfers, we also carefully considered how to serialize the data to the sending byte stream efficiently. Instead of repeatedly creating byte streams to serialize different data components and then combine them into one byte stream, we provided a new message interface and mechanism to enable efficient data serialization. We also overlap data serialization time and communication time to reduce data broadcasting time.

For shuffling, because Twister already leverages memory to do shuffling, the performance is much better than Hadoop. But for the applications like K-means clustering whose intermediate data can be partially reduced, we propose a new mechanism to do local reduction first before shuffling. Since Map tasks and Reduce tasks are running on thread level in Twister daemons, we can achieve this by utilizing shared memory.

We evaluated our improvement on IU PolarGrid cluster with more than 100 computing nodes. For broadcasting, all the methods are significantly faster than the original simple broadcasting method which is based on messaging brokers. Among three methods, Multi-Chain method is about twice faster than the other two methods which are very close in performance. For shuffling, with local reduction, we can reduce the shuffling time to one fourth of the original time.

The rest of paper is organized as follows. Section 2 discusses the background knowledge of Twister MapReduce framework, Kmeans-Clustering algorithm and the cluster IU PolarGrid where we ran our tests. 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 is about related work. Section 7 gives conclusion and future work.

# Backgrounds

Twister is a MapReduce framework which can accelerate iterative MapReduce job execution by caching the invariant data into the local memory of the computing nodes. However, it still suffers from the large variant data transfers. In this section, we provide an overview of Twister and discuss about the related issues. In addition, we also show how K-means Clustering works in Twister and the difficulty of parallelization. Finally we give an introduction of the fat-tree topology in IU PolarGrid.

## Twister Iterative MapReduce Framework

Twister has several components. A single client is used to drive MapReduce jobs. And daemon processes living on work nodes handle requests from the client and execute iterative MapReduce jobs. For connections between the components, some are through messaging brokers via a publish/subscribe mechanism and the rest use TCP sockets directly. Currently Twister supports two different kinds of brokers. One is ActiveMQ [], another is NaradaBrokering [].

Twister Architecture Chart

C:\Users\zhangbj\Pictures\Picture3.tif

Twister client allows users to configure an iterative MapReduce job with static data cached in Map tasks or Reduce tasks before job execution and to drive the job iteratively by loop control. To connect iterations, the variable data are sent to workers at the beginning iteration and collected back by Combine method at the end of the iteration. In this computation model, fault tolerance is also provided by setting checkpoints between iterations.

Daemons operate on compute nodes and handle requests from the client. During configuration stage, Map and Reduce workers are created and static data is loaded from the local disk according to records in a partition file and cached into memory. Then in execution stage, daemons start to execute MapReduce tasks threads. Twister uses static scheduling for workers in order to leveraging the local data cache.

In current released version, there is no support of distributed file system. Twister lets intermediate data be transferred directly between daemon processes. Twister uses scripts to operate static data on local disks of computing nodes in order to simulate characteristics of distributed file systems. Since this solution lacks reliability and scalability. We are moving to use distributed file systems and some development on adding HDFS support to Twister is done.

Past attempts are to use messaging brokers to do data transfers. First few Twister prototypes were only using one broker. One broker is sufficient for small messages transfers. But for large data transfers, it becomes a single hot spot and stops Twister from scaling. Then we move to solutions which can use a network of brokers in order to achieve better scalability and performance. But we still find several 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.
* Broker network doesn’t provide optimal route to transmit data between a set of brokers and clients. In a network of brokers, one broker gets the message and forwards it directly.
* Reliability and management issues. Brokers are not always reliable in message transmission. Message could get lost without notification. Since message broker could fail. The potential failure points in the system also increase as the number of brokers increase. This brings additional work to manage the status of the brokers.

For these reasons, in current released version of Twister, we already use data shuffling through TCP sockets since broker cannot sustain when data size is large. Some remedies such as broker full mesh network and broker all-to-all connection were used for broadcasting. They are faster than single broker broadcasting but far from the optimal. Due to the reasons above, they are abandoned finally.

## K-means Clustering

K-means Clustering is an iterative algorithm to partition data points into a given number of clusters. At the beginning, a set of centroid data points are generated (usually these centroids are picked from the data points). Then each data point goes through the whole centroids list to assign itself to a cluster by finding the closest centroid. Once all the data points are assigned, the positions of new centroids (cluster center) are recalculated and each is the mean of the points in that cluster. After several iterations, the positions of centroids reach the local optimization.

Algorithm flow

C:\Users\zhangbj\Pictures\Picture3.tif

We observe that only the positions of centroids change in the whole process, not the positions of data points. So for Twister K-means Clustering, the process of execution is as follows: firstly we partition data points and cache them to Map tasks. Centroid data are broadcasted to all the Map tasks. Then each Map task assigns the partial data points it owns to the clusters and output the partial sum of data points in each cluster. We can use only one reducer or multiple reducers here if we give different cluster an ID. Each Reduce task collects the partial sum of data points from all the Map tasks and calculates the mean with the total sum of all the data points. By combining these results from Reduce tasks and the driver gets the update of centroids data.

There are two issues here to stop the application from scaling. First, the centroids data is required to broadcast to all the nodes. Though centroids data size is much smaller than the size of data points, it could still go to MB or GB level. In this situation, simple broadcasting method such as sending data one by one through a broker is extremely slow. Considering that sending 1GB data from point to point needs 10 seconds through a 1Gbps link, then sending 1GB to 100 nodes probably needs about 1000 seconds.

Another issue is that, each Map task generates a partial sum of data points to each cluster so that each Map task has to output data with the size equal to the centroid data just broadcasted. That means in shuffling stage, there are 800 GB data required to be transmitted assuming there are 100 nodes, 800 Map tasks is 800 and 1 GB centroids data.

## IU PolarGrid

IU PolarGrid cluster 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.

Structure chart

C:\Users\zhangbj\Pictures\Picture3.tif

This kind of topology can easily cause contention when there are many inter-switch communication pairs 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 connections between different node pairs. Assuming that every 1 GigE link to each node is fully utilized, a 10 GigE connection can only support 10 parallel communication pairs across leaf switches in maximum. Otherwise the inter-switch communication pairs could affect each other in performance. As a result, in the design of efficient collective communication algorithms on fat-tree topology, reducing the inter-switch communication is the first thing to be considered.

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 is 16 GB.

# Broadcasting transfers

As what we analyzed above, large data broadcasting can easily become the bottleneck of many iterative applications such as K-means Clustering. To solve this problem, we did many attempts. Since first few prototypes of Twister are based on message brokers, initial we tried to use multiple brokers to improve broadcasting speed. Two types of broker topology are tried here. One is a full mesh broker network where each broker is connected with other brokers. Another is a set of unconnected brokers where each broker connects to all the client processes. Both these two methods have performance issue because message routes between brokers or between brokers and clients are not optimal. In addition, broker network also cannot scale well. Then we move to other broadcasting methods which don’t depend on brokers but are based on TCP socket connection directly. With ideas borrowed from MPI collective communication algorithms, we implemented two methods, Scatter-AllGather-BKT and Scatter-AllGather-MST. Both of these methods have similar good performance and scalability. But network contention can still happen in these two methods. At last, to fully utilize the bandwidth on each link, we implemented Multi-Chain; a method based on multiple pipelines which can outperform Scatter-AllGather based methods. We also considered how to utilize physical topology information to achieve better performance. To illustrate the performance model, we use as the number of daemon processes (each node has one process), as the number of brokers, as the number of parts of data if special partition is required, as the data size, as communication startup time and as data transfer time per unit.

## Broker-Based Methods

Some remedies to existing solutions are to move from a single broker to multiple brokers. Two methods are tried to improve the broadcasting performance. One is full mesh broker network and another is unconnected broker array.

In full mesh broker network, every broker connects with the rest of brokers. 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 different 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 the rest of brokers with each one a copy. 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 does data forwarding in parallel. Network contention can also be avoided if each broker and the related clients are in the same switch. The performance model can be established as follow:

When, we can get. Though it is much better than simple forwarding in performance, it is still very slow. Considering the example we used above for 1GB broadcasting on 100 nodes, this method needs 10 brokers and about 200 seconds to finish! Besides, due to the engineering issues, we notice that message could lose occasionally with unknown reason when the broker network grows larger.

Structure chart

C:\Users\zhangbj\Pictures\Picture3.tif

Another topology we tried is unconnected broker array. In this method, each broker is connected to all the client processes and each client tries to balance the workload of data sending on each broker connection. In this topology, broadcasting is done in a style of “divide, distribute and gather”. Data is split and sent to different brokers first. Then there is an AllGather operation. Each broker broadcasts the portion of data it has to all the client processes. Ideally, if everything goes parallel, we can have the following model:

Since can be ignored in large data transferring, we conclude that. However, this won’t be true in real experiments. The foundational problem is that we cannot control the routes of messages in broker based com-munication and congestion can happen on some links. Scatter part and AllGather part can overlap and the data sending in AllGather can affect each other. So the performance degrades and varies a lot. There is also issue in deploying this kind of structure in large scale. Here the number of connections on each node grows as grows; however, the number of connections each node can support is limited. As goes to a larger number, failures on some broker-client connection can happen. In the experiments, we notice that the number of brokers is hard to exceed 70 so that we have to set limited number of brokers. But we still see better performance compared to the first topology.

Structure chart

C:\Users\zhangbj\Pictures\Picture3.tif

## Scatter-Allgather Methods

Successes on MPI collective communication algorithms makes us turn to use TCP socket connection directly in order to control the message routes in broadcasting by ourselves. We call these methods Scatter-AllGather methods because they all follow the principle of “divide, distribute and gather”. The data are firstly split and scattered to all the nodes, and then an AllGather operation happens where data on every node is broadcasted to all the rest nodes. For this phase, we have two similar methods. One uses Bucket algorithm and another 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 firstly scatters the data to all the nodes without using any special algorithm. Then it views the nodes as a chain. At each step, all nodes send data to the node to their right. By taking advantage of the fact that messages traversing a link in opposite direction do not conflict, we can do AllGather operation in parallel without any network contention. The performance model can be es-tablished as follow:

Since we can control every step in the algorithm, we set a barrier between Scatter and Allgather to prevent them from affecting each other. In Allgather stage, we make the com-munication be topology-aware to the Fat-Tree topology, i.e. nodes connected to the same leaf switch are close to each other in the chain and each data only travel between any two leaf switches once. This makes the performance of Scatter-Allgather-BKT algorithm much better than broker based methods. But in experiments, we find the real performance is still slower than the theoretical performance. The reason is that, it is impossible to enable all the nodes to do Allgather at the same global time through sending control messages from the driver. As a result, some links have more load than the others and thus network contention happens.

Algorithm step chart

C:\Users\zhangbj\Pictures\Picture3.tif

An alternative method is Scatter-AllGather-MST which uses Minimum-Spanning-Tree (MST) algorithm rather than Bucket algorithm. To reduce the conflict, each node builds its own MST with itself as root and the barrier between two phases is also removed in order to let the data be forwarded in the tree without delay. In this fashion, the whole algorithm flow is as follow: the driver firstly scatters the data in a random order. Once each node gets the data, it forwards data in a MST.

In each MST, the ranks in the tree are arranged as follow: assuming rank 0 is node with ID (), then rank is the node with. So for every node, its rank is different in each tree. One node may be a leaf node in one tree but an inner node in another tree. We let these trees all be topology-aware by making nodes close in ID be also close in topology. 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 forwarding is balanced and the link contention between trees is also reduced by removing the barrier to let partial data transferred prior to the others. As a result, performance is still gained and the results are similar to Scatter-AllGather-BKT in experiments.

Algorithm step chart

C:\Users\zhangbj\Pictures\Picture3.tif

## Multi-Chain

Different from non-pipeline methods above, we present Multi-Chain method here, a broadcasting algorithm based on pipelines. In this method, nodes in Fat-Tree topology are treated as a linear array and data is forwarded from one node to its neighbor every time. The performance is gained by dividing the data into many small parts and overlapping the transmission stages. For example, one would send the first piece of the message 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 [pipeline bcast paper].

Since in Fat-Tree topology each node only has two links which is much less than the number of links per node in Mesh/Torus topology, chain broadcasting can maximize the utilization of the links per node. We also make the chain be topology-aware by putting nodes connected to the same switch to be close in the chain. If nodes are not evenly distributed among switches, e.g., then we put the nodes in at the beginning of the chain, then nodes in at the second, and then … by the same way.

Topology aware chain structure

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

Since is large in our transfers, when,

However the speed of data transfers on each link could not be at the same speed, then network congestion could happen at some time or place in the network if you keep forwarding the data into the pipeline. So we add barriers inside of the execution flow. The data is partitioned to blocks so that each block is broadcasted in a pipeline called “small pipeline” and the transmission of the whole data is also broadcasted in a pipeline called “big pipeline”. In the small pipeline, every node receives the partial of the data block and forwards it to the next node directly. No barrier is required to coordinate the behavior of this node. The first block from the first node is broadcasted in this way. Then before sending the second block, the first node needs to wait for an ACK sent from the second node. For the second node, once it finishes the action of receiving and forwarding the data from the last block, it sends an ACK to the first node to let the first node send the next data block and wait for an ACK from the third node to see if it gets the data forwarded by the second node. Once these two conditions are met, another small pipeline starts to transfer the second data block. In the experiments, we use block size 4MB and it works best.

To better utilize the bandwidth and improve the performance, we also extend chain to multiple chains with each thread managing a chain. For nodes in IU PolarGrid, we create 8 chains in different threads to do broadcasting.

## Auxillary Steps in Broadcasting

(Data Abstraction and data presentation, KeyValue pairs can be divided and small)

The broadcasting operation is to send data from the driver’s memory through the network to the daemons’ memory. To start and end the whole process, auxiliary work is required, including topology learning, data serialization, overlapping between serialization and transferring, and mechanism for providing resiliency.

Currently topology detection is not implemented. To let each node learns the topology, we put the topology information in a property file and let each node read it from the local disks. Then all the nodes have the global view of the topology before starting broadcasting.

In user program, broadcasting data are presented as objects. They are required to be serialized to byte arrays through user defined serialization method before broadcast-ing. Once the serialized data arrives to receivers, they are de-serialized to objects. For a single big data object, we find its serialization time is extremely long so that it makes the whole broadcasting process very inefficient. So we provide interfaces to enable users to send a series of small objects split from the original single large object instead of sending the whole directly.

Using small data objects also enable us to parallelize the serialization and overlap it with communication to improve the overall performance of broadcasting. In Multi-Chain, we make a producer/consumer model. Threads for serialization act as producers to produce serialized data objects and put them into a queue while threads for chain broadcasting pick data from the queue and send it. In Scatter phase of Scatter-AllGather methods, we let each thread serialize the data object it owns and send to the destination.

At the end of the broadcasting, the driver waits and checks if all the nodes have received all the data blocks. Each node sends ACK messages through the broker to tell the driver the current progress of data receiving. We also adapted several strategies to make the whole process fault tolerant. For failures in each node-to-node sending step, we do retry first or jump to the other destinations next. If driver doesn’t get all the ACK within a time range finally, it restarts the process of broadcasting.

C:\Users\zhangbj\Pictures\Picture1.tif

Small Pipeline

C:\Users\zhangbj\Pictures\Picture2.tif

Big Pipeline

# Shuffling transfers

During the shuffling phase, the KeyValue pairs generated from Map tasks are regrouped by keys and each <Key, <Value>> is processed by a Reduce task. In original MapReduce framework, this operation heavily depends on the distributed file system used because of repetitive merges and disk access. Since this could be very inefficient, in Twister, we leverage memory to do shuffling operation so that the whole process is different from the one in original MapReduce. We transfer intermediate data through the network from the memory of the node where the Map task stays to the memory of the node where the Reduce task locates.

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 bytes, is the number Map task threads per node, is the number of the node, and is the data size of centroids generated by each Map task. So even the data of centroids is small, it can cause large intermediate data. Considering a situation of IU PolarGrid with 5 switches and 20 nodes per switch and 8 Map tasks per node, 1 GB data of centroids and can generate about 800 GB data in shuffling. If all the computing nodes are fully utilized to execute the Reduce task, for a group of 20 nodes connected to the same switch in the shuffling stage, they need to receive data with size equal to from the other 4 switches. But for these 20 nodes, the link speed between leaf switch and core switch is only 10 Gbps, it turns out that at least 100 seconds are required to transmit these data. Because of the inefficiency of transferring large amount of data, we try to reduce the intermediate data size to minimum by using local reduction across Map tasks. Using the example above, after local reduction, the intermediate data size is one eighth of the original data.

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.

## Memory-leveraging Shuffling in Twister

Instead of disk-based repetitive merge in MapReduce frameworks like Hadoop, the current Twister does shuffling in memory. The initial prototypes of Twister relies brokers to send intermediate data. As what we analyzed in broadcast transfers, broker based method cannot scale. The current shuffling in Twister uses direct TCP transfers. The process has 2 stages including sending data head and fetching the real data.

In Twister, each Map task is located in a daemon process and executed by a thread. Once a KeyValue 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 the Map task finishes and all the KeyValue pairs are ready, the thread continues working to send out the data. There are two different kinds of routes. If the data size is really 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, then the daemon process where the Reduce task resides, processes the message and fetch the real data by using direct TCP transfers.

So this is the second stage. 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 becoming a bottleneck in fetching too much data at the same time moment. 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 fashion.

Flow Chart

C:\Users\zhangbj\Pictures\Picture3.tif

## Local Reduction

The mechanism currently used in Twister is efficient when it is 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) to balance the data transfers by making the number of transferring flow be proportional to the data size. But for K-means Clustering, this is not helpful at all because every Map task generates the same amount of data and then there is no space for optimization.

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

()

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

By leveraging the property we find above and the fact that Map tasks are executed as threads in Twister daemon processes, we do local reduction in shared memory first. Once a Map task is finished, it doesn’t send data out directly but cache the data to a shared memory pool. When the key conflict happens, the program invokes user defined operation to merge two KeyValue 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 is one eighth of the original data. As a result, much time can be saved in shuffling.

Flow Chart

C:\Users\zhangbj\Pictures\Picture3.tif

## New Interface Design

To support shuffling and local reduction, we provide new interfaces to let user define the Key and Value objects. We have a general interface named TwisterSerializable for message serialization and the interfaces for Key and Value extended from that. (Abstraction and data presentation)

Originally in Twister, we serialize each data object into a byte array. However, it is very inefficient when in shuffling stage a large number of KeyValue pairs are required to be serialized into a single byte array and sent to one destination because the byte streams have to be created repeatedly to serialize each KeyValue pair. Now the new interface is changed to delegate TwisterMessage object to do seriali-zation. With TwisterMessage object, user can use its APIs to serialize multiple data objects 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 of Key and Value are defined. In the interface of Key, an API isMergeableInShuffle is defined to check if the current KeyValue pair can be merged in shuffling. At the same time, the interface of Value, mergeInShuffle is defined. It can take a Value object and merge its contents to the current Value object. Besides, it only works when isMergeableInShuffle in Key returns true.

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);

}

# Expreiments

We do experiments on IU PolarGrid to evaluate the per-formance 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 benchmark-ing 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 out-perform the original shuffling significantly.

## Broadcasting

The following broadcasting methods are tested in IU PolarGrid: Scatter-Allgather-Broker, Scatter-Allgather-BKT, Scatter-Allgather-MST, 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. The major differences of the first three are in the Allgather stage, while the first one uses broker, the second one uses bucket algorithm, but the third one uses minimum-spanning tree.

1. 1GB bcast
2. 100 MB bcast
3. CDF

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.

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 broadcasting on different data size, including 100 MB and 1GB. Each test is done 10 times. The performance results are given in Figure 1 and 2.

We use different data chunking settings on different algorithms according to their behaviors. For Multi-Chain, each chunk is about 4MB. That means for 1 GB data broadcasting, there is about 250 chunks and for 100 MB data, there is about 25 chunks. For Scatter-Allgather-BKT/MST, 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

1. Topology comparison
2. xample of a figure caption. (figure caption)

to the settings in Multi-Chain. For the broker-based method, we set the number of chunks as the number of brokers. Except for 1 node test, we also use a small chunk size as what we use in Scatter-Allgather methods.

On 1 GB data broadcast, the performance results show that Multi-Chain has the best performance over all the scales. For Scatter-Allgather-BKT and Scatter-Allgather-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 dull route selection in Allgather stage. Multi-Chain outperforms it with.

However, on 100 MB data broadcast, the chart tells a different story. The performance of Multi-Chain jitters in large scale. Though it can achieve the highest performance, its average performance is slower than Scatter-Allgather-BKT due to the variation from executions to executions. 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.

1. xample of a figure caption. (figure caption)
2. CDF 1GB per Map Task

We also present CDF of completion times of data chunks received in 1 GB data broadcast on 125 nodes 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 messages. This is probably due to the contention of receivers for receiving messages from different minimum spanning trees.

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 outperform the other.

## 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 scale of 125 nodes and do experiments.

To benchmark the performance of shuffling, we lower the number of data points each Map task processes in order to shorten the execution time. Figure 6 shows the time difference on shuffling with/without local reduction.

1. 1GB centroids k-means clustering simulation on 125 nodes
2. 10K centroids and 1million data communication cost

We measure the time from the start of shuffling to the end of Reduce. Time costs on Reduce tasks are included. But they can be ignored since they are very small when compared with the data transfer time.

There are 1000 Map tasks on 125 nodes. 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 other. For 100 MB data per Map task, the time of shuffling with local reduction is about half of the other. We also give a closer view with CDF of completion times of Reduce tasks (figure). We see that for shuffling without local reduction, all the reduce tasks are required to wait for time.

We also benchmark the whole application to see how much the performance improvement we again in data transfers by changing from the old method to the new one. We tested K-means Clustering with 1 GB 250K 500D centroids 1 data point per node on 125 nodes. The old one method uses broker based method to do broadcasting and with no local reduction in shuffling. The new one uses Multi-Chain broadcasting and shuffling with local reduction. We find that per iteration time cost is only 20% of the original time.

We also test K-means Clustering with real data: 1 million data points and 10K cluster. Here each data point has 500D which illustrates 500 features from an image. We test it on a smaller scale with 80 nodes. The old method uses Scatter-Allgather-Broker method to do broadcasting and shuffling with no local reduction. The new one uses Scatter-Allgather-BKT to do broadcasting and shuffling with local reduction. We show that the communication cost per iteration of the new method is only 50% of the old one.

# 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 []. Basic algorithms are pipeline broadcast method [], minimum-spanning tree method, bidirectional exchange algorithm, and bucket algorithm []. Since these algorithms have different benefits, algorithm combination is widely used to improve the communication performance []. And some solution also provides auto algorithm selection [].

However, many solutions have their own focus which is different from our work. They focus on small data transfers and the largest data size tried is up to megabytes level. The data type is vectors and arrays which are not as complicated as structures like objects. Many algorithms such as Allgather have the assumption that each node has the same amount of data but this is not common in MapReduce computation model. As a result, though shuffling can be viewed as all-to-all communication such as Reduce-scatter operation, its algorithm cannot be applied directly on shuffling because the amount of data generated by each Map task is uneven in most cases.

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 between heterogeneous links. Several papers discussed about this and developed topology-aware broadcast, scatter/gather operations to map the logical com-munication topology to real network topology []. Not only the algorithm itself is optimized, but also auxiliary services such as topology detection [] and fault detection and recovery [] 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 [] is such a global control service and architecture to manage intra and inter-transfer activities on Spark. 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-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 K-means Clustering.

There are also other solutions to improve shuffling performance. One is Hadoop-A [] 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 RDMA-based protocol to leverage RDMA interconnects 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 [], a MapReduce-like framework working on EC2 and S3 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 and the design and imple-mentation are totally different.

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

# Conclusion

In this paper, we improved the performance of data transfers in Twister iterative MapReduce framework. We removed original broker-based methods because of its cumbersome and clumsy. We implemented 3 topology-aware algorithms including Multi-Chain, Scatter-Allgather-MST, and Scatter-Allgather-BKT. Among them, Multi-Chain method can improve broadcasting performance to 40% of original broadcast method and 50% of stand MPI broadcast algorithm for large vectors. We also improved shuffling performance to 25% of original time by using local reduction.

There are number of directions in future work. Currently we are transplanting our algorithms to Infiniband and test the performance gain from different methods. The initial observation suggests that the conclusion could be different from the situation on Ethernet. We are also try to add extend the algorithm to a whole service by building topology and link speed detection service and utilizing auxiliary service such as ZooKeeper to provide better coordination and fault detection and tolerance.

##### Acknowledgement

##### References

1. Twister
2. Hadoop
3. HaLoop
4. pipeline
5. Collective algorithms
6. Auto selection