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.**

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

I.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.

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 C++ OpenMPI methods [10] by 20% and Java MPJ by a factor of 4. We also use local aggregation in Map stage to reduce the size of intermediate data by at least 90%. 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.

II.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].

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 HOG 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 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.

TABLE 1. 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	$O(kl)$	$O(kl)$

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 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).

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. 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.

III.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 (See Figure 3). 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.

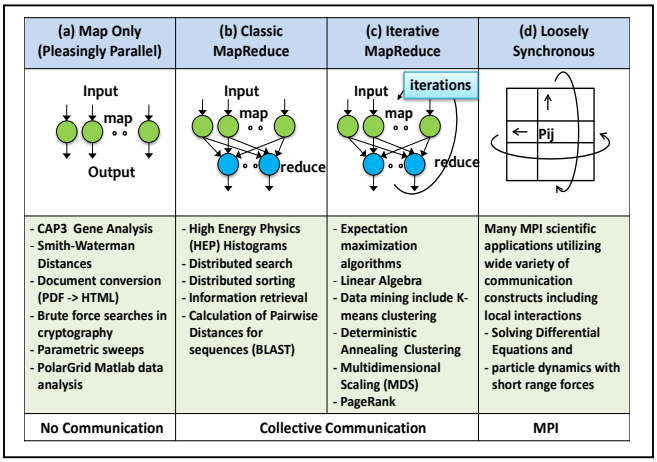

Figure 3: Classification of Applications and Communication Patterns

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 [14]. The first four, broadcast, scatter, gather, and allgather are called "data redistribution operations" [14]. The remaining three, reduce(-to-one), reduce-scatter, all-reduce are called "data consolidation operations" [14]. 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 [15] 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.

In MPI, several algorithms are used for broadcasting. MST (Minimum-Spanning Tree) method is a typical broadcasting method used in MPI [14]. This method is much better than simple broadcasting by changing the complexity term p to $\lceil log_2 p \rceil$ (p is the number of nodes). 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" [16]. 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 [14]. 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. In addition, there is also the InfiniBand [17] multicast based broadcasting method in MPI [18].

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. 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 BitTorrent [19] 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. So broadcasting doesn't execute on all the nodes but only on the nodes where tasks are located.

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, any word can be a key. 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. 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. Therefore 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.

As 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.

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).

IV.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.

A. Chain Broadcasting Algorithm

Here we propose chain method, an algorithm based on pipelined broadcasting [21]. In this method, compute nodes in Fat-Tree topology [22] 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 [21]. 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, α is communication startup time and β is data transfer time per unit. In large data broadcasting, assuming α 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 $\partial T/\partial k = 0$ [21]. 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.

B. 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 [23]. Since each node only has two links, which is less than the number of links per node in Mesh/Torus [24] 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.

C. 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.

D. 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.

V. 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) [20] 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:

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

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.

VI. 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.

A. 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. For computing capacity, each compute node in PolarGrid uses a 4-core 8-thread Intel Xeon CPU

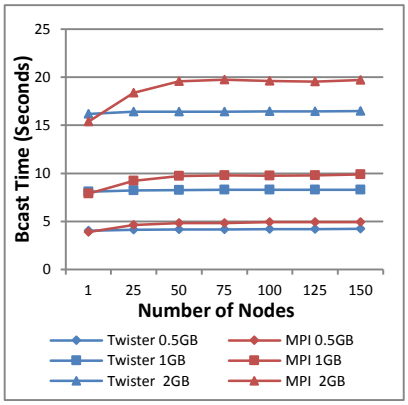

Figure 6. Performance comparison of Twister chain method and Open MPI MPI_Bcast

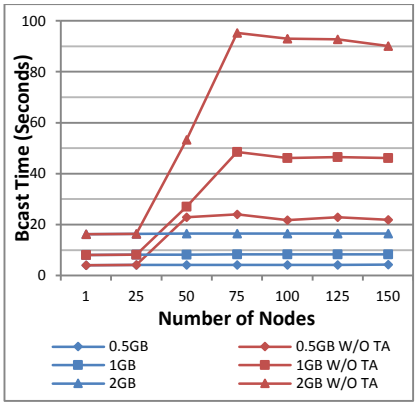

Figure 8. Chain method with/without topology-awareness

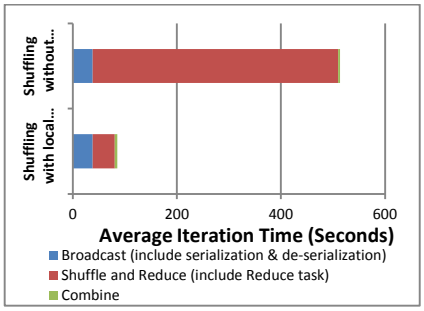

Figure 9. Comparison between shuffling with and without local aggregation

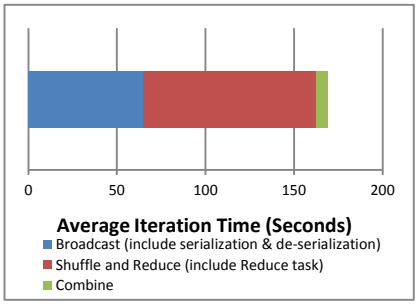

Figure 10. Communication cost per iteration of the image clustering application

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.

B. Broadcasting

We compared chain method with MPI_BCAST in Open MPI 1.4.1 [25]. We also compare the current 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.

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.

C. 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.

D. Image Clustering Application

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 l (the dimension which is currently 512 to 2048) and so the improved communication time is highly relevant.

VII.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 [14]. Basic algorithms are pipeline broadcast method [21], minimum-spanning tree method, bidirectional exchange algorithm, and bucket algorithm [14]. Since these algorithms have different advantages, algorithm combination (polymorphism) is widely used to improve the communication performance [14]. Furthermore some solutions also provide auto algorithm selection [26].

Other papers have a different focus than our work. Some of them only study small data transfers up to megabytes level [14] [27] while some solutions rely on special hardware support [16]. 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 [14] [16], 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 [20] 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 [28] 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 [29] is a MapReduce like framework for EC2 [30] and S3 [31]. 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 [32] and iHadoop [33] 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 [38] is a recently announced iterative MapReduce Azure runtime developed by Microsoft that builds on some of the ideas of the earlier Twister system. Currently Excel DataScope is presented as an application of Daytona where an Excel interface allows manipulation of cloud or local data. The results can be returned to the Excel client or remain in the cloud for further processing and visualization.

VIII.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][35] and apply the Map-Collective framework to other iterative applications [36] 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 [15] 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 [37] 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] 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] 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.
[15] Thilina Gunarathne, Bingjing Zhang, Tak-Lon Wu, and Judy Qiu, "Scalable Parallel Computing on Clouds Using Twister4Azure Iterative MapReduce ", Future Generation Computer Systems vol. 29, pp. 1035-1048, 2013.
[16] 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
[17] Infiniband Trade Association. http://www.infinibandta.org.
[18] 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
[19] BitTorrent. http://www.bittorrent.com.
[20] Mosharaf Chowdhury et al. Managing Data Transfers in Computer Clusters with Orchestra, Proceedings of the ACM SIGCOMM 2011 conference, 2011
[21] Watts J, van de Geijn R. A pipelined broadcast for multidimensional meshes. Parallel Processing Letters, 1995, vol.5, pp. 281–292.
[22] Charles E. Leiserson, Fat-trees: universal networks for hardware efficient supercomputing, IEEE Transactions on Computers, vol. 34 , no. 10, Oct. 1985, pp. 892-901.
[23] Radhika Niranjan Mysore, PortLand: A Scalable Fault-Tolerant Layer 2 Data Center Network Fabric, SIGCOMM, 2009
[24] 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
[25] Open MPI, http://www.open-mpi.org
[26] MPJ Express, http://mpj-express.org/
[27] 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
[28] 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.
[29] Yangdong Wang et al. Hadoop Acceleration Through Network Levitated Merge, International Conference for High Performance Computing, Networking, Storage and Analysis (SC'11), 2011
[30] 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
[31] EC2. http://aws.amazon.com/ec2/.
[32] S3. http://aws.amazon.com/s3/.
[33] Y. Zhang, Q. Gao, L. Gao, and C. Wang. imapreduce: A distributed computing framework for iterative computation. In DataCloud '11, 2011.
[34] E. Elnikety, T. Elsayed, and H. Ramadan. iHadoop: Asynchronous Iterations for MapReduce, Proceedings of the 3rd IEE International conference on Cloud Computing Technology and Science (CloudCom), 2011
[35] 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.
[36] Bingjing Zhang, Yang Ruan, Tak-Lon Wu, Judy Qiu, Adam Hughes, Geoffrey Fox. Applying Twister to Scientific Applications, Proceedings of the 2nd IEE International conference on Cloud Computing Technology and Science (CloudCom), 2010
[37] 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.
[38] Microsoft Daytona. Retrieved Feb 1, 2012 http://research.microsoft.com/en-us/projects/daytona/.