# High Performance Clustering of Social Images in a Map-Collective Programming Model

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

Large-scale iterative computations are common in many important data mining and machine learning algorithms needed in analytics and deep learning. In most of these applications, individual iterations can be specified as MapReduce computations, leading to the Iterative MapReduce programming model for efficient execution of data-intensive iterative computations interoperably between HPC and cloud environments. Further one needs additional communication patterns from those familiar in MapReduce and we base our initial architecture on collectives that integrate capabilities developed by the MPI and MapReduce communities. This leads us to the Map-Collective programming model which here we develop based on requirements of a range of applications by extending our existing Iterative MapReduce environment Twister. This paper studies the implications of large scale Social Image clustering where large scale problems study 10-100 million images represented as points in a high dimensional (up to 2048) vector space which need to be divided into up to 1-10 million clusters. This Kmeans application needs 5 stages in each iteration: Broadcast, Map, Shuffle, Reduce and Combine, and this paper focuses on collective communication stages where large data transfers demand performance optimization. By comparing and combining ideas from MapReduce and MPI communities, we show that a topology-aware and pipeline-based broadcasting method gives better performance than other MPI and (Iterative) MapReduce systems.

## Categories and Subject Descriptors

C.2.4 [**Computer-Communication Networks**]: Distributed Systems – *Distributed applications.*

## General Terms

Algorithms, Measurement, Performance, Design, Experimentation.

## Keywords

Social Images, Data Intensive, High Dimension, Iterative MapReduce, Collective Communication

## 1. INTRODUCTION

The rate of data generation now exceeds the growth of computational power predicted by Moore's law. Challenges to computation are related to mining and analysis of these massive data sources for the translation of large-scale data into knowledge-based innovation. 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.

However 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. These and other observations suggest that iterative data processing runtime will be important to a spectrum of e-Science or e-Research applications as the kernel framework for large scale data processing. Several new frameworks designed for iterative MapReduce have been proposed to solve this problem, including Twister [4], Spark [5] and HaLoop [6]. The initial version of Twister targeted optimization of data flow and reducing data transfer between iterations by caching invariant data in the local memory of compute nodes but it did not support the communication patterns needed in many applications and we observe that a systematic approach to collective communication is essential in many iterative algorithms. Thus we generalize the (iterative) MapReduce concept to Map-Collective noting that large collectives are a distinctive feature of data intensive and data mining applications. This is supported by the remarks that "MapReduce, designed for parallel data processing, was ill-suited for the iterative computations inherent in deep network training" [7] from a recent paper on deep learning.

Social image clustering is such an application which is not only a big data problem but also needs an iterative solver. This produces challenges for both new algorithms and efficiency of the parallel execution which involves very large collective communication steps. We are addressing [8] the overall performance with an extension of Elkan's algorithm [9] drastically speeding up the computing (Map) step of algorithm by use of the triangle inequality to remove unnecessary computation. However this improvement just highlights the need for efficient communication which is a major focus of this paper. Note communication has been well studied, especially in MPI, but social image clustering stresses different usage modes and message sizes from most previous applications. In this paper, we study characteristics of large-scale image clustering application and identify performance issues of collective communication. Our work is presented in the context of Twister but the analysis is applicable to both MapReduce and other data-centric computation solutions.

At this point, let us provide some details on the 7 million image feature vectors clustering problem previously mentioned. We execute the application on 1000 cores (125 nodes each of which has 8 cores) with 10000 Map tasks and 125 Reduce tasks. In broadcasting, the root node (driver) broadcasts 512 MB of data to all compute nodes therefore the overhead of a sequential broadcasting is substantial. In shuffling, 20 TB of intermediate data generated in Map stage are required to be transferred to

Reduce so that it is not possible to handle such a large amount of data in memory. In this paper, we propose a topology-aware pipeline-based method to accelerate broadcasting by at least a factor of 120 compared with simple algorithm (sequentially sending data from root node to each destination node). Our findings demonstrate that this strategy outperforms classic MPI methods [10] by 20%. We also use local aggregation in Map stage to reduce the size of intermediate data by at least 90% and reduce the 20 TB intermediate data to 250 GB. These methods provide important collective communication capabilities to our new iterative Map-Collective framework for data intensive applications. Finally we evaluate our new methods on the PolarGrid [11] cluster at Indiana University.

The rest of the paper is organized as follows. Section 2 discusses the image clustering application. Section 3 discusses collective communication in Twister and other environments Section 4 presents the design of the broadcast Collective. Section 5 investigates how the local aggregation mechanism works. Section 6 details the experiments and results while Section 7 discusses related work. Finally in Section 8 we present our conclusions and discuss future projects.

## 2. IMAGE CLUSTERING APPLICATION

Areas involving studies of images have recently been revolutionized by the Internet that is providing an incredible volume of data. For example, there are 500 million images uploaded everyday on Facebook, Instagram and Snapchat (such sites are what we term social and surprisingly are much larger than Flickr) with 100 hours of video (video can be considered as several images per second) uploaded to Youtube every minute. This is motivating large scale computer vision and deep learning studies that need the infrastructure studied here. Our target image clustering application groups millions of images into millions of clusters each of which contains images with similar visual features. Before starting image clustering, the dimensionality reduction is done on original images first and each image is represented in a much lower space (although retaining dimensions of 512-2048) with a set of important visual components which are called “feature vectors”. Analogous to the use of “key words” in a document retrieval system, these “features vectors” become the “key words” of an image. Here we select 5 patches from each image and represent each patch by a HOG (Histograms of Oriented Gradients) feature vector of 512 dimensions. The basic idea of HOG features is to characterize the local object appearance and shape by the distribution of local intensity gradients or edge directions [12] (See Figure 1). In input data, each HOG feature vector is presented as a line of text starting with picture ID, row ID and column ID, which are then followed by 512 numbers $f_1$, $f_2$ …and $f_{dim}$.

We apply K-means Clustering [13] to cluster the similar HOG feature vectors as well as using Twister MapReduce framework to parallelize the computation. We depict K-means Clustering algorithm as a chain of MapReduce jobs. The input data consists of a large number of feature vectors each of which contains 512 dimensions and use Euclidean distance calculation to compare the distances between feature vectors and the cluster center vectors (centroids). Since the vectors are static over iterations, we partition (decompose) the vectors and cache each partition in memory. Afterwards a Map task is assigned to it in the job configuration. During each iteration execution, the job driver broadcasts centroids to all Map tasks. Each Map task then assigns feature vectors to their nearest cluster centers based on Euclidean

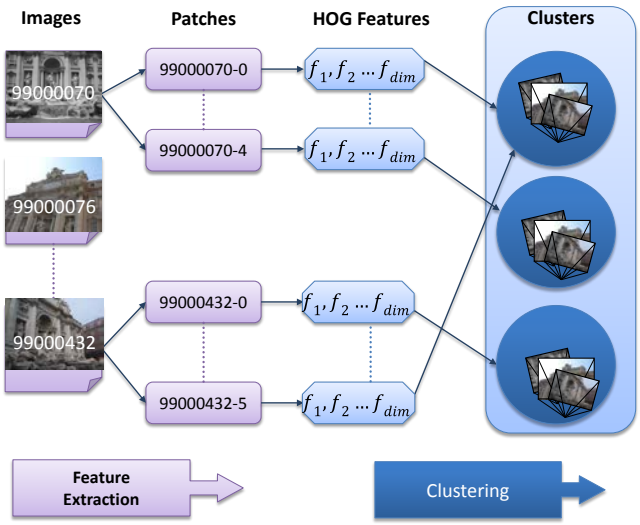


**Figure 1. Workflow of the image clustering application**

distance calculation. Map tasks calculate the sum of vectors associated with each cluster and count the total number of such vectors. The Reduce task (to simplify this description, we use only one Reduce task here but 125 are used in implementation) processes the output collected from each Map task and calculates new cluster centers of the iteration by adding all partial sums of partial cluster center values together, then dividing it by the total count of the data points in the cluster. By combining these new centroids from Reduce tasks, the job driver gets all updated centroids and the control flow enters the next iteration (see Table 1).

One major challenge of this application is the amount of image data can be very large. Currently we have near 1 TB of data and we expect problems to grow in size by one to two orders of magnitude. For such a large amount of input data, we can increase the number of machines to reduce the data size per node, but the

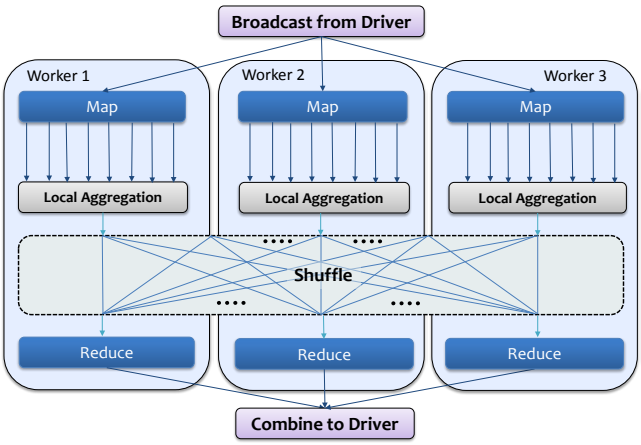


**Figure 2. Image clustering control flow in Twister with new local aggregation feature in Map stage**

total data size (of cluster centers) transferred in broadcasting and shuffling still grows as the number of centers multiplies.

For example, we cluster 7 million vectors to 1 million clusters. In one iteration, the execution is done on 1000 cores in 10 rounds with a total of 10000 Map tasks. Each task only needs to cache 700 vectors (358KB) and each node needs to cache 56K vectors,

about 30MB in total. But for broadcasting data, the number of cluster centers is very large and the total size of 1 million cluster centers is about 512MB. Therefore the centroids data per task received through broadcasting is much larger than the image feature vectors per task. Since each Map task needs a full copy of the centroids data, the total data sent through collective communication grows as the problem size and number of nodes increases. For the example above, the total data broadcasted is about 64 GB (because Map tasks are executed on thread level, broadcast data can be shared among tasks on one node).

**Table 1. Algorithms and implementation of Image Clustering Application (one Reduce task only)**

**Algorithm 1 Job Driver**

```
numLoop ← maximum iterations
centroids[0] ← initial centroids value
driver ← new TwisterDriver(jobConf)
driver.configureMapTasks(partitionFile)

for(i ← 0;  i < numLoop; i ← i+1)
   driver.broadcast(centroids[i])
   driver.runMapReduceJob()
   centroids[i+1] ←driver.getCurrentCombiner().getResults()
```

**Algorithm 2 Map Task**

```
vectors ← load and cached from files
centroids ← load from memory cache
minDis ← new int[numVectors]
minCentroidIndex ← new int[numVectors]

for (i ← 0; i < numVectors; i ← i+1)
   for (j ← 0; j < numCentroids; j ← j+1)
      dis ← getEuclidean(vectors[i], centroids[j])
      if (j = 0)
         minDis[i] ← dis
         minCentroidIndex[i] ← 0
      if (dis < minDis[i])
         minDis[i] ← dis
         minCentroidIndex[i] ← j
localSum ← new int[numCentroids][512]
localCount ← new int[numCentroids]
for(i ← 0; i < numVectors; i ← i+1)
    localSum[minCentroidIndex[i]] +← vectors[i]
    localCount[minCentroidIndex[i]] +← 1
collect(localSum, localCount)
```

**Algorithm 3 Reduce Task**

```
localSums ← collected from Map tasks
localCounts ← collected from Map tasks
totalSum ← new int[numCentroids][512]
totalCount ← new int[numCentroids]
newCentroids ← new byte[numCentroids][512]

for (i ← 0; i < numLocalSums; i ← i+1)
   for (j ← 0; j < numCentroids; j← j+1)
      totalSum[j] = totalSum[j] + localSums.get(i)[j]
      totalCount[j] = totalCount[j] + localCounts.get(i)[j]
for (i ← 0; i < numCentroids; i← i+1)
   newCentroids[i] = totalSum[i]/ totalCount[i]
collect(newCentroids)
```

We now reach the shuffling stage. Here each Map task generates about 2 GB of intermediate data so that the total intermediate data size is about 20 TB. This far exceeds the total memory size of 125 nodes (each of which has 16 GB memory; 2 TB in total). Besides it also makes the computation difficult to scale as the data size grows with the number of nodes. In this paper, we successfully reduce 20 TB of intermediate data to 250 GB with local aggregation in the Map Stage (See Figure 2). But due to the memory limitation, 250 GB still cannot be handled by one Reduce task. We further divide the chunk size of the output from each Map task to 125 blocks (numbered with Block ID from 0 to 124) and use 125 reduce tasks (one task per node) to process the intermediate data. In this way, each Reduce task only processes 2 GB of data. Reduce task 0 processes all Block 0 from all Map tasks, Reduce task 1 processes all Block 1 from all Map tasks, and so on and so forth. The output from each Reduce task is only about 4 MB so that the total data on 125 Reduce tasks that needs to send back to the driver in Combine stage is about 512 MB which is relatively small and easy to handle.

In Table 2, we give the time complexity of each part of the algorithm; we use $p$ as the number of nodes, $m$ as the number of Map tasks and $r$ as the number of Reduce tasks. For the data, $k$ is the number of centroids, $n$ is the total number of image feature vectors, and $l$ is the number of dimensions. We note for map, an approximate estimate from [8] of the improvement gotten by using triangle inequalities.

**Table 2. Time complexity of each stage**

| Stage | Simple | Improved |
|---|---|---|
| Broadcasting | $O(klp)$ | $O(kl)$ |
| Map | $O(knl/m)$ | $O(kn/m)$ [8] |
| Shuffle | $O(mkl/r)$ | $O(pkl/r)$ |
| Reduce | $O(mkl/r)$ | $O(pkl/r)$ |
| Combine | $O(kl)$ | $O(kl)$ |

# COLLECTIVE COMMUNICATION IN PARALLEL PROCESSING FRAMEWORKS

In this section, we compare several big data parallel processing tools and show how they are applied on big data problems. These tools are MPI, Hadoop MapReduce and MapReduce-like tools supporting iterative algorithms such as Twister and Spark [5]. Furthermore, we analyze the pattern of collective communication and how intermediate data is handled in each tool. In future, we expect the ideas of these tools to be all converged in a single environment for which our new optimal communication is aimed in order to serve big data applications. Section 3.1 discusses the runtime model of these tools and Section 3.2 talks about collective communication and in these tools.

## 3.1 Runtime Models

MPI, Hadoop, Twister and Spark are four tools which have very different runtime models, which are aimed at different types of applications and data. We classify parallel data processing and communication patterns [14] in Figure 3. In the whole data tool spectrum, Hadoop and MPI are two tools at opposite ends while Twister, Spark and other MapReduce-like tools congregate in the middle with mixed features extended from both Hadoop and MPI. Here we propose using systematic support of collectives to unify these models.

MPI is a computation-centric solution. It mainly serves scientific applications which are not only complicated in communication patterns but also intensive in computation. It can spawn parallel processes to compute nodes, although users need to define the computation in each process and handle communications between

them. MPI is highly optimized in communication performance. It not only offers basic point-to-point communication but also provides collective communication operations. MPI runs on HPC or supercomputers where data is decoupled from computation and stored in a separate shared and distributed file system. MPI doesn't have unified data abstraction as Key-Value pairs in MapReduce-related tools. In contrast, it is flexible enough to organize and process different types of data. MPI doesn't have fixed control flow, endowing it with the flexibility to emulate the MapReduce model or other user defined models [15-17].

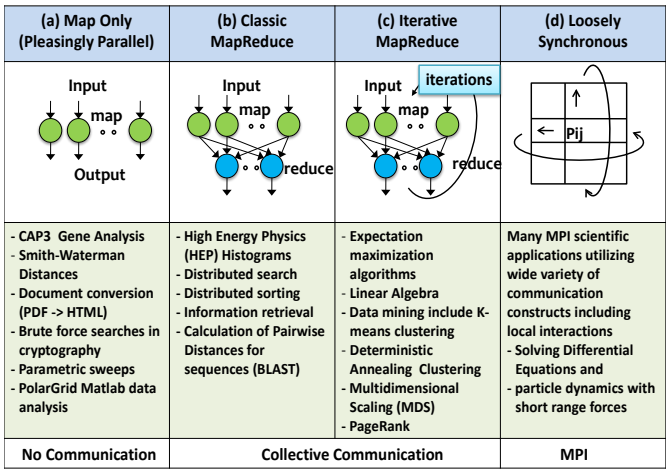


**Figure 3: Classification of Applications and Communication Patterns**

On the other hand, Hadoop is a data-centric solution. HDFS [18] is used to store and manage big data so that users do not need to think about data accessing and loading steps that must be presented in MPI programs. In addition, all computations are performed in the same place where the data is located, so that this framework is scalable when processing big data. Hadoop is inefficient for processing data mining algorithms and scientific applications served by MPI because its control flow is constrained to a Map-Shuffle-Reduce pattern. However, Hadoop is suitable for processing records and logs. This kind of data is easy to split into small Key-Value pairs with words or lines. Key-Value pairs are the core data abstraction in MapReduce. With keys, intermediate data values are labeled and regrouped automatically without using explicit communication commands. A typical records or logs processing includes information extraction and regrouping. It can be easily expressed in Map-Reduce: intermediate Key-Value pairs are first extracted from records and logs in Map tasks then regrouped in shuffling and last processed by Reduce tasks.

The difference of data and application also influences the scheduling strategies. In many scientific applications, the workload can be evenly distributed on each compute node. As a result, MPI uses static scheduling. But for logs and records processing, the workload in each task is hard to estimate. Some tasks generate more intermediate Key-Value pairs while others may do less. Because of this, Hadoop uses dynamic scheduling. The purpose of which is to make empty task slots be used for unprocessed data in order to balance the workload on each node. Hadoop also provides task level fault tolerance for scheduling, a feature MPI doesn't support.

Twister and Spark reside somewhere between the range of MPI and Hadoop. Twister is aimed at providing an easy-to-use and data-centric solution to process big data in data mining or scientific applications. Twister makes the control flow as iterations of MapReduce jobs. The output of each MapReduce iteration is collected and sent as the input to the next iteration. The data in Twister is also abstracted as Key-Value pairs for intermediate data regrouping as per needs of the application. Twister uses static scheduling. Data is first pre-split and evenly distributed to compute nodes based on the available computing slots (the number of cores). Tasks are then sent to where the data is located.

Spark also serves for iterative algorithms but boasts more flexible iteration control with separated RDD operations called transformations and actions. Here RDD is another layer of data abstraction higher than Key-Value pairs. A RDD includes a set of Key-Value pairs and describes the distribution of this data in the whole environment. Typical operations on RDDs include not only MapReduce-like operations such as Map, GroupByKey (close to Shuffle but without sort) and ReduceByKey (same as Reduce), but also operations related to relational database such as Union, Join, and Cartesian-Product. Scheduling in Spark is similar to Dryad but with the consideration of the availability in memory of RDD partitions. RDD's lineage graph is examined to build a DAG of stages for late execution.

The data abstraction in MapReduce also requires more work in the form of data partition before data loading. This is because the data abstracted in computation is usually not organized in the same way as the data stored in the file systems. For example, the data in the image clustering application is stored in a set of text files. Each file contains feature vectors generated from a related set of images. The file lengths and the total number of files usually vary. However, in computation we make the number of data partitions to be the same as the number of cores or the multiple of the number of cores so that we can evenly distribute the computation. Ultimately we need to convert "raw" data on disks to "cooked" data ready for computation. Currently we split original data files into evenly sized data partitions. But Hadoop can automatically load data from blocks with self-defined InputSplit or InputFormat class. At the same time, MPI requires user to split data or use special file format HDF5 [19] and NetCDF [20] commonly used in scientific applications.

## 3.2 Collective Communication and Intermediate Data Handling

In the last few decades, MPI researchers made major progress on communication optimization. However as a computation-centric application, MPI focuses on low latency communication while for example our example is notable for large messages where latency less relevant. With the support of high-performance hardware, communication is well optimized. Users can communicate in two ways. One is to call send/receive APIs to customize communication between processes. Another is to invoke libraries to do collective communication operations, which is a type of communication in which all the workers are required to participate.

Often data-centric problems run on clouds which consist of commodity machines, and the cost of transferring big intermediate data is high. For example, in the image clustering application example of this paper, broadcasting in each iteration is needed and the size is about 500MB. Our findings show that this operation and the big data can be a great burden to current data-centric

technology. This makes it necessary to systematically develop a Map-Collective approach with a wide range of collectives and with big data not the MPI big simulation optimizations.

Traditionally, there are 7 collective communication operations discussed in MPI [21]. The first four, broadcast, scatter, gather, and allgather are called "data redistribution operations" [21]. The remaining three, reduce(-to-one), reduce-scatter, all-reduce are called "data consolidation operations" [21]. In "data redistribution operations", neither Hadoop, Twister nor Spark covers all 4 operations. In detail, Hadoop only has "broadcast" with no explicit "scatter" or "gather". Considering that in Hadoop data is managed by HDFS, direct memory-to-memory collective communication does not in fact exist. Twister has "broadcast", "scatter" and "gather". Spark has "broadcast" and "gather". Our Twister4Azure system [22] supports "allgather" and "allreduce" and in a later paper we will describe the integration of these different collectives into a single system that runs interoperably on HPC clusters (Twister) or PaaS cloud systems (Twister4Azure) changing the implementation to optimize performance for each infrastructure. The same high level collective primitive is used on each platform with different under-the-hood optimizations.

Between runtimes, broadcasting data abstraction and methods are very different. In MPI, data is abstracted as an array buffer. In Hadoop it is a file on HDFS. Twister and Spark treat broadcasting data as an object. But in detail, Twister treats the data as a Key-Value pair unlike Spark which treats it as arbitrary objects. Objects are much easier to manipulate compared with files and array buffers.

In MPI, several algorithms are used for broadcasting. MST (Minimum-Spanning Tree) method is a typical broadcasting method used in MPI [21]. In this method, nodes form a minimum spanning tree and data is forwarded along the links. In this way, the number of nodes which have the data grows in geometric progression. Here we use $p$ as the number of nodes, $n$ as the data size, $\alpha$ as communication startup time and $\beta$ as data transfer time per unit. The performance model can be described by the formula below:

$$T_{MST}(p,n) = \lceil log_2 p \rceil (\alpha + n\beta) \quad (1)$$

This method is much better than simple broadcasting by changing the complexity term $p$ to $\lceil log_2 p \rceil$. But it is still insufficient when compared with scatter-allgather bucket algorithm. This algorithm is used in MPI for long vectors broadcasting which follows the style of "divide, distribute and gather" [23]. In "scatter" phase, it scatters the data to all the nodes. Then in "allgather" phase, it does bucket algorithm. This method views the nodes as a chain. At each step, every node sends data to its right neighbor [25]. By taking advantage of the fact that messages traversing a link in opposite direction do not conflict, "allgather" is done in parallel without any network contention. The performance model can be established as follow:

$$T_{bucket}(p,n) = p(\alpha + n\beta/p) + (p-1)(\alpha + n\beta/p) \quad (2)$$

In large data broadcasting, assuming α is small, the broadcasting time is about $2n\beta$. This is much better than the MST method because the time appears constant. However, it is not easy to set global barrier between "scatter" and "allgather" phases in cloud system to enable all the nodes to do "allgather" at the same global time through software control. As a result, some links will have more load than the others and thus we arrive at network contention. We implement this algorithm and provide the test results on IU PolarGrid (See Table 3). The execution time is roughly kept at $2n\beta$ level. But as the number of nodes increase, it gets slightly slower.

There is also the InfiniBand [24] multicast based broadcasting method in MPI [25]. Currently many clusters support hardware-based multicast. But it is not a reliable method, the sending order is not guaranteed and the package size of each sending is limited. So after the first stage of multicasting, broadcasting is enhanced with a chain-like broadcasting, which is reliable enough to make sure every process has completed data receiving. In the second stage, the nodes are formed into a virtual ring topology. Each MPI process that gets the message via multicast serves as a new "root" within the virtual ring topology and exchange data to the predecessor and successor in the ring. This is similar to the bucket algorithm we discuss above.

**Table 3. Scatter-allgather bucket algorithm performance on IU PolarGrid with 1 GB data broadcasting**

| **Node#** | 1 | | 25 | 50 | 75 | 100 | 125 |
|---|---|---|---|---|---|---|---|
| **Seconds** | 11.4 | | 20.57 | 20.62 | 20.68 | 20.79 | 21.2 |

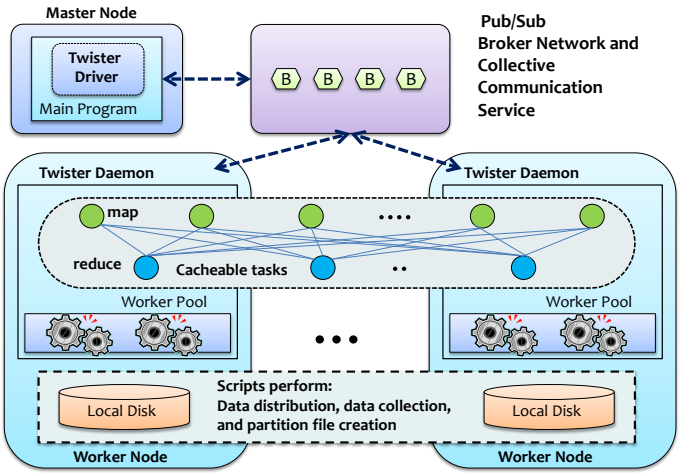


**Figure 4. Initial Twister architecture with brokers as main communication components**

Though the methods heretofore reviewed are not perfect, they all can reduce broadcasting time to a great extent. Still, none of them are applied in data-centric solutions. However, simple algorithm is commonly used. Hadoop system relies on HDFS to do broadcasting. A component named Distributed Cache is used to cache data from HDFS to local disk of compute nodes. The API addCacheFile and getLocalCacheFiles work together to complete the process of broadcasting. There is no special optimization. The data downloading speed depends on the number of replicas in HDFS [18]. This method generates significant overhead (a factor of $p$) when handling big data broadcasting. This will be shown in later experiments.

We call this "simple algorithm" because it basically sends data to all the nodes one by one. Initially in Twister, a single message broker is used to do broadcasting in a similar way (See Figure 4). Though using multiple brokers in Twister or using multiple replicas in HDFS could contain a simple 2-level broadcasting tree

and ease the performance issue, they won't fundamentally address the problem. As a result, to replace the current broadcasting in Twister, in the next section, we propose a chain-based broadcasting algorithm suitable for cloud systems.

Meanwhile, other than using simple algorithm, Spark adds

**Table 4. Broadcasting programming interface**

| Runtime | Broadcasting Interface |
|---|---|
| MPI | MPI_Bcast(bcast_data, total_num_data, MPI_CHAR, 0, MPI_COMM_WORLD ); |
| Twister | driver.addToMemCache(bcastData); |
| Spark | val barr1 = sparkContext.broadcast(arr1)<br>sparkContext.parallelize(1 to 10, slices).foreach<br>{<br>i => println(barr1.value.size)<br>} |

BitTorrent [26] to enhance broadcasting speed. BitTorrent is a well-known technology in internet file sharing. The programming interface of broadcasting in Spark is very different from MPI and Twister. Due to the mechanism of late execution, broadcasting is not finished in a single step but in two stages. When broadcasting is invoked, the data is not broadcast until the parallel tasks are executed (See Table 4). The code in Table 4 is from Spark example code. Broadcasting happens when 10 printing tasks are invoked. So broadcasting doesn't execute on all the nodes but only on the nodes where tasks are located. The performance of Spark Broadcasting is discussed with a simple case in Section 6.6.

For data consolidation operations, "reduce(-to-one)" and "reduce-scatter" are parallel to a "shuffle-reduce" operation in data-centric solutions. "Reduce-(to-one)" can be viewed as using shuffling with only one Reducer while "reduce-scatter" can be viewed as using shuffling with all workers as reducers. However, these operations are fundamentally different in terms of semantics because "shuffle-reduce" is based on Key-Value pairs while "reduce-(to-one)" and "reduce-scatter" are based on vectors. The data abstraction of the former is more flexible than the latter. In "shuffle-reduce" the number of keys in one worker can be arbitrary. For example, in word count, for a particular word we shall call "word1", one worker could generate multiple Key-Value pairs with this "word1" as key and count "1" as the value. Alternatively there might even be no such Key-Value pairs if the work couldn't find any examples of "word1". Furthermore, a value can be any arbitrary object which encapsulates many different data types. However, "reduce-scatter" requires the size of the vectors for reduction to be identical in all workers. Because the number of words and counts in each worker is hard to estimate, it is difficult to replace "shuffle-reduce" to "reduce-scatter" in word count.

**Table 5. MPI Shuffling Pseudo Code**

**Algorithm 1 MPI shuffling**

```
for( i←0; i<max_rank; i←i+1) {
  if(my_rank = i) {
    for(j←0; j<max_rank&&j!=i; j←j+1)
      MPI_Send(numSendKVpairs[j]);
      for(k←0; k<numSendKVpairs[j]; k←k+1)
        MPI_Send(sendKVpairs[j][k])
  else
    MPI_Recv(numRecvKVpairs[i]);
    for(j←0; j<numRecvKVpairs[j]; j←j+1)
      MPI_Recv(recvKVpairs[i][j]);
```

To simulate "shuffle-reduce" in MPI, we cannot use collective communication in MPI directly. Instead we have to customize the communication with send/receive calls. The following pseudo code represents how shuffling may look based on send/receive APIs (See Table 5). We simplify the code by using a matrix to hold all the Key-Value pairs for send/receive but from the code we still can see another weakness of MPI in shuffling: the program is not simple and users have to explicitly designate where the data goes. By contrast, in data-centric solutions, data is managed by the framework, and automatically goes to the destination according to their keys.

A a result, shuffling can be viewed as a unique collective communication in data-centric solutions. The implementation is also different between runtimes. Hadoop manages intermediate data on disk, so data is first partitioned, sorted and spilled to disk, then transferred, merged and sorted again at Reducer side. However, shuffling in Twister is much simpler than it is in Hadoop. Data is only regrouped by keys and transferred in memory and there is no sorting [4]. So shuffling in Twister has much better performance than in Hadoop. Though it is well optimized, it is still not scalable in handling large intermediate data. Then we use local aggregation across Map threads at Map stage. Since each worker in Twister runs on the thread level and data generated by each worker can be shared, we are able to optimally shrink the intermediate data size on each compute node and accelerate shuffling.

In Spark, there are two APIs related to shuffling. One is "groupByKey", and another is "sort". Remembering that "shuffle" in Hadoop includes "regroup" and "sort". Since "shuffle" in Twister only contains "regroup", it seems that "shuffle" operation is not well defined. So is "sort" necessary in "shuffle"? The answer is no. Firstly, in Twister, all the intermediate data is managed in memory so that keys can be regrouped through a large hash map. But for Hadoop, since merging is done on disk, sorting becomes a required step to put keys with the same hash code together. Secondly, many applications such as word count and image clustering applications mentioned above, it is sufficient that the data is regrouped without being sorted. The ranking of each key is not important to the application. As a result, we view "shuffle" as only "regroup".

In summary, we notice that collective communication is not well studied in the context of MapReduce and data-centric solutions. Furthermore it may not be optimally implemented in the current runtimes. Though collective communication operations have been used in MPI for decades, they are still missing in MapReduce despite still being required by the applications. In the image clustering application, "broadcast" and "shuffle" are two important operations involved. With optimization, we introduce new Twister control flow with optimized broadcasting and local aggregation feature (See Figure 2).

We note that our collectives are implemented asynchronously but the broadcast step of Kmeans naturally synchronizes the algorithm at each iteration

# 3. BROADCAST COLLECTIVE COMMUNICATON

To address the need for high performance broadcasting in the image clustering application, we replace the original broker methods in Twister with a new chain method based on TCP sockets to provide customized control of the message routing in broadcasting.

## 4.1 Chain Broadcasting Algorithm

Here we propose chain method, an algorithm based on pipelined broadcasting [28]. In this method, compute nodes in Fat-Tree topology [29] are treated as a linear array and data is forwarded from one node to its neighbor chunk-by-chunk. Performance is enhanced by dividing the data into many small chunks and overlapping the transmission of data on nodes. For example, the first node would send a data chunk to the second node. Then, while the second node sends the data to the third node, the first node would send another data chunk to the second node, and so on and so forth [28]. This kind of pipelined data forwarding is called "a chain". It is particularly suitable for the large data sizes in our communication problem.

The performance of pipelined broadcasting depends on the selection of chunk size. In an ideal case, if every transfer can be overlapped seamlessly, the theoretical performance is as follows:

$$T_{Pipeline}(p,k,n) = p(\alpha + n\beta/k) + (k-1)(\alpha + n\beta/k) \quad (3)$$

Here $p$ is the number of nodes, $k$ is the number of data chunks, $n$ is the data size, $\alpha$ is communication startup time and $\beta$ is data transfer time per unit. In large data broadcasting, assuming $\alpha$ is small and $k$ is large, the main term of the formula is $(p+k-1)n\beta/k \approx n\beta$ which is close to constant. From the formula, the best number of chunks is $k_{opt} = \sqrt{(p-1)n\beta/\alpha}$ when $\partial T/\partial k = 0$ [28]. However, in practice, the actual chunk size per sending is decided by the system and the speed of data transfers on each link could vary as network congestion might occur when data is continuously forwarded into the pipeline. As a result, formula (3) cannot be applied directly to predict real performance of our chain broadcasting implementation. But the experiment results we will present later still show that as $p$ increases, the broadcasting time remains constant and close to the bandwidth limit.

## 4.2 Rack-Awareness

This chain method is suitable for racks of machines with Fat-Tree topology connection, which is a commonly used network topology in clusters or in data centers [30]. Since each node only has two links, which is less than the number of links per node in Mesh/Torus [31] topology, chain broadcasting can maximize the utilization of the links per node. We also make the chain topology-aware by allocating nodes within the same rack nearby in the chain. Assuming the racks are numbered as $R_1$, $R_2$ and $R_3$…, the nodes in $R_1$ are put at the beginning of the chain, then the nodes in $R_2$ follow the nodes in $R_1$, and then nodes in $R_3$ follow nodes in $R_2$, etc. Otherwise, if the nodes in $R_1$ are intertwined with nodes in $R_2$ in the chain sequence, the chain flow will jump between switches, which overburdens the core switch.

To support rack-awareness, as seen in Hadoop, we write and save configuration information on each node. Each node can discover its predecessor and successor by loading this information when starting. In the future, we are also looking into supporting automatic topology detection to replace the static specification of topology information.

## 4.3 Buffer Management

Another important factor affecting broadcasting speed is buffer management. The cost of buffer allocation and data copying between buffers is not included in formula (3). There are 2 levels of buffers used in data transmission. The first level is the system buffer and the second level is the application buffer. System buffer is used by TCP socket to hold the partial data transmitted from the network. The application buffer is created by the user to integrate the data from the socket buffer. Usually the socket buffer size is much smaller than the application buffer size. The default buffer size setting of Java socket object in IU PolarGrid is 128KB while the application buffer we chose for broadcasting is the total size of the data required to be broadcasted.

We observed performance degradation caused by buffer usage. One issue is that if the socket buffer is smaller than 128 KB, the broadcasting performance can be slowed down due to the TCP window being unable to open up fully, which results in throttling of the sender. Further large-sized user buffer allocation during the pipeline forwarding can also slightly slow-down the overall performance. To make a clean comparison with MPI, which does buffer initialization before broadcasting, we initialize a pool of free buffers once the receiver program starts instead of allocating buffers during the broadcasting.

## 4.4 Object Serialization and De-serialization

In memory-to-memory broadcasting, data is stored as an object in memory. So we need to serialize the object to byte array before broadcasting and de-serialize byte array back to an object afterwards. We observe that object serialization and de-serialization can be slow for large data sizes. As a result, the serialization speed depends on the data type. Our experiments show that serializing 1 GB "double" data is much faster than serializing 1 GB "byte" data. Moreover, de-serializing 1 GB "byte" data demands even greater time than serializing it. Since it is local operation and can be optimized at a cost in portability, we measure these overheads and separate them from the core broadcasting operation.

## 4.5 Fault Tolerance

Communication fault tolerance intrinsic to Collective, should be considered in chain broadcasting. When large data is transmitted among a vast number of nodes, communication failures become likely. Several strategies are applied here in our approach. Firstly if there are failures in establishing connection from node-to-node, a retry is issued. Alternatively one can try other destinations. Secondly, if the chain is seriously broken the whole broadcasting will restart. Finally, at the end of broadcasting, the root waits and checks if all the nodes have received all the data blocks. If the root doesn't get the ACK from the last node in the chain within a time window, it restarts the whole broadcasting.

## 4.6 Implementation

We implement the chain broadcasting algorithm in the following way: it starts with a request from the root to the first node in the topology-aware chain sequence. Then the root keeps sending a small portion of the data to the next node. In the meantime, for the nodes in the chain, each node creates a connection to the successor node in the chain. Next each node receives a partial data from the socket stream, store it into the application buffer and forward it to the next node (See Table 6).

**Table 6. Broadcasting algorithm**

| **Algorithm 1 root side "send" method** |
|---|
| nodeID ← 0 |
| connection ← connectToNextNode(nodeID) |
| dout ← connection.getDataOutputStream() |
| bytes ← byte array serialized from the broadcasting object |
| totalBytes ← total size of bytes |

```
SEND_UNIT ← 8192
start ← 0

dout.write(totalBytes)
while (start + SEND_UNIT < totalBytes)
   dout.write(bytes, start, SEND_UNIT)
   start ← start + SEND_UNIT
   dout.flush()
if (start < totalBytes)
   dout.write(bytes, start, totalBytes - start)
   dout.flush()
waitForCompletion()
```

**Algorithm 2 Compute node side "receive" method**

```
connection ← serverSocket.accept()
dout ← connection.getDataOutputStream()
din ← connection.getDataInputStream()
nodeID ← this.nodeID + 1
connNextN ← connectToNextNode(nodeID)
doutNextN ←connToNextN.getDataOutputStream()
dinNextN ← connToNextN.getDataInputStream()

totalBytes ← din.readInt()
doutNextN.writeInt(totalBytes)
doutNextN.flush()
bytesBuffer ← getFromBufferPool(totalBytes)
RECV_UNIT ← 8192
recvLen ← 0
while ((len ← din.read(bytesBuffer, recvLen, RECV_UNIT)) > 0)
   doutNextN.write(bytesBuffer, recvLen, len)
   doutNextN.flush()
   recvLen ← recvLen + len
   if (recvLen = totalBytes) break
notifyForCompletion()
```

# 4. LOCAL AGGREGATION IN MAP STAGE

We already discussed the difference between shuffling in Twister and other runtimes in Section 3.2. Based on the facts presented in Section 2, the performance of shuffling depends on the size of intermediate data. Since the data transferred is very large and the number of links available for data transmission is limited, the cost of shuffling is very high and the whole process is unstable.

Some solutions try to use Weighted Shuffle Scheduling (WSS) [27] to balance the data transfers by using the data size to determine scheduling. However this strategy will not help for this image clustering application, because the data size generated for each Map task is the same.

We reduce the intermediate data size by using local aggregation across Map tasks in Map stage. To support local aggregation, we provide appropriate interface to help users define the aggregation operation.

We notice that each Key-Value pair in intermediate data is a partial sum of the components of data points associated with a particular cluster. Since addition is an operation with both commutative and associative properties, for any two values belonging to the same key, we can do addition on them and merge them to a single Key-Value pair, which has no effect on the final result. This property can be illustrated by the following formula:

$$f(kv_1, \cdots, kv_i, \cdots, kv_j, \cdots, kv_n) = f(kv_1, \cdots, (kv_i \oplus kv_j), \cdots, kv_n) = f(kv_1, \cdots, (kv_j \oplus kv_i), \cdots, kv_n) \; \forall \, i, j, 1 \le i, j \le n \qquad (4)$$

Here $\oplus$ represents a set of operations which are similar to addition operation that can be applied on any two Key-Value pairs. This will then generate a new Key-Value pair by operating, $f$ is the Reduce function and $n$ is the number of Key-Value pairs belonging to the same key. In our image clustering application, $\oplus$ is the addition of two partial sums. In other applications, we can also find an appropriate operator. In Word Count [2], $\oplus$ is the addition of two partial counts of the same word and can be operations other than addition, such as multiplication and max/min value selection, or just simple logical combination of the two values.

With $\oplus$ operation and also noting that Map tasks work at thread level on compute nodes, we do local aggregation in the memory shared by Map tasks. Once a Map task is finished, it doesn't send data out immediately but instead caches the data to a shared memory pool. When the key conflict happens, the program invokes a 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 in a node are finished. By trading communication time with computation time, the data necessary to be transferred can be significantly reduced.

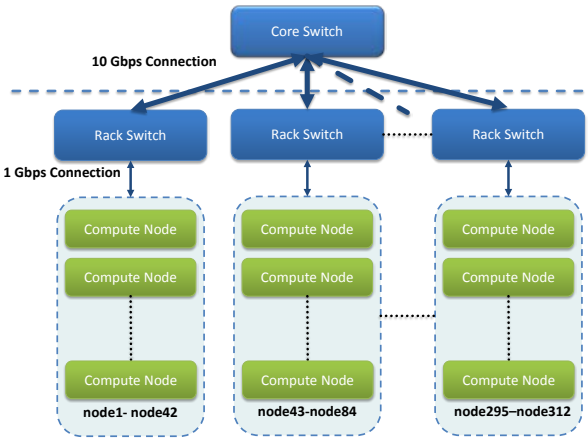


**Figure 5. Fat-Tree topology in IU PolarGrid**

# 5. Experiments

To evaluate performance of the new proposed broadcasting method and local aggregation mechanism, we conducted experiments about broadcasting and shuffling on IU PolarGrid in the context of both kernel and application benchmarking. The results demonstrate that chain method achieves the best performance on big data broadcasting compared with the other MapReduce and MPI methods. In addition, shuffling with local aggregation can out-perform the original shuffling significantly.

## 6.1 IU PolarGrid Cluster

IU PolarGrid cluster [11] uses a Fat-Tree topology to connect compute nodes. The nodes are split into sections of 42 nodes which are then tied together with 10 GigE to a Cisco Nexus core switch. For each section, nodes are connected with 1 GigE to an IBM System Networking Rack Switch G8000. This forms a 2-level Fat-Tree structure with the first level of 10 GigE connections

and the second level of 1 GigE connections (See Figure 5). 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. Each compute node has 16 GB total memory.

The bottleneck of this topology is that inter-switch communication is through the one and only core switch and the connection is limited to 10 GigE. As a result, reducing the number of inter-switch communication times is considered the highest priority in design of efficient collective communication algorithms for a fat-tree topology.

## 6.2 Broadcasting

We test the following methods: chain method in Twister, MPI_BCAST in Open MPI 1.4.1 [32], and the broadcasting method in MPJ Express 0.38 [33]. We also compare the current Twister chain broadcasting method with other designs such as chain method without topology awareness and simple broadcasting as a means to show the efficiency of the new method.

We measure the broadcasting time from the start of calling the broadcasting method to the end of the calling return. We test the performance of broadcasting from a small scale to a medium large scale. The range includes 1 node, 25 nodes with 1 switch, 50 nodes under 2 switches, 75 nodes with 3 switches, 100 nodes with 4 switches, 125 nodes with 5 switches, and 150 nodes with 5 switches. The tests are for different data size, including 0.5 GB (500MB), 1 GB, and 2 GB. Each result is the average of 10 executions. There are only milliseconds of differences between execution times therefore we omit the error in the following charts.

Figure 6 shows the comparison between chain method and MPI_BCAST method in Open MPI. The time cost of the new chain method is stable as the number of processes increases. This matches the broadcasting formula (3) and we can conclude that with proper implementation, the actual performance of the chain method can achieve near constant execution time. Besides, the new method achieves 20% better performance than MPI_BCAST in Open MPI.

Figure 7 shows the comparison between Twister chain method and broadcasting method in MPJ. Due to exceptions, we couldn't launch MPJ broadcasting on 2GB data. So we draw a dashed line to mark the prediction. Since 1GB MPJ broadcasting uses twice the time of 0.5GB MPJ broadcasting, we assume 2 GB MPJ broadcasting also costs double the time of 1 GB MPJ broadcasting. MPJ broadcasting method is also stable as the number of processes grows, but is four times slower than our Java

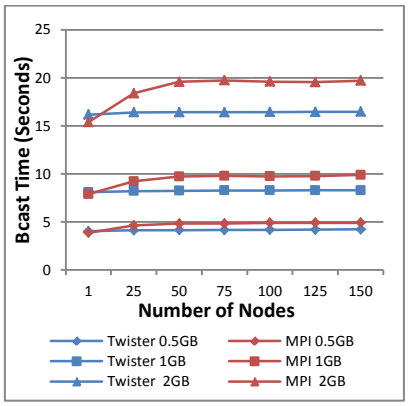


**Figure 6. Performance comparison of Twister chain method and Open MPI MPI_Bcast**

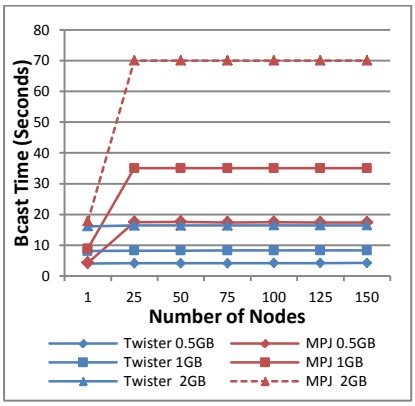


**Figure 7. Performance comparison of Twister chain method and MPJ broadcasting method (MPJ 2GB is prediction only)**

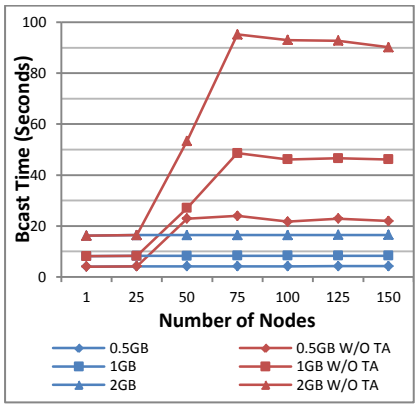


**Figure 8. Chain method with/without topology-awareness**

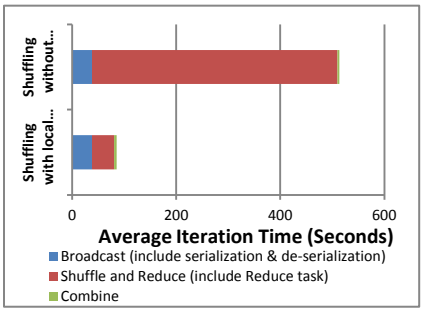


**Figure 9. Comparison between shuffling with and without local aggregation**

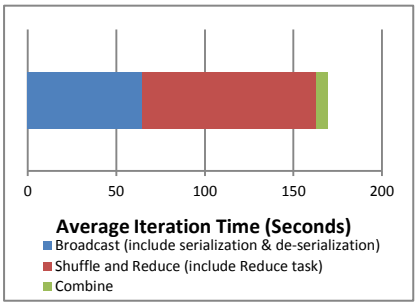


**Figure 10. Communication cost per iteration of the image clustering application**

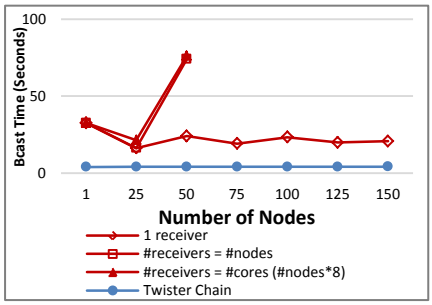


**Figure 11. Spark broadcasting performance of 500MB data (Twister Chain results are also provided as reference)**

implementation. Further there is a significant gap between 1-node broadcasting and 25-node broadcasting in MPJ.

However if the chain sequence is randomly generated but not topology-aware, the performance degrades quickly as the scale grows. Figure 8 shows that chain method with topology-awareness is 5 times faster than that of the chain method without topology-awareness. For broadcasting within a single switch, we see that as expected, there is not much difference between the two methods. However, as the number of nodes and the number of racks increase, the execution time increases significantly. When there are more than 3 switches, the execution time become stable and doesn't change much. Because there are many inter-switch communications, the performance is constrained by the 10 Gb bandwidth and the throughput ability of the core switch.

In Table 7, we show the performance of simple broadcasting and compare it with Twister chain method. Since simple broadcasting takes a great deal of time, we won't present a chart here. The purpose is to show the baseline of broadcasting performance in IU PolarGrid. Owing to 1 Gb connection on each node, we see the transmission speed is about 8 seconds per GB which matches the setting of the bandwidth. With our new algorithm, we successfully reduce the cost by about a factor of $p$ from $O(pn)$ to $O(n)$. Here $p$ is the number of compute nodes and $n$ is data size.

**Table 7. Performance comparison between chain broadcasting and simple broadcasting (in seconds)**

| Node# | Twister Chain | | | Simple Broadcasting | | |
|---|---|---|---|---|---|---|
| | 0.5 GB | 1 GB | 2 GB | 0.5 GB | 1 GB | 2 GB |
| **1** | 4.04 | 8.09 | 16.17 | 4.04 | 8.08 | 16.16 |
| **25** | 4.13 | 8.22 | 16.4 | 101 | 202 | 441.64 |
| **50** | 4.15 | 8.24 | 16.42 | 202.01 | 404.04 | 882.63 |
| **75** | 4.16 | 8.28 | 16.43 | 303.04 | 606.09 | 1325.63 |
| **100** | 4.18 | 8.28 | 16.44 | 404.08 | 808.21 | 1765.46 |
| **125** | 4.2 | 8.29 | 16.46 | 505.14 | 1010.71 | 2021.3 |
| **150** | 4.23 | 8.3 | 16.48 | 606.14 | 1212.21 | 2648.6 |

**Table 8. Chain method performance under different socket buffer sizes**

| Buffer Size (KB) | 8 | 16 | 32 | 64 |
|---|---|---|---|---|
| Time (seconds) | 65.5 | 45.46 | 17.77 | 10.8 |
| Buffer Size (KB) | 128 | 256 | 512 | 1024 |
| Time (seconds) | 8.29 | 8.27 | 8.27 | 8.27 |

By looking inside the chain method, we also examine the potential impact from socket buffer size. As we mentioned in Section 4.3, a small socket buffer could cause slow-down of the sender. We take broadcasting 1 GB data on 125 nodes as an example and increase the socket buffer size gradually from 8KB to 1MB. We find that when buffer size is 8 KB, the performance is terrible, then as the buffer size grows the time cost gets lower. When the buffer size is larger than 128 KB, we get the best performance and stable execution time. The experiment shows that the socket buffer size can affect the performance greatly because the TCP window cannot open up fully when the buffer size is small. With a proper buffer size, the broadcasting performance can be improved by almost an order of magnitude from small to large buffer sizes (see Table 8).

## 6.3 Shuffling and Local Aggregation

To benchmark the performance of shuffling using local aggregation, we choose the following settings to run the image clustering application. For job settings, we choose 125 nodes to run the application with 1000 Map tasks (each node with 8 Map tasks) and 125 reduce tasks (each node with 1 Reduce task). For data settings, we restrict the number of centroids to 500K and focus on testing the performance of collective communication. Since 500K centroids can generate about 1 GB of intermediate data per task, the overhead from shuffling is significant. We measure the total time from the start of shuffling to the end of the Reduce phase noting that reducers start asynchronously (a reducer starts once it gets all the data). Time costs on Reduce tasks are included but on average it is just around 1 second and is negligible compared with the data transfer time.

Figure 9 shows the time difference of shuffling with or without local aggregation in Map stage in the settings above. Without using local aggregation, the output per node is 8 GB and the total data for shuffling is about 1 TB. After using local aggregation, the output per node is reduced to 1 GB and the total data for shuffling is only about 125 GB and the time cost on shuffling is only 10% of the original time; an improvement from about 8 minutes to only 40 seconds. To reduce intermediate data from 1 TB data to 125 GB data, we only need an extra 20 seconds in local aggregation.

## 6.4 Image Clustering Application

Finally we present a full execution of the image clustering application here. We successfully cluster 7.42 million vectors into 1 million cluster centers. We create 10000 map tasks on 125 nodes. Each node has 80 tasks. Each task caches 742 vectors. For 1 million centroids, broadcasting data size is about 512 MB. Shuffling data is 20 TB, while the data size after local aggregation is about 250 GB. Since the total memory size on 125 nodes is 2 TB, we even cannot execute the program unless local aggregation is performed. Figure 10 presents the collective communication cost per iteration, which is 169 seconds (less than 3 minutes). Note that we are currently in development of a new faster Kmeans algorithm [8][9] that will drastically reduce the current hour-long computation time in Map stage by up to a factor of factor of *l* (the dimension which is currently 512 to 2048) and so the improved communication time is highly relevant.

## 6.5 Analysis of Spark Broadcasting

Here we look at the performance of the BitTorrent broadcasting in Spark, which is reported as a method which has excellent performance [27]. In our testing however, the current Spark version 0.7.0 shows that the performance is good in a small number of nodes but degrades quickly as the number of nodes increases (See Figure 11). Because broadcasting is related to nodes as well as tasks, we designed the following experiments. We start with testing on 500MB data broadcasting. Firstly, we execute only 1 task after invoking broadcasting. The result is stable as the number of nodes grows. However, when we set the number of the receivers equal to the number of nodes, performance issues emerge. On 25 nodes with 25 tasks, the performance is still same as with 1 receiver. But on 50 nodes with 50 tasks, broadcasting time increases three-fold. We also try to execute broadcasting from 75 nodes to 150 nodes, but none of the tests are executed successfully. The third test we have is to increase the number of receivers to the number of cores. The result is similar. So broadcasting in Spark can only scale to 50 nodes in our tests. We also try 1 GB and 2 GB broadcasting but these cases do not scale to 25 nodes.

Since broadcasting topology in BitTorrent is built dynamically, It is unknown if the broadcasting topology follows the patterns in MPI broadcasting such as minimum spanning tree.

Also important in broadcasting topology is that this topology follows rack-awareness. A special dynamic topology detection technique is mentioned [27] but from the experiments it may not be applied to the current version. For chunk size in sending, it is mentioned in [27] 4 MB is good for performance but without any further analysis. In Scatter-allgather bucket algorithms, data is also split based on the number of the receivers.

# 6. RELATED WORK

In Section 3 we discussed the runtime of several data processing tools and compared the collective communication within them. Here we summarize the analysis and add other observations. Collective communication algorithms are well studied in MPI runtime although the Java implementations are less well optimized. Each communication operation has several different algorithms based on message size and network topology such as linear array, mesh and hypercube [21]. Basic algorithms are pipeline broadcast method [28], minimum-spanning tree method, bidirectional exchange algorithm, and bucket algorithm [21]. Since these algorithms have different advantages, algorithm combination (polymorphism) is widely used to improve the communication performance [21]. Furthermore some solutions also provide auto algorithm selection [34].

Other papers have a different focus than our work. Some of them only study small data transfers up to megabytes level [21] [35] while some solutions rely on special hardware support [23]. The data type is typically vectors and arrays whereas we are considering objects. Many algorithms such as "allgather" operate under the assumption that each node has the same amount of data [21] [23], which is uncommon in a MapReduce model. As a result, although shuffling can be viewed as a Reduce-Scatter operation, its algorithm cannot be applied directly on shuffling since the data amount generated by each Map task is unbalanced in most MapReduce applications.

There are several solutions to improve the performance of data transfers in MapReduce. Orchestra [27] is one such global control service and architecture to manage intra- and inter-transfer activities in the Spark system, where we gave some test results in section 3.1. It not only provides control, scheduling and monitoring on data transfers, but also optimization on broadcasting and shuffling. For broadcasting, it uses an optimized BitTorrent-like protocol called Cornet, augmented by topology detection. For shuffling, Orchestra employs weighted shuffle scheduling (WSS) to set the weight of the flow as proportional to the data size; we noted earlier this optimization is not relevant in our application.

Hadoop-A [36] provides a pipeline to overlap the shuffle, merge and reduce phases and uses an alternative Infiniband RDMA based protocol to leverage RDMA inter-connects for fast data shuffling. MATE-EC2 [37] is a MapReduce like framework for EC2 [38] and S3 [39]. For shuffling, it uses local aggregation and global aggregation. This strategy is similar to what we did in Twister but as it focuses on EC2 cloud environment, the design and implementation are totally different. iMapReduce [40] and iHadoop [41] are iterative Mapreduce frameworks that optimize the data transfers between iterations asynchronously, where there exists no barrier between two iterations. However, this design 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.

Microsoft Daytona [45] is a recently announced iterative MapReduce runtime developed by Microsoft Research for Microsoft Azure Cloud Platform that builds on some of the ideas of the earlier Twister system. Currently Excel DataScope is presented as an application of Daytona. Users can upload data in their Excel spreadsheet to the DataScope service or select a dataset already in the cloud, then select an analysis model from the Excel DataScope research ribbon to run against the selected data. The results can be returned to the Excel client or remain in the cloud for further processing and visualization. Daytona is available as a "Community Technology Preview" for academic and non-commercial use.

# 7. CONCLUSIONS AND FUTURE WORK

In this paper, we have demonstrated the first steps towards a high performance Map-Collective programming model and runtime using the requirements of a large scale clustering algorithm. We replaced broker-based methods and designed and implemented a new topology-aware chain broadcasting algorithm. Compared with the simple broadcast algorithm, the new algorithm reduces the time burden of broadcasting by at least a factor of 120 over 125 nodes. It gives 20% better performance than best C/C++ MPI methods (and four times faster than Java MPJ) and a factor of 5 improvements over non-optimized (for topology) pipeline-based method over 150 nodes. The shuffling cost after using local aggregation is only 10% of the original time. In particular, collective communication has significantly improved the intermediate data transfer for large scale image clustering problems.

In future work, we will improve the Kmeans algorithm [8][9][42] and apply the Map-Collective framework to other iterative applications [43] including Multi-Dimensional Scaling where the allgather primitive is needed. We will also extend current work to include an allreduce collective that is an alternative approach to Kmeans. The resultant Map-Collective model that captures the full range of traditional MapReduce and MPI features will be evaluated on Azure [22] as well as IaaS/HPC environments. We will integrate Twister with Infiniband RDMA based protocol and compare various communication scenarios. Initial observation suggests a different performance profile from that of the Ethernet network evaluated here. Furthermore we will integrate topology and link speed detection services and utilize services such as ZooKeeper [44] to provide coordination and fault detection.

# 8. ACKNOWLEDGEMENT

The authors would like to thank Prof. David Crandall at Indiana University for providing the social image data. This work is in part supported by National Science Foundation Grant OCI-1149432

# 9. 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] Jaliya.Ekanayake, Hui Li, Bingjing Zhang, Thilina Gunarathne, Seung-Hee Bae, Judy Qiu, Geoffrey Fox. 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] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster Computing with Working Sets. In HotCloud, 2010.
[6] 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.
[7] J. Dean, G. Corrado, R. Monga, K. Chen, M. Devin, Q. Le, M. Mao, M. Ranzato, A. Senior, P. Tucker, K. Yang, A. Ng, Large Scale Distributed Deep Networks, in proceedings of NIPS 2012: Neural Information Processing Systems Conference.
[8] Judy Qiu, Bingjing Zhang, "Mammoth Data in the Cloud: Clustering Social Images", to appear in the book on "Clouds, Grids and Big Data" to be published in the series "Advances in Parallel Computing" by IOS Press publishers, 2013. Book Editors: Charlie Catlett, Wolfgang Gentzsch, Lucio Grandinetti, Gerhard Joubert, and Jose Vasquez-Polett
[9] Charles Elkan, Using the triangle inequality to accelerate k-means, in TWENTIETH INTERNATIONAL CONFERENCE ON MACHINE LEARNING, Tom Fawcett and Nina Mishra, Editors. August 21-24, 2003. Washington DC. pages. 147-153.
[10] MPI Forum, “MPI: A Message Passing Interface,” in Proceedings of Supercomputing, 1993.
[11] PolarGrid. http://polargrid.org/.
[12] N. Dalal, B. Triggs. Histograms of Oriented Gradients for Human Detection. CVPR. 2005
[13] 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.
[14] Jaliya Ekanayake, Thilina Gunarathne, Judy Qiu, Geoffrey Fox, Scott Beason, Jong Youl Choi, Yang Ruan, Seung-Hee Bae, and Hui Li, Applicability of DryadLINQ to Scientific Applications. January 30, 2010, Community Grids Laboratory, Indiana University.
[15] S. Plimpton, K. Devine, MapReduce in MPI for Large-scale Graph Algorithms, Parallel Computing, 2011
[16] X. Lu, B. Wang, L. Zha, and Z. Xu, Can MPI Benefit Hadoop and MapReduce Applications? International Conference on Parallel Processing Workshops, 2011
[17] T. Hoefler, A. Lumsdaine1, J. Dongarra, Towards Efficient MapReduce Using MPI, Proceedings of the 16th European PVM/MPI Users' Group Meeting, 2009
[18] 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
[19] HDF5, http://www.hdfgroup.org/HDF5/whatishdf5.html
[20] NetCDF, http://www.unidata.ucar.edu/software/netcdf/
[21] 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.
[22] Thilina Gunarathne, Bingjing Zhang, Tak-