Clustering Social Images with MapReduce and High Performance Collective Communication

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**

Social Image clustering is a data intensive application that provides novel challenges to high performance computing. Already this field has reached 10-100 million images represented as points in a high dimensional (up to 2048) vector space that are to be divided into up to 1-10 million clusters. In recent years MapReduce has become popular in processing big data problems due to its attractive programming interface with scalability and reliability. However, because social image clustering is an iterative data mining application, the performance of traditional MapReduce frameworks is poor on this problem due to the overhead of repeated disk access. By caching invariant data between iterations into local memory on each node, our Twister system is able to accelerate MapReduce iterations. In each iteration, data is processed through 5 stages: Broadcast, Map, Shuffle, Reduce and Combine and this paper focuses on the four collective communication stages where the large data sizes allow and demand performance optimization where we combine ideas from the MapReduce and MPI communities. We present detailed analysis of the division of 7 million image feature vectors in 512 dimensions into 1 million clusters, executing the application on 1000 cores (125 nodes each of which has 8 cores) with 10000 Map tasks and 125 Reduce tasks. Particular challenges are the 20TB of intermediate data generated in the shuffling stage and 512MB of data to be broadcast. We compare several approaches to broadcasting showing that a topology-aware and pipeline-based method gives improved performance of a factor of 100 compared to simple original method; a factor of 5 compared to basic pipelined methods and a factor 1.2 compared to MPI. Further we add a local reduction stage before the shuffle which reduces the 20 TB intermediate data to “just” 250 GB. We discuss the next steps that will scale to larger problems and introduce new algorithms that will speed up the map stage by one to two orders of magnitude and highlight need for high performance collectives developed in this paper.

**Categories and Subject Descriptors**

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

**General Terms**

Algorithms, Measurement, Performance, Design, Experimentation.

**Keywords**

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

# 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. However, many existing analysis tools are not capable of handling such big data sets. MapReduce frameworks have become popular in recent years for their scalability and fault tolerance in large data processing and simplicity in programming interface. Hadoop [1], an open source implementation following original Google’s MapReduce [2] concept, has been widely used in industry and academia.

Intel’s RMS (Recognition, Mining and Synthesis) taxonomy [3] identifies iterative solvers and basic matrix primitives as the common computing kernels for computer vision, rendering, physical simulation, financial analysis and data mining. 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. Several new frameworks designed for iterative MapReduce are proposed to solve this problem, including Twister [4] and HaLoop [5].

Social image clustering is such a kind of application which is not only a big data problem but also needs an iterative solver. Already problems in this space can involve 10-100 million images represented as points in a high dimension (500-2000) space which are to be divided into 1-10 million clusters. This produces challenges for both new algorithms and efficiency of the parallel execution which involves very large collective communication steps. We are addressing all these issues with an extension of Elkan's algorithm [40 (needs reording)] drastically speeding up the computing (map) step of algorithm by use of the triangle inequality to remove unnecessary computation. However his just highlights the need for efficient communication that we study here. Note communication has been well studied, especially for MPI, but the new application area stresses different usage modes from most previous work. Secondly k-means clustering algorithm, the algorithm used to solve the clustering problem needs several iterations to reach the local optimization. However, classic MapReduce frameworks such as Hadoop are too inefficient to meet the requirement of executing this iterative algorithm. The reason is that input data and all kinds of intermediate data are very large which can arrive to terabytes in our experiments and they are loaded again and again between iterations.

We use Twister to process this image clustering application. Twister is an iterative MapReduce framework developed by our group. In the early version of Twister, we extend the original MapReduce flow to iterative MapReduce flow by adding a new “Combine” stage to bridge MapReduce iterations. To avoid repeated disk access, we cache invariant data in memory and use TCP streaming to do shuffling. However, there are performance issues of collective communication in iterative algorithms execution not addressed in Twister. We notice that in the image clustering application, broadcasting and shuffling could cost lots of execution time and limit the scalability of the execution. To cluster 7 million image feature vectors to 1 million clusters, we execute the application on 1000 cores (125 nodes each of which has 8 cores) with 10000 Map tasks and 125 Reduce tasks. In broadcasting, 512 MB data are required to be broadcasted to all the Map tasks. In shuffling, 20 TB intermediate data generated in Map stage must be transferred in the shuffling stage. In past, to broadcasting 512 MB data, Twister needs about 10 minutes with brokers. For 20 TB intermediate data, Twister even cannot handle it due to its in-memory model. In this paper, we propose topology-aware pipeline-based method to accelerate broadcasting by at least a factor of 120 comparing with naïve algorithm (sequentially sending data to each destination) and show that it even outperforms classic MPI methods [6] by 20%. We also use local reduction before shuffling to reduce the size of intermediate data by at least 90% and reduce 20 TB intermediate data to 250 GB. These methods provide important capabilities of our new iterative MapReduce framework for data intensive applications. Finally we evaluate our new methods in PolarGrid [7] cluster at Indiana University.

The rest of paper is organized as follows. Section 2 discusses the image clustering application. Section 3 introduce Twister tool and analyzes the data model inside Twister and how it is used to process big data problem. Section 4 presents the design of broadcasting algorithm. Section 5 investigates how the new shuffling mechanism works. Section 6 shows experiments and results. Section 7 discusses related work and Section 8 is about conclusion and future work.

# IMAGE CLUSTERING APPLICATION

Image clustering application is to group millions of images to millions of clusters each of which contains as set images with similar visual features. Before doing image clustering, we notice that the original data of each image is high-dimensional and the total data set is huge, so the dimensionality reduction is done first and each image is represented in a much lower space with a set of important visual components which are called “feature vectors”. Analogous to how “key words” are used in a document retrieval system, these “features vectors” become the “key words” of an image. In this application, 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 [8] (See Figure 1). In the application input 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 f1, f2 …and fdim.

We apply K-means Clustering [9] to cluster the similar HOG feature vectors and use Twister MapReduce framework to paralyze the computation. We depict K-means Clustering algorithm as a chain of MapReduce jobs with each iteration per MapReduce job. We treat input data, a large number of feature vectors as high dimensional data points each of which contains 512 dimensions and use Euclidean distance calculation to compare the distances between data points. We notice that the vectors are static over iterations. So we partition the vectors and cache each partition in memory and assign it to a Map task during the job configuration. Later in each iteration execution, the job driver firstly broadcasts cluster centers to all Map tasks and then each Map task assign points to their nearest cluster centers based on Euclidean distance calculation and for each cluster, for each cluster center, Map tasks collect the sum of coordination values of data points in the cluster and count the total number of these data points. The Reduce task (To simplify the algorithm introduction, we use only one Reduce task here) processes the output collected from each Map task and calculate new cluster centers of the iteration by adding all partial sums of coordination values together and letting it be divided by the total count of the data points in the cluster. By combining these new centroids from Reduce tasks, the job driver gets all updated centroids and the control flow enters the next iteration (See Table 1).

D:\my research\HPDC\chart2.tif

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

A major challenge of this application is the amount of the image data can be very large. Currently we have near 1 TB data and it can grow as long as new images are put into the data set. For such a large input data, though we can increase the number of machines to lower down the data size per node, the total data size required for broadcasting and shuffling still grows.

For example, in a real job execution, we need to cluster 7 million vectors to 1 million clusters. The execution is done on 1000 cores (125 nodes, each of which has 8 cores) with 10000 Map tasks (Using 10000 short Map tasks instead of using 1000 long Map tasks is due to an unknown crash from JVM observed after iterations during 1000 long Map tasks execution). For 7 million image data, each task only needs to cache 700 vectors which is about 358KB and each node only needs to cache 56K vectors which are about 30MB in total. But for broadcasting data, by the requirement of image clustering, the number of cluster centers is very large and the total size of 1 million cluster centers is about 512MB. So 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 broadcasting grows as the number of node grows. For 125 nodes and the execution above, the total data sent through broadcasting is about 64 GB (Because Map tasks are executed on thread level, broadcasting data can be shared among tasks on one node).

Besides, for shuffling data, because each map task generates about 2 GB intermediate data, the total intermediate data size in shuffling is about 20 TB. This kind of big data cannot be handled by Twister in-memory work model because 20 TB far exceeds the total memory size of 125 nodes (each of which has 16 GB memory, 2 TB in total). 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 intermediate data to 250 GB. But due to the memory limitation, 250 GB still cannot be handled by one Reduce task. So we chunk 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 process 2 GB 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. The output from each Reduce task is only about 4 MB. For 125 Reduce tasks, the total data required to be combined back to the job driver is about 512 MB which is very small and easy to handle.

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

|  |
| --- |
| **Algorithm 1 Job Driver** |
| numLoop ← maximum iterations  centroids[0] ← initial centroids value  driver ← new TwisterDriver(jobConf)  driver.configureMapTasks(partitionFile)  **for**(i ← 0; i < numLoop; i ← i+1)  driver.broadcast(centroids[i])  driver.runMapReduceJob()  centroids[i+1] ←driver.getCurrentCombiner().getResults() |
| **Algorithm 2 Map Task** |
| vectors ← load and cached from files  centroids ← load from memory cache  minDis ← new int[numVectors]  minCentroidIndex ← new int[numVectors]  **for** (i ← 0; i < numVectors; i ← i+1)  **for** (j ← 0; j < numCentroids; j ← j+1)  dis ← getEuclidean(vectors[i], centroids[j])  **if** (j = 0)  minDis[i] ← dis  minCentroidIndex[i] ← 0  **if** (dis < minDis[i])  minDis[i] ← dis  minCentroidIndex[i] ← j  localSum ← new int[numCentroids][512]  localCount ← new int[numCentroids]  **for**(i ← 0; i < numVectors; i ← i+1)  localSum[minCentroidIndex[i]] +← vectors[i]  localCount[minCentroidIndex[i]] +← 1  collect(localSum, localCount) |
| **Algorithm 3 Reduce Task** |
| localSums ← collected from Map tasks  localCounts ← collected from Map tasks  totalSum ← new int[numCentroids][512]  totalCount ← new int[numCentroids]  newCentroids ← new byte[numCentroids][512]  **for** (i ← 0; i < numLocalSums; i ← i+1)  **for** (j ← 0; j < numCentroids; j← j+1)  totalSum[j] = totalSum[j] + localSums.get(i)[j]  totalCount[j] = totalCount[j] + localCounts.get(i)[j]  **for** (i ← 0; i < numCentroids; i← i+1)  newCentroids[i] = totalSum[i]/ totalCount[i]  collect(newCentroids) |

Here we also give the time complexity of each algorithm, we use as the number of daemons, as the number of Map tasks and as the number of Reduce tasks. For data, as the size of centroid data, as the total number of image feature vectors, as intermediate data output per Map task, as intermediate data output per Reduce task (See Table 2).

**Table 2. Time Complexity of Each Stage**

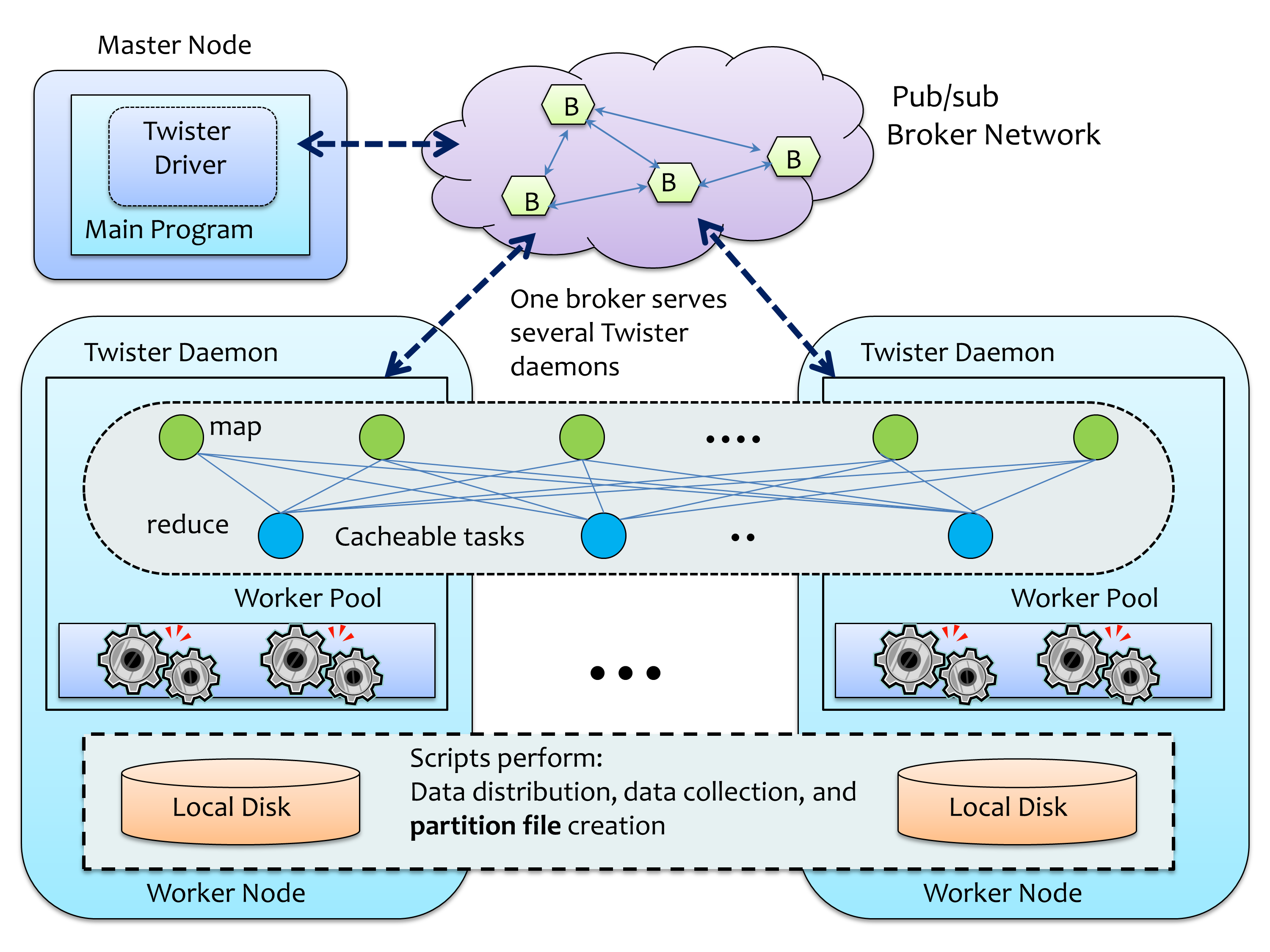
|  |  |
| --- | --- |
| **Stage** | **Cost** |
| Broadcasting (naïve) |  |
| Map |  |
| Shuffle |  |
| Reduce |  |
| Combine |  |

# DATA TOOLS

In this section, we give an overview of Twister iterative MapReduce framework and show how it is used to process big data problems. We discuss the work model and data model in Twister and compare it with two other main data processing tools: Hadoop and MPI at the two ends of the whole tool spectrum.

## **Work Model in Twister**

Twister iterative MapReduce framework is implemented in Java. It generally run on local clusters or HPC but can also be deployed on virtual machines in cloud system. It has several components, a job driver to control the chain of MapReduce jobs, and a large number of daemons running on work nodes to handle requests from the driver and execute iterative MapReduce jobs (See Figure 2.). Through the discussion of the application, we already present the basic workflow of iterative MapReduce in Twister. To support iterative algorithms, Twister program allows users to configure an iterative MapReduce job and then to drive the job execution iteratively with a loop control. Besides Twister has a Combine stage after Reduce stage. At the end of each iteration, the driver can collect output back from worker nodes by Combine task and then goes to the next iteration.



**Figure 2. Work model of Twister [4]**

As a result, though Twister and Hadoop both follow MapReduce work model but Twister makes some extensions to support iterative algorithms (See Table 3). For fault tolerance, Hadoop provides task level fault tolerance, but currently Twister only provides checkpointing between iterations. There are other detailed differences in implementation. The work unit for Map tasks and Reduce tasks in Twister are threads but they are processes in Hadoop. For communication, both rely on TCP. But on top of it, Twister components are not only communicated through TCP stream directly but also messages via broker software including ActiveMQ [10] and NaradaBrokering [11] in a publish/subscribe mechanism. For scheduling, Twister uses static scheduling, but Hadoop used more complicated dynamic task scheduling.

Another commonly used tool in parallel data processing under distributed environment is MPI which can spawn several processes working in parallel. MPI tool doesn’t follow MapReduce model. On the contrary, programmer needs to design what each process does and how they communicate. It is flexible to simulate MapReduce model or other user defined work models but is very complicated to use. MPI is highly optimized in performance with added fault tolerance. In implementation, it supports different kinds of communication protocols including not only TCP but also shared memory and Infiniband [12]. MPI also uses static scheduling (See Table 3)

**Table 3. Comparison of the work models**

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Twister** | **Hadoop** | **MPI** |
| **Language** | Java | Java | C |
| **Environment** | clusters, HPC, cloud | clusters, cloud | HPC, super computers |
| **Job Control** | Iterative MapReduce | MapReduce | parallel processes |
| **Fault Tolerance** | iteration level | task level | added fault tolerance |
| **Work Unit** | thread | process | process |
| **Communication Protocol** | broker, TCP | RPC, TCP | TCP, shared memory, Infiniband |
| **Scheduling** | static | dynamic, speculative | static |

## **Data Model in Twister**

The reason why these 3 tools have different work models is that they serve different application and data. We see MPI and Hadoop are at the two ends of the whole data tool spectrum. MPI is a computation-centric solution. It has no specific work model so that users can customize and optimize the work flow for any applications. MPI basically serves scientific applications which are not only complicated in workflow but also intensive in computation. Though the whole framework is highly optimized in performance, it is not friendly to big data processing.

At the same time, Hadoop is a data-centric solution. With the support of HDFS [13], big data is well stored and managed. Users don’t need to think about data accessing and loading as what they do in MPI programs. Besides, computation is moved to the place where data is stored. This framework is scalable when processing big data but its work model is constrained to MapReduce pattern. For complicated data mining algorithms and scientific applications which needs iterations or cannot be simply expressed with Map and Reduce tasks, Hadoop is inefficient for processing this type of data. As a result, we see Hadoop is strong but slow. The typical data processed in Hadoop is records or logs. This type of data is easy to split into small Key-Value objects and not like scientific data which contains large chunks of vectors or matrices. Usually the computation on these data can be easily expressed in Map-Reduce pattern.

As a result, Twister is at the middle between Hadoop and MPI. We hope to provide an easy-to-use and data-centric solution to process big data in data mining or scientific applications efficiently. As what is mentioned above in work model part, we extend MapReduce model to iterative MapReduce model to support iterative algorithms. This kind of model is more powerful than traditional MapReduce model but still keep the simplicity without much interfering and controlling from the user. For data model, we move toward Hadoop direction, but not follow MPI which has little data management and control (See Table 4).

For data source, Twister mainly uses local disks and does simple management through scripts. Twister can also use distributed file systems which are mounted through network and can be accessed in a manner which is just like accessing local disks. This is similar to MPI but different from Hadoop which relies on HDFS. We are also moving toward using HDFS. At this stage in Twister, files and their replicas are stored on local disks of compute nodes and recorded in a partition file.

We notice that the data used in computation is not organized in the same way as the data stored in disk. For example, the data in the image clustering application are stored in a set of text files. Each file contains feature vectors generated from a related set of images. The length of file and the number of files usually varies. However, in computation we hope the number of data partitions is the same as the number of cores so that we can evenly distribute the computation. So we need to convert “raw” data in disk to “cooked” data objects in memory. Currently we implement an application to split original data files into a set of even sized data partitions. Then later Twister can load the data partitions based on the partition file which records all the available partitions and let them be consumed by Map workers. The Max Flow algorithm [14] is used to balance the mapping between Map workers and the file replicas. But Hadoop can automatically load data with self-defined InputSplit or InputFormat class. This gives a better option for future Twister implementation. At the same time, MPI requires user to split data or use special file format HDF5 [15] and NetCDF [16] commonly used in scientific applications.

One significant improvement in Twister to support iterative algorithms is to do data caching along the iterations. We cache data as Key-Value objects during the job configuration stage. Though Hadoop also processes data in memory as Key-Value objects but it doesn’t provide caching mechanisms. MPI can cache data in memory depending on how programmers write the application, but it only caches data as basic types or vectors without higher abstraction.

We also notice that in many applications data is not processed and outputted in local directly. It is common that intermediate data generated during processing are required to be exchanged under collective communication operations. Currently Twister supports two communication operations on data. One is broadcasting and another is shuffling. Since data is cached in memory, we optimize the memory-to-memory collective communication. Hadoop also supports these two operations but only simply support them with file transfers. On the contrary, MPI provide abundant options for collective communication operations [17].

**Table 4. Comparison of the data models**

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Twister** | **Hadoop** | **MPI** |
| **Application Data Category** | scientific data (vectors, matrices) | records, logs | scientific data (vectors, matrices) |
| **Data Source** | local disk, DFS | local disk, HDFS | DFS |
| **Data Format** | text/binary | text/binary | text/binary/ HDF5  /NetCDF |
| **Data Loading** | partition based | InputSplit, InputFormat | customized |
| **Data Caching** | yes | no | yes |
| **Data Caching Unit** | Key-Value objects | Key-Value objects | basic types, vectors |
| **Collective Communication** | in memory | file transfer | in memory |

# BROADCASTING TRANSFERS

To solve the performance issue of broadcasting in image clustering application, we replace original broker methods with new methods based on TCP sockets to provide customized control the message routes in broadcasting. We propose our new chain broadcasting method which can utilize the bandwidth per link and topology advantage more efficiently.

## **Broadcasting and in Hadoop and MPI**

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** co-work together to finish the process of broadcasting. However, there is no special optimization for the whole process. The data downloading speed depends on the number of replicas in HDFS [18].

These kinds of methods are naïve because they basically send data to all the nodes one by one. 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 solve the problem.

In MPI, several algorithms are used for broadcasting. MST (Minimum-spanning Tree) method is a typical broadcasting method used in MPI [17]. In this method, nodes form a minimum spanning tree and data is forwarded along the links. In this way, the number of nodes which have the data grows in geometric progression. Here we use as the number of daemon nodes, as the data size, as communication startup time and as data transfer time per unit. The performance model can be described by the formula below:

(1)

Though this method is much better than the naïve broadcasting by changing the factor to , the method is still slow because the term is getting large as the size of message increases.

Scatter-allgather-bucket algorithm is another algorithm used in MPI for long vectors broadcasting which follows the style of “divide, distribute and gather” [19]. In “scatter” phase, it scatters the data to all the nodes. To do this, it can use MST algorithm or a naïve algorithm. Then in “allgather” phase, it views the nodes as a chain. At each step, each node sends data to its right neighbor [17]. By taking advantage of the fact that messages traversing a link in opposite direction do not conflict, we can do “allgather” in parallel without any network contention. The performance model can be established as follow:

(2)

In large data broadcasting, assuming is small, the broadcasting time is about. This is much better than MST method because the time looks constant. However, since it is not practical to set barrier between “scatter” and “allgather” phases to enable all the nodes to do “allgather” at the same global time through software control, some links will have more load than the others and thus it causes network contention. Here is performance result of our rough implementation of this method on PolarGrid (See Table 5). We see that the time is stable as the number of nodes grows and about twofold time cost of 1 GB transferring between 2 nodes.

**Table 5. Scatter-allgather-bucket performance on IU PolarGrid with 1 GB data broadcasting**

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

There is also InfiniBand multicast based broadcasting method in MPI [20]. Since many clusters have hardware-supported multicast operation, multicast has advantage to do broadcasting. However, multicast also has problems mainly because its transportation is not reliable, order is not guaranteed and the package size is limited. So in MPI, after the first stage of multicasting, broadcasting is enhanced with a chain-like broadcasting in the second stage. The second stage of broadcasting is reliable to make sure every process has completed data receiving. In the second stage, the nodes are formed into a virtual ring topology. Each MPI process that gets the message via multicast serves as a new “root” within the virtual ring topology and exchange data to the predecessor and successor in the ring. This is a little similar to the bucket algorithm we discuss above.

## **Broadcasting in Twister**

Early Twister work used messaging brokers to conduct data broadcasting. We firstly use one broker but later extend to multiple brokers. Though multiple-broker method is better than single-broker method, we still find the following issues. Firstly, unnecessary communication hops through brokers are added in data transfers between clients, which give poor performance for big messages as they often need significant time to transfer from one point to another point. Secondly, the broker network doesn’t provide optimal routing for data transferring between a set of brokers and clients in collective communication operations. Thirdly, brokers are not always reliable in message transmission and message loss can happen. As a result, we abandon broker based methods.

As what we see in the introduction of the image clustering application, broadcasting is a separate and independent operation in Twister APIs. Similar to the concept of Distributed Cache in Hadoop, the operation is called **addToMemCache** which means 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 process of the API has 3 stages: serialization, broadcasting and de-serialization. In the following sections, we firstly talk about serialization and de-serialization, and then we focus on discussing the details of broadcasting.

## **Object Serialization and De-serialization**

In Twister broadcasting, data are abstracted and presented as an object in memory. So we need to serialize the object to byte array before broadcasting and de-serialize byte array back to an object after broadcasting. We manage serialization and deserialization inside of Twister framework and we provide interfaces to let user be able to write different basic types into the byte array, such as “int”, “long”, “double”, “byte” and “String”.

We observe that large-sized data object serialization and de-serialization can take very long time. Depending on the data type, the serialization speed varies. Our experiments show that serializing 1 GB data in double type is much faster than serializing 1 GB byte type data. Moreover, desterilizing 1 GB byte type data even uses longer time than serializing it. The time cost on this part can take tens of seconds. Since it is local operation, currently we leave them there and separate them from the core byte array broadcasting.

## **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. 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 [21]. This kind of pipelined data forwarding is called “a chain”.

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:

(3)

Here is the number of daemon nodes (each node is controlled by one daemon process), is the number of data chunks, is the data size, is communication startup time and is data transfer time per unit. In large data broadcasting, assuming is small and is large, the main item of the formula is which is close to constant. From the formula, the best number of chunks when [21]. 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 (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 grows, the broadcasting time keeps constant and close to the bandwidth boundary.

## **Topology Impact**

This chain method is suitable for Fat-Tree topology which is a commonly used network topology in clusters or in data centers [22] [23]. Since each node only has only 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 be topology-aware by allocating nodes within the same rack close in the chain. Assuming the racks are numbered as, and …, the nodes in are put at the beginning of the chain, then the nodes in follow the nodes in, and then nodes in follow nodes in …. Otherwise, if the nodes in are intertwined with nodes in 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. In future, we are also looking into supporting Automatic topology detection to replace the static topology information loading.

## **Buffer Usage**

Another important factor affecting broadcasting speed is the buffer usage. The cost of buffer allocation and data copying between buffers are not presented 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 is set to the total size of the data required to be broadcasted.

We observed the performance degradation caused by buffer usage. One is that if the socket buffer is smaller than 128 KB, the broadcasting performance can be slowed down probably because the TCP window cannot open up fully and result in throttling of the sender. Besides the large-sized user buffer allocation during broadcasting can also slight slow-down of the overall performance. To make an apple-to-apple comparison with MPI which does buffer initialization before broadcasting, we initialize a pool of free buffers once Twister daemon starts instead of allocating one during the broadcasting.

## **Fault Tolerance**

Furthermore, fault tolerance is also considered in chain broadcasting. When large data are transmitted among large number of nodes, node failure is inevitable. Several strategies are applied here. Firstly it there are failures in establishing connection from node-to-node, a retry is issued. Alternatively it moves on to other destinations. Secondly, if the chain is broken and exceptions thrown in the driver side, the whole broadcasting will restart. Thirdly, at the end of broadcasting, the driver waits and checks if all the nodes have received all the data blocks. If the driver doesn’t get all the ACK within a time window, it restarts the whole broadcasting.

## **Implementation**

We implement the 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, for the nodes in the chain, each node creates a connection to the successor node in the chain. Next each node receives a partial data from the socket stream, store it into the application buffer and forward it to the next node (See Table 6).

**Table 6. Broadcasting algorithm**

|  |
| --- |
| **Algorithm 1 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() |

# SHUFFLING TRANSFERS

There is no similar shuffling operation in MPI because MPI doesn’t group data into Key-Value objects. In Hadoop MapReduce framework, shuffling operation relies on the distributed file system and causes repetitive merges and disk access. As this could be very inefficient, in Twister iterative MapReduce, we leverage memory to do shuffling operation by directly transferring intermediate data through the network from memory to memory between Map task and Reduce tasks. In implementation, due to the poor reliability and scalability of the brokers, we turn to use direct TCP transfers instead of relying on brokers to send intermediate data.

In Twister, each Map task is located in a daemon process and executed by a thread. Once a Key-Value pair is output from a Map task, it is hashed according to the key and regrouped according to the destination, i.e., the node ID where the Reduce task which is selected to process this key locates. The reduce task selection can be redefined by the user but the default implementation is based on the key’s hash code and modulo operation. When a Map task finishes, it sends out all the Key-Value pairs it collects. There are two different kinds of routes. If the data size is small, e.g. less than 1MB, they are sent through the broker network as messages. Otherwise, a small control message which contains the metadata information of the real data is sent through brokers to the daemon process where the Reduce task resides. Then it processes the message and fetches the real data by using direct TCP transfers.

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

The performance of shuffling mainly depends on the size of intermediate data. As the data size increases, the performance degrades drastically. For example, in the image clustering application, the data required to be transferred in shuffling is about bytes, is the number Map task threads per node, is the number of the node, and is the data per Map task. Therefore, even if the data per task is small, as long as and are large, the program can generate large intermediate data. We reduce the intermediate data size by using local reduction across Map tasks. To support local reduction, we provide appropriate interface to help users define the reduction operation.

## **Local Reduction**

The current memory-based shuffling mechanism in Twister is efficient compared with original disk-based shuffling mechanism. However, in big data processing, the data transferred in the shuffling stage is incredibly large and the number of links can be used for data transmission is limited, therefore the cost of shuffling is very high and the whole process is unstable. Some solutions try to use Weighted Shuffle Scheduling (WSS) [18] to balance the data transfers by making the number of transferring flow to be proportional to the data size. But for K-means Clustering, But this won’t help our application because the data size generated per Map task is same.

We notice that each Key-Value pair in intermediate data is a partial sum of the coordination values of data points in a cluster. Since addition is an operation with both commutative and associative properties, for any two values belonging to the same key, we can do addition on them and merge them to a single Key-Value pair and this doesn’t change the final result. This property can be illustrated by the following formula:

(4)

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

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

## **Interface Support**

To support shuffling and local reduction, we provide new interfaces to allow users define the Key and Value objects and operation. We abstract data presentation through interface Key and Value extended from TwisterSerializable, which defines the interface for object serialization. In interface Key, an API named isMergeableInShuffle is defined to check if the current Key-Value pair can be merged in shuffling. At the same time, an API mergeInShuffle is defined in interface Value. It can take a Value object as a parameter and merge the data to the current Value object (See Table 7).

**Table 7. New interfaces of “Key” and “Value”**

|  |
| --- |
| **Interface “Key”** |
| public interface Key extends TwisterSerializable {  public boolean equals(Object key);  public int hashCode();  public boolean isMergeableInShuffle();  } |
| **Interface “Value”** |
| public interface Value extends TwisterSerializable {  public void mergeInShuffle(Value value);  } |

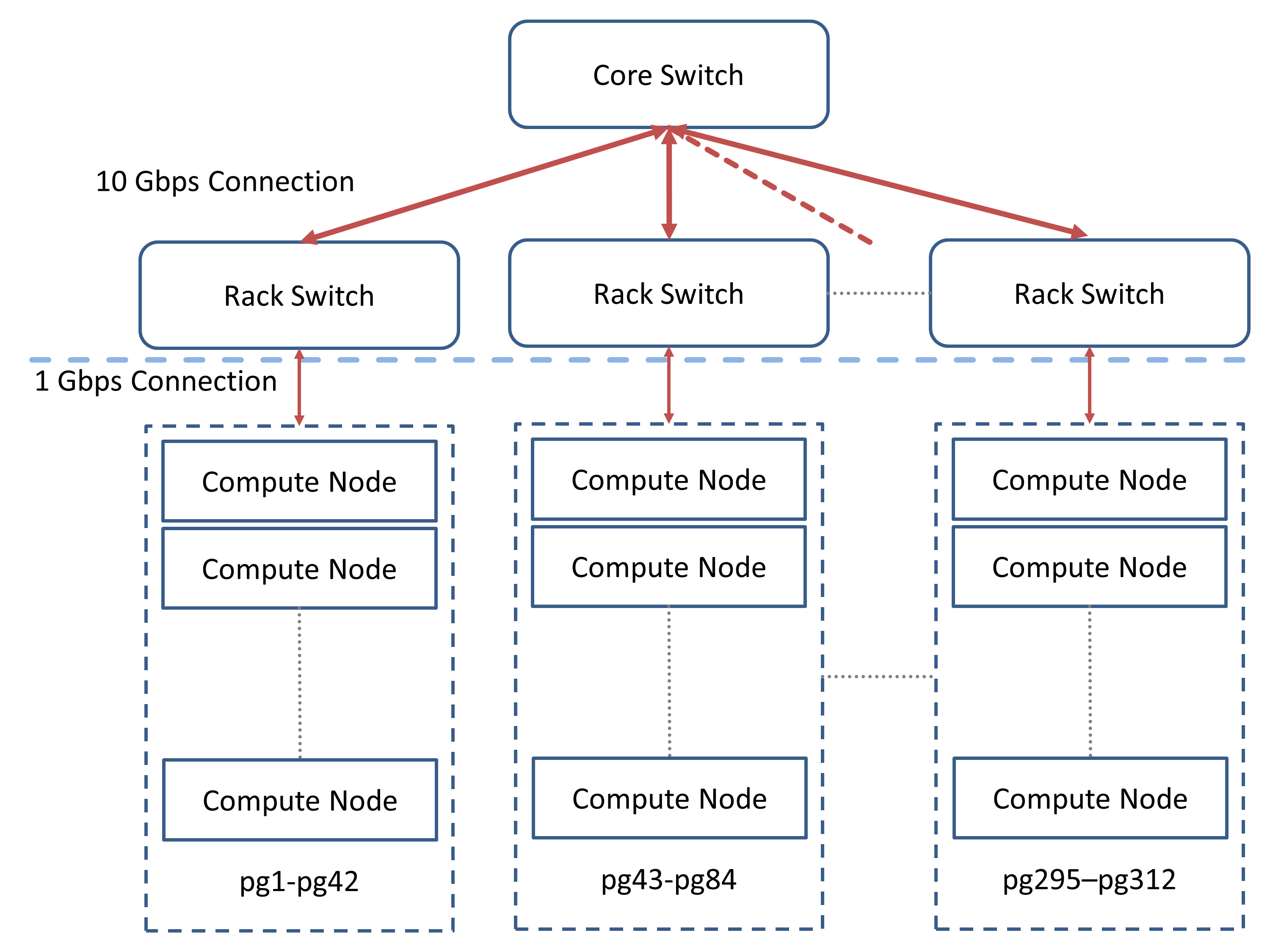
# Experiments

To evaluate the performance of new collective communication methods proposed, we conduct experiments about broadcasting and shuffling on IU PolarGrid cluster in the context of micro-benchmarking and application benchmarking. The results demonstrate that chain method achieves the better performance on big data broadcasting compared with MPI methods and shuffling with local reduction can out-perform the original shuffling significantly.

## **IU PolarGrid**

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 connections and the second level of 1 GigE connections (See Figure 3). For computing capacity, each compute node in PolarGrid uses a 4-core 8-thread Intel Xeon CPU E5410 2.33 GHz processor. The L2 cache size per core is 12 MB. Each compute node has 16 GB total memory.

This kind of topology can easily generate contention when there are many inter-switch communication pairs. The bottleneck is that inter-switch communication is through the one and only core switch and the connection is also limited to 10 GigE. Assuming that every 1 GigE link under the leaf switch is fully utilized, a 10 GigE connection can only support 10 parallel communication pairs across two leaf switches in maximum. If there are more inter-switch communication pairs between any two leaf switches, they could affect each other in performance. As a result, reducing the number of inter-switch communication times is considered the highest priority in the design of efficient communication algorithms under fat-tree topology.



**Figure 3. Fat-Tree topology in IU PolarGrid**

## **Broadcasting**

We test several broadcasting methods on IU PolarGrid: chain method in Twister, MPI\_BCAST in Open MPI 1.4.1 [25], and broadcasting method in MPJ Express 0.38 [26]. We also compare the current Twister chain broadcasting method with other designs such as chain method without topology awareness and naïve broadcasting 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 return of the calling. 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. Since there are only milliseconds of differences between execution times we don’t show the error bar in charts.

**Figure 4. Performance comparison of Twister chain method and MPI\_Bcast**

Figure 4 shows the comparison between chain method and MPI\_BCAST method. 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 real performance of the chain method can achieve near constant execution time. Besides, the new method achieves 20% better performance than MPI\_BCAST.

**Figure 6. Chain method with/without topology-awareness**

**Figure 5. Performance comparison of Twister chain method and MPJ broadcasting method (MPJ 2GB is prediction only)**

Figure 5 shows the comparison between Twister Chain method and broadcasting method in MPJ. Due to exceptions, we couldn’t launch MPJ broadcasting on 2GB data. So we draw a dash line to mark the prediction. Since 1GB MPJ broadcasting uses twice the time of 0.5GB MPJ broadcasting, we assume 2 GB MPJ broadcasting also costs double time of 1 GB MPJ broadcasting. MPJ broadcasting method is also stable as the number of processes grows, but it is pretty slow. Twister chain broadcasting is only about 25% of the time cost in MPJ broadcasting. Besides, there is a significant gap between 1-node broadcasting and 25-node broadcasting.

However if the chain sequence is randomly generated but not topology-aware, the performance degrades quickly as the scale grows. Figure 6 shows that chain method with topology-awareness is 5 times faster than time of the chain method without topology-awareness. For broadcasting in 1 switch, we see there is no much difference between two methods. However, as the number of nodes and the number of switches 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.

We show the performance of naïve broadcasting (sequential sending) and compare it with Twister chain method in Table 8. Since naïve broadcasting takes very long time, we don’t present a chart here. The purpose is to show the baseline of broadcasting performance in IU PolarGrid. Because of 1 Gb connection on each node, we see the transmission speed is about 8 s/GB which matches the setting of the bandwidth. If we don’t use any method to do broadcasting efficiently, the overhead of broadcasting is tremendous in this platform.

**Table 8. Comparison between Twister chain broadcasting time and Naïve Broadcasting**

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

By looking inside Twister chain method, we also examine the potential affect from socket buffer size. As what we mention above in Section 2.5, small socket buffer could cause slow-down of the sender. We take broadcasting 1 GB data on 125 nodes as an example and increase the socket buffer size gradually from 8KB to 1MB. We find that when buffer size is 8 KB, the performance is the worst of all. Then as the buffer size grows the time cost gets lower. When the buffer size is larger than 128 KB, we get the best performance and stable execution time. The experiment shows that as what is analyzed above, the socket buffer size can affect the performance a lot because TCP window cannot open up fully when buffer size is small. With proper buffer size, the broadcasting performance can be improved by 90% (See Figure 7).

Serialization and de-serialization are necessary steps to provide byte data array format required by broadcasting operation. We measure the time cost of these steps in Figure 8. We see the cost of serialization and de-serialization both are very high. We notice that serialization and de-serialization operations are sensitive to data types. For the same-sized data, “byte” type data uses more time to serialize and de-serialize than “double” type data. And for “byte” data, de-serialization even uses longer time than serialization. For image clustering application, we use “byte” to store broadcasting data in order to reduce the data size. As a result, the time cost on broadcasting is only about 10% of the total broadcasting time cost. Other 90% is spent on serialization and deserialization. Since these operations are required steps and they are local operation with stable time cost, currently we don’t have special optimization for them yet.

**Figure 8. Serialization, broadcasting and de-serialization**

**Figure 7. Chain method performance under different socket buffer sizes**

## **Shuffling**

To benchmark the performance of shuffling, we choose the following settings to run the image clustering application. For job settings, we choose 125 nodes, a relatively large scale to run the application with 1000 Map tasks and 125 reduce tasks. For data settings, we keep the number of centroids to 500K and focus on testing the performance of collective communication. Since 500K centroids can generate about 1 GB intermediate data per task, we can see the overhead from shuffling obviously. We measure the total time from the start of shuffling to the end of Reduce phase because reducers start asynchronously. Time costs on Reduce tasks are included but averagely it is just around 1 second and is negligible compared with the data transfer time.

**Figure 10. Shuffling with non-empty centroid blocks output per Map task**

**Figure 9. Shuffling with full centroid blocks output per Map task**

We also try to reduce the intermediate data size from the application side in image clustering. To distribute the workload of 125 Reduce tasks, we chunk the output from each Map task to 125 blocks and send them to different reducers. So for a block without sum update from local vectors, we don’t need generate this block in the output. In this way, we can reduce the output from Map tasks a little. However, this happens randomly and depends on what kinds of vectors are cached in the Map tasks and the vector values of the centroids.

**Figure 11. Communication cost per iteration of the image clustering application**

As a result, we execute image clustering application with shuffling benchmarking in two different modes. In the first mode, we force each Map task output all the blocks. Figure 9 shows the time difference of shuffling with or without local reduction in this mode. Without using local reduction, the output per node is 8 GB and the total data for shuffling is about 1 TB, after using local reduction, 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. In the second execution mode, we don’t force Map tasks output all the blocks but only allow non-empty blocks are outputted. The original intermediate data before shuffling is around 300 GB. Figure 10 shows that time cost on shuffling is still reduced by 55% and the total collective communication cost per iteration is still reduced by 40%.

## **Image Clustering Application**

Finally we present a real execution of the image clustering application here. We successfully cluster 7420000 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 500 MB. Shuffling data before local reduction is 20 TB, while the data size after local reduction is about 250 GB. Since the total memory size on 125 nodes is 2 TB, we even cannot execute the program if no local reduction. Figure 11 presents the main collective communication cost per iteration, which is 169 seconds (less than 3 minutes). We're developing a new fast Kmeans algorithm which will be presented as a separate work, and can drastically reduce hour-long computation time in Map stage.

# 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 [21], 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 [17]. And some solution also provides auto algorithm selection [27].

However, many solutions have a different focus from our work. Some of them only study small data transfers up to megabytes level [17][28]. Some solution relies on special hardware support [19]. The data type is typically vectors and arrays whereas we are considering objects. Many algorithms such as “allgather” have the assumption that each node has the same amount of data [17][19], 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 [18] is such a global control service and architecture to manage intra and inter-transfer activities on Spark [29]. 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 [30] like protocol called Cornet, augmented by topology detection. Although this method achieves similar performance as our chain method, it is still unclear about its internal design and details of communication graph formed in data transfer and we will compare it with our methods in future. For shuffling, it uses weighted shuffle Scheduling (WSS) to set the weight of the flow to be proportional to the data size.

Hadoop-A [31] 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 [32] is a MapReduce like framework for EC2 [33] and S3 [34]. 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 [35] iHadoop [36] 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.

# CONCLUSION

In this paper, 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. Compared with the naive algorithm, the new algorithm reduces the time cost of broadcasting by at least a factor 120 over 125 nodes. It reduces 20% cost than MPI methods and 80% of the cost than un-optimized pipeline-based method over 150 nodes. The shuffling cost with local reduction is only 10% of the original time. In summary, the acceleration of broadcasting communication has significantly improved the intermediate data transfer for large scale image clustering problems.

There are a number of directions for future work. We will apply the new Twister framework to other iterative applications [37]. 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 [38] to provide coordination and fault detection. We are also planning to improve K-means clustering algorithm in the image clustering application based on [40] [41].

# 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. 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. MPI Forum, “MPI: A Message Passing Interface,” in Proceedings of Supercomputing, 1993.
7. PolarGrid. http://polargrid.org/polargrid.
8. N. Dalal, B. Triggs. Histograms of Oriented Gradients for Human Detection. CVPR. 2005
9. 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.
10. ActiveMQ. http://activemq.apache.org/
11. S. Pallickara, G. Fox, NaradaBrokering: A Distributed Middleware Framework and Architecture for Enabling Durable Peer to-Peer Grids, Middleware 2003, 2003.
12. Infiniband Trade Association. http://www.infinibandta.org.
13. 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
14. Ford L.R. Jr., Fulkerson D.R., Maximal Flow through a Network, Canadian Journal of Mathematics , 1956, pp.399-404.
15. HDF5, http://www.hdfgroup.org/HDF5/whatishdf5.html
16. NetCDF, http://www.unidata.ucar.edu/software/netcdf/
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. Mosharaf Chowdhury et al. Managing Data Transfers in Computer Clusters with Orchestra, Proceedings of the ACM SIGCOMM 2011 conference, 2011
19. Nikhil Jain, Yogish Sabharwal, Optimal Bucket Algorithms for Large MPI Collectives on Torus Interconnects, ICS '10
20. Proceedings of the 24th ACM International Conference on Supercomputing, 2010
21. 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
22. Watts J, van de Geijn R. A pipelined broadcast for multidimensional meshes. Parallel Processing Letters, 1995, vol.5, pp. 281–292.
23. Charles E. Leiserson, Fat-trees: universal networks for hardware efficient supercomputing, IEEE Transactions on Computers, vol. 34 , no. 10, Oct. 1985, pp. 892-901.
24. Radhika Niranjan Mysore, etc. PortLand: A Scalable Fault-Tolerant Layer 2 Data Center Network Fabric, SIGCOMM, 2009
25. 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
26. Open MPI, http://www.open-mpi.org
27. MPJ Express, http://mpj-express.org/
28. 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
29. 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.
30. M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster Computing with Working Sets. In HotCloud, 2010.
31. BitTorrent. http://www.bittorrent.com.
32. Yangdong Wang et al. Hadoop Acceleration Through Network Levitated Merge, International Conference for High Performance Computing, Networking, Storage and Analysis (SC'11), 2011
33. 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
34. EC2. http://aws.amazon.com/ec2/.
35. S3. http://aws.amazon.com/s3/.
36. Y. Zhang, Q. Gao, L. Gao, and C. Wang. imapreduce: A distributed computing framework for iterative computation. In DataCloud '11, 2011.
37. 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
38. 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
39. 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.
40. 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.
41. 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,.