Mammoth Data in the Cloud: Clustering Social Images

Judy Qiu
xqiu@cs.indiana.edu

Bingjing Zhang
zhangbj@cs.indiana.edu

Indiana University Bloomington

Abstract— Social image datasets have grown to dramatic size with images classified in vector spaces with high dimension (512-2048) and with potentially billions of images and corresponding classification vectors. We study the challenging problem of clustering such sets into millions of clusters using Iterative MapReduce. We introduce a new Kmeans algorithm in the Map phase which can tackle the challenge of large cluster and dimension size. Further we stress that the necessary parallelism of such data intensive problems are dominated by particular collective operations which are common to MPI and MapReduce and study different collective implementations, which enable cloud-HPC cluster interoperability. Extensive performance results are presented.

KeyWords. Social Images; High Dimension; Fast Kmeans Algorithm; Collective Communication; Iterative MapReduce

Introduction

The rate of data generation has now exceeded the growth of computational power predicted by Moore's law. Challenges from computation are related to mining and analysis of these massive data sources for the translation of large-scale data into knowledge-based innovation. This requires innovative algorithms and core technologies in scalable parallel platforms. However, many existing analysis tools are not capable of handling such big data sets.

Intel's RMS (Recognition, Mining and Synthesis) taxonomy [1] identifies iterative solvers and basic matrix primitives as the common computing kernels for computer vision, rendering, physical simulation, (financial) analysis and data mining applications. These observations suggest that iterative MapReduce will be a runtime important to a spectrum of e-Science or e-Research applications as the kernel framework for large scale data processing.

Classic MapReduce [2] and Hadoop [3] frameworks cannot meet the requirement of executing iterative algorithms due to the inefficiency of repetitive disk access for fetching and merging data over iterations. Several new frameworks designed for iterative MapReduce are proposed to solve this problem, including Twister [4] and HaLoop [5]. Twister, developed by our group, is an iterative MapReduce framework. The early version of Twister targets optimizing data flow and reducing data transfer between iterations by caching invariant data in the local memory of compute nodes. The scheduling mechanism assigns tasks to the node where corresponding invariant

data is located. However, there are other performance issues in iterative algorithms execution not addressed in Twister. We observe that collective communication is missing in current MapReduce frameworks and is essential in many iterative algorithms. 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" [6] in a recent paper on deep learning. We explore high performance broadcasting and shuffling and add them to Twister iterative MapReduce framework. We generalize the MapReduce concept to Map-Collective noting that large collectives are a distinctive feature of data intensive and data mining applications.

In this paper, we introduce a fast Kmeans algorithm that drastically reduces the computation time for data mining in high dimensional social image data. We propose a pipeline-based method with topology awareness to accelerate broadcasting and demonstrate that it outperforms traditional MPI methods [7]. We use local reduction before shuffling to improve performance, which can reduce intermediate data by num_nodes/num_maps×100%. These methods provide important capabilities to our new iterative MapReduce framework for data intensive applications. We evaluate our new methods with a real application of image clustering using K-means clustering in the PolarGrid [8] cluster at Indiana University.

The rest of paper is organized as follows. Section 1 describes the image clustering application and the new K-means algorithm. Section 2 focuses on the design of broadcasting algorithm and presents the experiment results. Section 3 discusses related work and Section 4 contains the conclusion and future work.

1. Clustering Application and New K-means Algorithm

1.1. Social Image Clustering Application

"Social Image Clustering" is an application using social media data, where the data and required analysis is rather different from traditional scientific applications. Note that the volumes of data for large scale artificial intelligence applications have been revolutionized by the social web with for example Flicker and Facebook giving huge pools of data. Both structured and unstructured data require new data mining technologies to cope with the size, high dimensionality, complex structure and hidden features of the data. Clustering is one of key step in image analysis. Later we intend to apply our ideas to "deep learning" which has had substantial popular press [9] and significant results recently in unsupervised feature learning for areas such as computer vision, speech recognition, and natural language processing. Both neural networks [10][11] and clustering [12] are used in this work with billions of parameters determined (as with a 1-10 million clusters with 512-2048 dimensions). It is worth noting that the current optimized (Google) approach to this algorithm does not scale well past around 500-2000 nodes (see Figure 5 of [6]) and a powerful new environment is needed.

In image clustering, the data set is huge and each image is high-dimensional, the dimensionality reduction is done first and each image is represented in a much lower space by a set of important visual components which are called "features." It is analogous to how "key words" are used in a document retrieval system. In this application, 5 patches are selected from each image and each patch is represented 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 [13] (See Figure. 1). In the application data, each HOG feature vector is presented as a line of text starting with picture ID, row ID and column ID, then being followed by 512 numbers f_1, f_2 …and f_{dim}.

We apply K-means Clustering [14] to cluster similar HOG feature vectors and use Twister to parallelize the computation. Because the vectors are static over iterations, we partition the vectors and cache each partition in memory and assign it to a Map task during the configuration. Later in each iteration execution, the driver broadcasts centroids (cluster centers) to all Map tasks and then each Map task updates centroids through assigning points to their corresponding clusters. We use one or more reducers to collect partial local centroids updates from each Map task and calculate new centroids of the iteration. By combining these new centroids from Reduce tasks, the driver gets all updated centroids for the next iteration.

A major challenge of this application is not only the large amount of image data (up to TB level) but also the huge size of clusters. Although we can increase the number of machines to reduce the task size per node, the total intermediate data size for broadcasting and shuffling also grows. Due to the application requirement, the number of centroids is very large. For example, we need to cluster 7 million of image vector data to 1 million clusters (centroids). The execution is conducted on 125 nodes with 10000 Map tasks. For 7 million image data, each node only needs to cache 56K vectors which are approximately 30MB and each task only needs to cache 700 vectors which is about 358KB. However, the total size of 1 million centroids is about 512MB. The centroids data per task is much larger than the image feature vectors per task. As a consequence, the total data for broadcasting is about 64GB. In addition, each map task

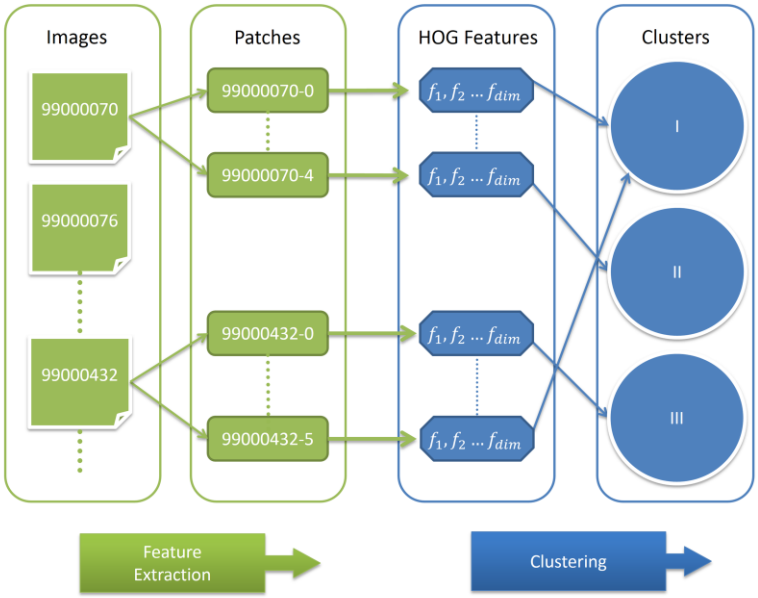

Figure 1. The process of image clustering

generates about 2GB intermediate data. The total intermediate data size in shuffling is about 20TB. This makes the computation difficult to scale.

1.2. Fast Kmeans Algorithm

We propose to incorporate K-means enhancements finding the lower bounds for nearest cluster centers, which can give a large speedup for high dimensional problems. In particular we build on work of Elkan [15] which are independently (and differently) extended by Drake and Hamerly [16]. We address here the step in K-means where points are associated with clusters which is independent of our discussion on reduction which occurs in the step when cluster centers are found from average of their associated points. We have a set of centers c= 1…C with position m(k, c) at iteration k and a set of N fixed points x(P) P=1…N. Then Elkan's algorithm uses two types of inequalities illustrated in Figure 2 below for two iterations k=now and the previous k=last with a distance metric d(a, b) between vectors a and b.

$$d\big(x(P), m(now, c_1)\big) \geq d\big(x(P), m(last, c_1)\big) - d(m(now, c_1), m(last, c_1)) \quad (1)$$

The right side of (1) gives a lower bound on the distance of P from center c_1 in terms of the distance in the previous iteration and the distance between the current and previous positions at centers. One loops through centers c in an order that (based on previous iteration) is most likely to find the center associated with point. Then the lower bound (1) can rule out candidate associated centers c if

$$lower\_bound = d(x(P), m(last, c)) - d(m(now, c), m(last, c)) \geq d(x(P), m(last, c - current\_best)) \quad (2)$$

If (2) is not satisfied, one resorts to explicit calculation of d(x(P), m(now,c)) which takes time of O(Dimension D of space) while the test (2) is independent of D. Elkans also notes a second inequality

$$d(x(P), m(now, c_2)) \geq d(m(now, c_2), m(now, c_1)) - d(x(P), m(now, c_1)) \quad (3)$$

which can be used to rule out centers c2 which are far from c-current best. For our data

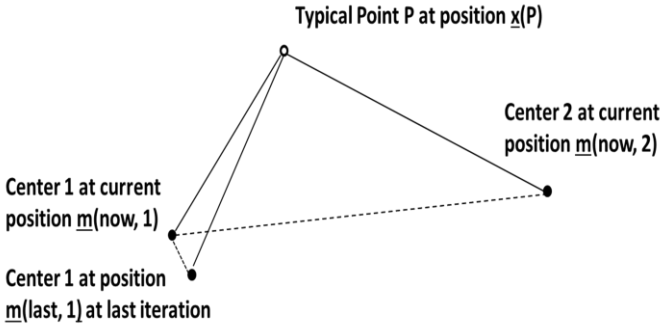

Figure 2. Set of Cluster centers and a Point P to illustrate inequalities used

this test is not as useful as (1). One reason is the closeness of clusters in high dimensional space illustrated by the distances shown in Figure 3 with center-center distances being typically smaller than 2 d(x(P), m(now,c)) which implies that (3) is not effective.

Application of these inequalities drastically reduces the total number of point-center distances needed as shown in Figure 4 where the triangles correspond to this basic algorithm where a lower bound is kept for every point center combination. Figure 4 shows the fraction of point-center distances calculated as a function of iteration. This fraction starts of course at 100% but at the largest iteration count, we are converged and the inequality test (1) is fully effective; one just needs to calculate the new value of the distance between each point and its associated center. Note that it's critical in this (Elkan's style) algorithm to calculate distances in "optimal" order that gives best chance of identifying the cluster associated with each data point as soon as possible and at first try when nearly converged.

The original Elkan's algorithm is not realistic for large problems. One cannot keep a million lower bounds needed for 100 million points clustered with a million centers. Therefore we implemented an improved Elkan's algorithm to give a parallel fast Kmeans algorithm. Each point only keeps the lower bounds for the nearest C_{near} centers plus a single number that bounds the distances d(x(P), m(now,c)) for the remaining C-C_{near} centers. Results are shown in Figure 4 for C_{near} = 400 and 800 for the case of C=3200. The reduction in distance calculations is dramatic for all choices of the number of stored lower bounds.

Implementing this idea has many subtle points which are still being optimized. One starts at the first iteration by calculating all the d(x(P), m(first iteration,c)), sorting them and keeping the lowest C_{near} values and setting the upper bound on the remainder as the C_{near} +1'th entry. Thereafter one tracks at each iteration the current bound or explicit calculation used for each center c. These values are then resorted to produce

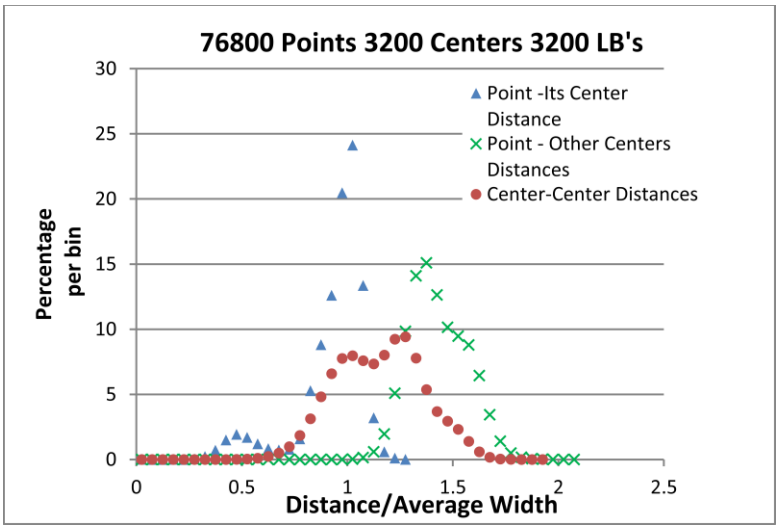

Figure 3. Histograms of distance distributions for 3200 clusters for 76800 points in a 2048 dimensional space. The distances of points to their nearest center is shown as triangles; the distance to other centers (further away) as crosses; the distances between centers as filled circles

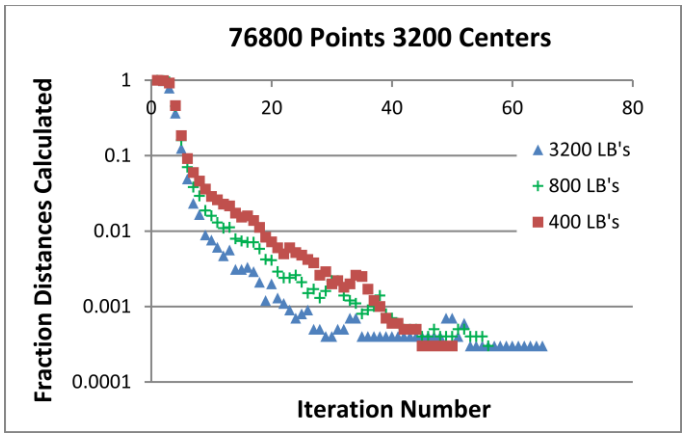

Figure 4: Fraction of Point-Center Distances calculated for 3 versions of the algorithm for 76800 points and 3200 centers in a 2048 dimensional space for three choices of lower bounds LB kept per point

new values for the C_{near} +1 bounds. As one iterates, this approach accumulates ∑(k= start to end) d(m(k,c), m(k-1,c)) which is typically larger than d(m(k=end,c), m(k=start,c)). This is addressed by keeping two sets of C_{near} lower bounds; one calculated from centers at last iteration and other associated with a "start" set of centers. These are slowly updated when center position move significantly and are independent of point. Our contribution here consists of the parallelization of the method; the reduction in number of needed lower bounds and the collection of optimizations illustrated by last discussion.

Note that this algorithm only changes the "map" stage of computation and is perfectly parallel over points x(P) except for the center-only parts of algorithm (calculating d(m(now,c2), m(now,c1)) and d(m(now,c), m(last,c)) that can be performed once and for all independently of points.

One reason of one time calculating center-only is the closeness of clusters in high dimensional space illustrated by the distances shown in Figure 3 with center-center distances being typically smaller than 2 d(x(P), m(now,c)) which implies that (3) is not effective. Note the "curse of dimensionality" produces non-intuitive effects. If you divide a 2D space into a million clusters, they naturally have linear size around 0.001 of total; if you do the same in 20-48 dimensions the linear size of a cluster is naturally 99% (10^{-6} to power 1/2048) of original; this illustrates that in high dimensions clusters are not small. Observations like explain distance plots like that in Figure 3. Note however that inequality (2) is often effective as the change from iteration to iteration is a small fraction of the distances shown in Figure 4.

2. Broadcasting Transfers

2.1. Twister Iterative MapReduce Framework

Twister investigates a data-centric programming model with in-memory data abstraction and efficient collective communication to process big data for data mining or scientific applications. It interpolates between Hadoop and MPI and aims to capture

the best features of these two ecosystems (efficient MPI collective communication libraries and the high level data management of MapReduce with add-ons like Hive, Pig Latin, HBase, MongoDB). Twister leverages the control flow as iterations of MapReduce jobs. By caching invariant data in memory, it can effectively support an iterative computation. Twister uses static scheduling where data are pre-split and evenly distributed to computing nodes based on the availability of computing slots. Tasks are sent to compute nodes where data are located. In Twister, tasks are executed at thread-level. This design feature supports a powerful local combine exploiting an in-memory data pool managing the output Key-Value pairs from each task. In contrast, Hadoop's tasks run at process level and local combining cannot work across tasks.

2.2. Broadcasting in Twister, Hadoop and MPI

Broadcasting is a common operation in iterative algorithms such as the image clustering application. However, cloud systems are mainly based on commodity machines with high-latency network connections, where big data broadcasting could become a bottleneck of overall performance. Many MapReduce or iterative MapReduce frameworks, such as Hadoop, Piccolo [19] or Priter [20] currently do not support optimized collectives including broadcasting. Collective communication operations are familiar from MPI, which include broadcast, reduce(-to-one), scatter, gather, all-gather, reduce-scatter, all-reduce. However, with the support of high-performance hardware, MPI focuses on low latency communication optimizations which are suitable for simulations rather than big data transfers. At high level, MapReduce data model is object with <key,value> pair while MPI messages are primitive data types. We conclude that highly efficient broadcasting and different collective communication trade-offs are important features for data intensive problems.

In MPI, several algorithms are used for broadcasting. MST (Minimum-spanning Tree) method is a broadcasting method used in MPI [17]. Here we use p as the number of compute nodes, n as the data size, α as communication startup time and β as data transfer time per unit. The performance model can be described by the formula below:

$$T_{MST}(p,n) = \lceil log_2 p \rceil (\alpha + n\beta) \quad (4)$$

Although the MST method is much better than the simple method and it changes the factor p to $\lceil log_2 p \rceil$, it is still slow because $(\alpha + n\beta)$ is getting large as the size of message increases.

Scatter-allgather-bucket [17] is another algorithm used in MPI which follows the style of "divide, distribute and gather" [18]. The performance model can be established as follow:

$$T_{bucket}(p,n) = p(\alpha + n\beta/p) + (p-1)(\alpha + n\beta/p) \quad (5)$$

In big data broadcasting, assuming α is small, the broadcasting time is about $2n\beta$. This is much better than MST method because the time looks constant. We show performance result of our implementation for this method on PolarGrid cluster (See Table 1). We observe that the time is stable as the number of nodes grows and about 2 times of 1 GB transferring time between 2 nodes. Nevertheless, it is difficult to directly apply MPI broadcasting methods to cloud system. For scatter-allgather-bucket

algorithm, it is not easy to set a barrier between "scatter" and "allgather" phases to enable all compute nodes to do "allgather" at the same global time through software control in cloud system – some links may have more load than the others that causes network contention.

Table 1 Scatter-allgather-bucket performance on PolarGrid with 1 GB data broadcasting

Number of Nodes	1	25	50	75	100	125
Time	11.4	20.57	20.62	20.68	20.79	21.2

There exists broadcasting method based on InfiniBand multicast implementation in MPI [18]. Many clusters have hardware-supported multicast operation. Although multicast has advantage over broadcasting, it has several problems: its transportation is not reliable; order is not guaranteed and the package size is limited. In this method, after the first stage of multicasting, broadcasting is enhanced with a chain-like method in the second stage. The chain-like broadcasting is reliable by making sure every process has completed data receiving.

Hadoop relies on HDFS to do communication. A component named Distributed Cache is used to cache data from HDFS to local disk of compute nodes. The APIs of addCacheFile and getLocalCacheFiles are used to the process of broadcasting. The data downloading speed depends on the number of replicas in HDFS [21]. Earlier Twister framework used one or multiple messaging brokers [22] [23] to conduct data broadcasting. There are multiple issues. Firstly, unnecessary communication hops through brokers are added between clients, which give poor performance for large messages as they often need extreme long time to transfer from one point to another point. Secondly, brokers do not provide optimal routing for data transferring between brokers and clients. Thirdly, brokers are not always reliable in message transmission and message loss can happen.

Both of these two methods have simple implementations typically sending data to all the nodes one by one. Although using multiple brokers or replicas in HDFS could form a simple 2-level broadcasting tree, they cannot fundamentally solve the scaling problem.

In Twister, broadcasting is implemented as a separate operation. Similar to the concept of Distributed Cache in Hadoop, the operation is called addToMemCache which implies that this method will cache a data object in driver node to all the worker nodes. However it is non-trivial to broadcast objects to remote nodes. The whole broadcasting process consists of 3 stages: serialization, broadcasting and de-serialization. In Twister, data are abstracted and presented as an object in memory. We serialize the object to byte array before broadcasting and de-serialize byte array to form an object after broadcasting. The serialization and deserialization are managed inside of Twister framework and provided with interfaces to allow users to write primitive data types into the byte array, such as *int*, *long*, *double*, *byte* and *String*.

Though separately measured, it is observed that serialization and de-serialization for large-sized data object can take a long time, depending on the data type. For example, our experiments show that serializing 1 GB data of double type is much faster than serializing 1 GB of byte type. Moreover, desterilizing 1 GB byte type data uses longer time than serializing operation. The time it takes is in tens of seconds as shown in Figure 6.

2.3. Chain Broadcasting Algorithm

We propose Chain method, an algorithm based on pipelined broadcasting [27] [28]. In this method, compute nodes in Fat-Tree topology are treated as a linear array and data is forwarded from one node to its neighbor chunk by chunk. The performance is gained by dividing the data into many small chunks and overlapping the transmission 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 forth. This kind of pipelined data forwarding is called "a chain".

The chain algorithm can be deployed differently. Hadoop uses the chain method to write data between replications on HDFS but the replication number on HDFS is usually small with a default value of 3. Our chain broadcasting is targeted for big data broadcasting which could involve more than 100 nodes. By changing user socket buffer size, it indirectly justifies the message size in TCP window for optimal settings. Further Chain broadcasting in Twister is for in-memory data transfer while data replication on HDFS is for on-disk data transfer.

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:

$$T_{Pipeline}(p,k,n) = p(\alpha + n\beta/k) + (k-1)(\alpha + n\beta/k) \quad (6)$$

Here p is the number of daemon nodes (each node is controlled by one daemon process), k is the number of data chunks, n is the data size, α is communication startup time and β is data transfer time per unit. In big data broadcasting, assuming α is small and k is large, the main item 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 $k_{opt} = \sqrt{(p-1)n\beta/\alpha}$ when $\partial T/\partial k = 0$ [27]. However, in practice, the real chunk size per sending is decided by the system and the speed of data transfers on each link could vary as network congestion could happen when data is kept forwarded into the pipeline. As a result, formula (6) cannot be applied directly to predict real performance of our chain broadcasting implementation. The experiment results we will present later still show that as p grows, the broadcasting time keeps constant and close to the bandwidth boundary.

2.4. Topology Impact

This chain method is suitable for Fat-Tree topology [29]. Since each node has only two links, which is less than the number of links per node in Mesh/Torus [30] topology, chain broadcasting can maximize the utilization of the links per node. We make the chain topology-aware by allocating nodes within the same rack close 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 … 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, and makes the core switch overburdened. To support topology-awareness, we define the chain

sequence based on the topology and save the information on each node. Daemons can tell its predecessor and successor by loading the information when starting.

Our topology-aware chain method works remarkably well in cloud system. A main reason is that fat-tree is a common topology in data centers. Furthermore, like our solution, Hadoop utilizes rack locality in data replication. However, the rack locality in Hadoop is pre-configured but not dynamically detected. Dynamic detection could be a good extension for our framework. The point is that the chain algorithm is simple and adaptive to a data center environment.

2.5. Buffer Usage

An important factor that affects broadcasting speed is the buffer usage. The cost of buffer allocation and data copying between buffers are not presented in formula (6). 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 is set to the total size of the data required to be broadcasted.

We observe performance degradation caused by the socket buffer. If the buffer size is smaller than 128 KB, the broadcasting performance slows down. The TCP window may not open up fully, which results in throttling of the sender. Further, large user buffer allocation during broadcasting can also slow down the overall performance. Therefore we initialize a pool of user buffers once Twister daemon starts, instead of allocating dynamically during broadcast communication phase.

2.6. Implementation

We implement chain broadcasting algorithm in the following way: it starts with a request from Twister driver to the first node in the topology-aware chain sequence. Then driver keeps sending a small portion of the data to the next node. At the meanwhile, each node in the chain creates a connection to the successor node. Finally each node receives a partial data from the socket stream, stores it into the application buffer and forwards it to the next node (See Table 2).

Table 2 Broadcasting algorithm

Algorithm 1 Twister Driver side "send" method

```
daemonID ← 0
connection ← connectToNextDaemon(daemonID)
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 Twister Daemon side "receive" method

```
connection ← serverSocket.accept()
dout ← connection.getDataOutputStream()
din ← connection.getDataInputStream()
daemonID ← this.daemonID + 1
connNextD ← connectToNextDaemon(daemonID)
doutNextD ←connToNextD.getDataOutputStream()
dinNextD ← connToNextD.getDataInputStream()

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

2.7. Experiments

To evaluate the performance of the proposed broadcasting method, we conduct experiments on IU PolarGrid cluster. IU PolarGrid cluster 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 connection and the second level of 1 GigE connection. For computing capacity, each compute node in PolarGrid uses a 4-core 8-thread Intel Xeon CPU E5410 2.33 GHz processor. Each compute node has 16 GB total memory.

We test four broadcasting methods: chain method in Twister, MPI_BCAST in Open MPI [31], and broadcasting method in MPJ Express [32], and chain method in Twister without topology awareness. We measure the time from the start of calling the broadcasting method, to the end of return of the calling. Broadcasting is measured from small to medium large scale.

Figure 5 shows that the new chain method produces stable performance results with increasing number of processes, which is explained in Section 2.3. The new method achieves slightly better performance than MPI_BCAST in Open MPI and the time cost is reduced by 20%. However, if the chain sequence is randomly generated without topology-awareness, the performance degrades as the scale increases.

Table 4 compares Twister Chain, MPJ and the simple method. As system errors occur in MPJ when broadcasting 2 GB of data, we use 500MB and 1 GB data in broadcasting experiments. The MPJ broadcasting method is a factor of 4 slower than Twister chain method.

Table 4 Performance Comparison of Twister Chain method and MPJ and simple broadcasting

	Twister Chain			MPJ			Simple Broadcasting		
	500 MB	1 GB	2 GB	500 MB	1 GB	2 GB	500 MB	1 GB	2 GB
1	4.04	8.09	16.17	4.3	8.9	×	4.04	8.08	16.16
25	4.13	8.22	16.4	17.5	35	×	101	202	441.64
50	4.15	8.24	16.42	17.6	35	×	202.01	404.04	882.63
75	4.16	8.28	16.43	17.4	35	×	303.04	606.09	1325.63
100	4.18	8.28	16.44	17.5	35	×	404.08	808.21	1765.46
125	4.2	8.29	16.46	17.4	35	×	505.14	1010.71	2021.3
150	4.23	8.30	16.48	17.4	35	×	606.14	1212.21	2648.6

The results in Table 4 show that highly efficient collective communication can be included in an iterative MapReduce runtime for data intensive problems. Our comparisons suggest that chain method can achieve excellent performance compared to the highly optimized MPI/MPJ algorithms.

The performance model supports the observation of nearly constant chain broadcasting time. Though it is not easy to manipulate the chunk size for sending as being implemented at network level, the socket buffer size can indirectly affect TCP sending window. The impact of socket buffer size is given in Table 5. By examining different socket buffer sizes, we show our implementation leading to excellent performance. Although broadcasting time doesn't include serialization and deserialization, we measure the overheads independently from the communication part of broadcasting in experiments. Figure 6 shows high serialization and de-serialization cost. Note that for the same-sized of data, "byte" type uses more time than "double" type in serialization and de-serialization.

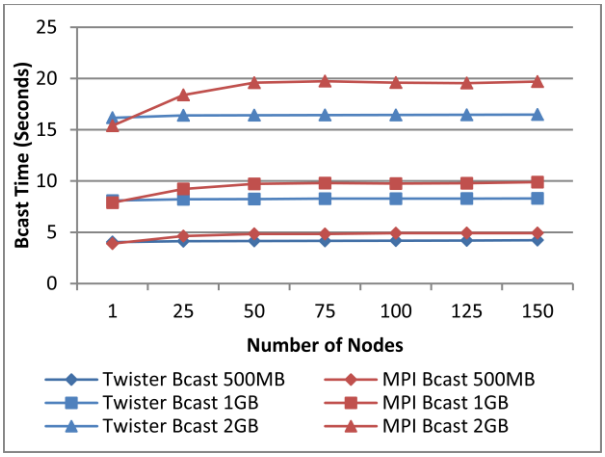

Figure 5. Performance Comparison of Twister Chain method and MPI_Bcast

Table 5 Twister chain broadcasting time of 1GB data on 125 nodes with different socket buffer size

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

3. Related Work

Collective communication algorithms are well studied in MPI runtime. Each communication operation has several different algorithms based on message size and network topology such as linear array, mesh and hypercube [17]. Basic algorithms are pipeline broadcast method [27], minimum-spanning tree method, bidirectional exchange algorithm, and bucket algorithm [17]. Since these algorithms have different advantages, algorithm combination is widely used to improve the communication performance and some solution provides auto algorithm selection [32].

However, many solutions have a different focus from our work. Some of them only study small data transfers up to megabytes level [17][34]. Some solution relies on special hardware support [26]. The data type is typically vectors and arrays whereas we are considering objects. Many algorithms such as "allgather" make the assumption that each node has the same amount of data [17][18], which is not common in MapReduce computation model. As a result, though shuffling can be viewed as a Reduce-Scatter operation, its algorithm cannot be applied directly on shuffling because 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 [25] is such a global control service and architecture to manage intra and inter-transfer activities on Spark [35]. 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 [36] like protocol called Cornet, augmented by topology detection. For shuffling, it uses weighted shuffle Scheduling (WSS) to set the weight of the flow to be proportional to the data size.

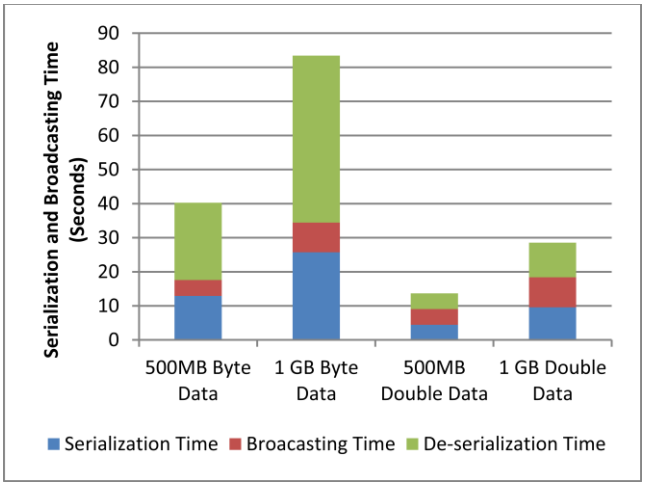

Figure 6. Serialization, Broadcasting and De-serialization

Hadoop-A [37] provides a pipeline to overlap the shuffle, merge and reduce phases and uses an alternative Infiniband RDMA [38] based protocol to leverage RDMA interconnects for fast data shuffling. MATE-EC2 [39] is a MapReduce-like framework for EC2 [40] and S3 [41]. For shuffling, it uses local reduction and global reduction. The 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 [42] and iHadoop [43] are iterative Mapreduce frameworks that optimize the data transfers between iterations asynchronously, where there's 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.

4. Conclusion

We have illustrated the challenges of big data through a social image feature clustering problem and shown the value of a new algorithm that tackles simultaneously the high dimension (reduce number of scalar products calculated) and large cluster count (minimize amount of information needed for each cluster-point combination). This algorithm can be used for other applications and other clustering methods like deterministic annealing. We have also pointed out the new challenges in collective communications which need to be optimized for new regimes. In particular we have demonstrated performance improvement of big data transfers in Twister iterative MapReduce framework enabling data intensive applications. We replace broker-based methods and design and implement a new topology-aware chain broadcasting algorithm. The new algorithm reduces the time cost of broadcasting by 20% of the MPI methods, which pushes it for a new high for (iterative) MapReduce performance in cloud-HPC systems.

There are a number of directions for future work. We will apply the new Twister framework to other iterative applications [44]. We will integrate Twister with Infiniband RDMA based protocol and compare various communication scenarios. The initial observation suggests a different performance profile from that of Ethernet. Further we will integrate topology and link speed detection services and utilize services such as ZooKeeper [45] to provide coordination and fault detection.

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.

References

[1] 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.

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

[4] J.Ekanayake, H. Li, B. Zhang, T. Gunarathne, S. Bae, J. Qiu, G. 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.
[5] Yingyi Bu, Bill Howe, Magdalena Balazinska, and Michael D. Ernst. Haloop: Efficient Iterative Data Processing on Large Clusters. Proceedings of the VLDB Endowment, 3, September 2010.
[6] J. 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.
[7] MPI Forum, "MPI: A Message Passing Interface," in Proceedings of Supercomputing, 1993.
[8] PolarGrid. http://polargrid.org/polargrid.
[9] John Markoff, "Scientists See Promise in Deep-Learning Programs", New York Times, November 23, 2012.
[10] Quoc V. Le et al., "Building High-level Features Using Large Scale Unsupervised Learning", in proceedings of International Conference in Machine Learning 2012.
[11] Adam Coates, Andrej Karpathy, Andrew Y. Ng, "Emergence of Object-Selective Features in Unsupervised Feature Learning", in proceedings of NIPS 2012: Neural Information Processing Systems Conference.
[12] Adam Coates and Andrew Y. Ng, "Learning Feature Representations with K-means", in proceedings of Neural Networks: Tricks of the Trade, Reloaded, Springer LNCS, 2012.
[13] N. Dalal, B. Triggs. Histograms of Oriented Gradients for Human Detection. CVPR. 2005
[14] 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.
[15] 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.
[16] Jonathan Drake and Greg Hamerly, *Accelerated k-means with adaptive distance bounds*, in *5th NIPS Workshop on Optimization for Machine Learning*. Dec 8th, 2012. Lake Tahoe, Nevada, USA,.
[17] 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.
[18] Nikhil Jain, Yogish Sabharwal, Optimal Bucket Algorithms for Large MPI Collectives on Torus Interconnects, ICS '10 Proceedings of the 24th ACM International Conference on Supercomputing, 2010
[19] R. Power and J. Li. Piccolo: building fast, distributed programs with partitioned tables. In OSDI '10, pages 1–14, 2010.
[20] Yanfeng Zhang, Qixin Gao, Lixin Gao, and Cuirong Wang, "PrIter: A Distributed Framework for Prioritized Iterative Computations", in Proceeding of the 2nd ACM Symposium on Cloud Computing (ACM SOCC 2011)
[21] ActiveMQ. http://activemq.apache.org/
[22] S. Pallickara, G. Fox, NaradaBrokering: A Distributed Middleware Framework and Architecture for Enabling Durable Peer to-Peer Grids, Middleware 2003, 2003.
[23] Ford L.R. Jr., Fulkerson D.R., Maximal Flow through a Network, Canadian Journal of Mathematics, 1956, pp.399-404.
[24] 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
[25] Mosharaf Chowdhury et al. Managing Data Transfers in Computer Clusters with Orchestra, Proceedings of the ACM SIGCOMM 2011 conference, 2011
[26] T. Hoefler, C. Siebert, and W. Rehm. Infiniband Multicast A practically constant-time MPI Broadcast Algorithm for large-scale InfiniBand Clusters with Multicast. Proceedings of the 21st IEEE International Parallel & Distributed Processing Symposium. 2007
[27] Watts J, van de Geijn R. A pipelined broadcast for multidimensional meshes. Parallel Processing Letters, 1995, vol.5, pp. 281–292.
[28] T. Kielmann, H. Bal, S. Gorlatch, K. Verstoep, R. Hofman, Network performance-aware collective communication for clustered wide-area systems, Journal of Parallel Computing. PP. 1431-1446, Volume 27 Issue 11, October 2001.
[29] Charles E. Leiserson, Fat-trees: universal networks for hardware efficient supercomputing, IEEE Transactions on Computers, vol. 34 , no. 10, Oct. 1985, pp. 892-901.
[30] S. Kumar, Y. Sabharwal, R. Garg, P. Heidelberger, Optimization of All-to-all Communication on the Blue Gene/L Supercomputer, 37th International Conference on Parallel Processing, 2008
[31] Open MPI, http://www.open-mpi.org

[32] MPJ Express, http://mpj-express.org/
[33] H. Mamadou T. Nanri, and K. Murakami. A Robust Dynamic Optimization for MPI AlltoAll Operation, IPDPS'09 Proceedings of IEEE International Symposium on Parallel & Distributed Processing, 2009
[34] P. Balaji, A. Chan, R. Thakur, W. Gropp, and E. Lusk. Toward message passing for a million processes: Characterizing MPI on a massive scale Blue Gene/P. Computer Science - Research and Development, vol. 24, pp. 11-19, 2009.
[35] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster Computing with Working Sets. In HotCloud, 2010.
[36] BitTorrent. http://www.bittorrent.com.
[37] Yangdong Wang et al. Hadoop Acceleration Through Network Levitated Merge, International Conference for High Performance Computing, Networking, Storage and Analysis (SC'11), 2011
[38] Infiniband Trade Association. http://www.infinibandta.org.
[39] T. Bicer, D. Chiu, and G. Agrawal. MATE-EC2: A Middleware for Processing Data with AWS, Proceedings of the 2011 ACM international workshop on Many task computing on grids and supercomputers, 2011
[40] EC2. http://aws.amazon.com/ec2/.
[41] S3. http://aws.amazon.com/s3/.
[42] Y. Zhang, Q. Gao, L. Gao, and C. Wang. imapreduce: A distributed computing framework for iterative computation. In DataCloud '11, 2011.
[43] E. Elnikety, T. Elsayed, and H. Ramadan. iHadoop: Asynchronous Iterations for MapReduce, Proceedings of the 3rd IEEE International conference on Cloud Computing Technology and Science (CloudCom), 2011
[44] B. Zhang, Y. Ruan, T. Wu, J. Qiu, A. Hugh, G. Fox. Applying Twister to Scientific Applications, Proceedings of the 2nd IEEE International conference on Cloud Computing Technology and Science (CloudCom), 2010
[45] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed, ZooKeeper: wait-free coordination for internet-scale systems, in USENIXATC'10: USENIX conference on USENIX annual technical conference, 2010, pp. 11–11.