**Harp: Collective Communication on Hadoop**

Bingjing Zhang

Indiana University

*Abstract— Big data tools evolve rapidly recent years. At the beginning, there was only MapReduce. But later tools in other different models also emerge. For example, Iterative MapReduce frameworks improves performance of MapReduce job chains through caching. Pregel, Giraph and GraphLab abstract data as a graph and process it in iterations. However, all these tools are designed with fixed data abstraction and computation flow so each of them cannot adapt to all kinds of problems and applications. In this paper, we define Map-Collective model and abstract a collective communication layer to serve the communication demands in different applications* *through providing communication operations on different data abstractions such as arrays, key-values and graphs. . And we design and implement a communication library and plug it into Hadoop so that Hadoop can do in-memory communication without writing intermediate data to HDFS. With improved expressiveness and acceptable performance on collective communication, we can cover many applications from HPC system to Cloud system together.*

# Introduction

It has been 10 years since the publication of Google MapReduce paper [1]. After that, we see its open source version Hadoop [2] has become the mainstream of big data processing and we also see many other tools emerge for processing different big data problems. To accelerate iterative algorithms, original MapReduce model is extended to iterative MapReduce model. Tools such as Twister [3] and HaLoop [4] are in this model and they can cache loop invariant data at local to avoid repeat input data loading in a MapReduce job chain.

Spark [5] also use caching to accelerate iterative algorithms but abstract data as RDDs and defines computation as transformations on RDDs without restricting computation to a chain of MapReduce jobs.

To processing graph data, Google announced Pregel [6]. And later its open source version Giraph [7] and Hama [8] also come out. These tools also use caching to accelerate iterative algorithms but abstract input data not as Key-Value pairs but as a graph and defines communication in each iteration as messaging between vertices along edges.

However, all these tools are in a kind of “top-down” design. The whole programming model, from data abstraction to computation and communication pattern, are fixed. Users have to put their applications into the programming model and could cause performance inefficiency in many applications. For example, in K-Means clustering [9] (Lloyd's algorithm [10]), every task needs all the centroids generated in the last iteration. Mahout [11] on Hadoop chooses to reduce the results of all the map tasks in one reduce task and store it on HDFS. And then this result will be read by all the map tasks in the job at the next iteration. The “reduce” stage can be done more efficiently by chunking intermediate data to partitions and using multiple reduce tasks to reduce each of them. This “(multi-)reduce-gather-broadcast” strategy is also applied in other frameworks but done through in-memory communication, e.g. Twister and Spark. At the same time, Hama uses a different way to exchange intermediate data with all-to-all communication directly.

But “gather-broadcast” is not an efficient way to relocate the new centroids generated by reduce tasks, especially when the centroids data get large in size. The time complexity of “gather” is at least kdβ where k is the number of centroids, d is the number of dimensions and β is the communication time used to send each element in the centroids (communication initiation time α is ignored here). And the time complexity of “broadcast” is also at least kdβ [12] [13]. In sum, the time complexity of “gather-broadcast” is about 2kdβ. But if we use allgather bucket algorithm [14], the time complexity is reduced to kdβ. The direct all-to-all communication used in Hama K-Means clustering has even higher time complexity which is at least pkdβ where p is the number of processes (The potential network conflicts in communication is ignored).

As a result, choosing the correct algorithm for communication is important to an application. In iterative algorithms, communication may happen at least once or more than once in each iteration. This makes communication algorithm performance crucial to the efficiency of the whole application. We call this kind of communication which happen between tasks as “Collective Communication”. Rather than fixing communication pattern, we decide to separate this layer out from other layers and provide collective communication abstraction.

In past, this kind of abstraction has been shown in MPI [15] which is mainly used in HPC systems and supercomputers. The abstraction is well defined but has many limitations. It cannot support high level data abstractions other than arrays and objects and related communication patterns on them, e.g. shuffling on Key-Values or message passing along edges in graph. Besides, the programming model forces users to pay much attention to every detail of a communication call. For example, users have to know the buffer size for data receiving. This is hard to obtain in many applications as the amount of sending data may be very dynamic in each task.

To improve the expressiveness and performance in big data processing, and combining the advantages of big data processing in HPC systems and cloud systems we present Harp library, to provide data abstractions and related communication abstractions in big data processing. The word “harp” symbolizes the effort to make parallel processes cooperate together through collective communication for efficient data processing just as strings in harp can make concordant sound.

Currently Harp is implemented in java and plugged into Hadoop. We make Harp loosely coupled with Hadoop so that we can plug it into other big data tools in future. With Hadoop-Harp, we converts MapReduce model to Map-Collective model and enables efficient in-memory communication between map tasks to avoid HDFS read/write.

With these two important contributions, collective communication abstraction and Map-Collective model, we make Harp neither a replication of MPI nor some other research work which try to transplant MPI into Hadoop system, e.g. [16].

In the rest part of this paper, Section 2 talks about related work. Section 3 discusses the abstraction of collective communication. Section 4 shows how Map-Collective model works in Hadoop. Section 5 gives examples of several applications implemented in Harp. Section 6 shows the performance of Harp through experiments.

# Related Work

In Introduction, we mentioned many names of different big data tools. After years of development and evolution, the world of big data tools is getting bigger and more complicated. Each tool has its own computation flow and data abstraction and try to do optimization for different applications. In general, we can divide tools into four models: DAG (Directed Acyclic Graph) model, MapReduce model, Graph model and BSP/Collective model (See Figure 1).



Figure 1. The world map of big data tools

Before MapReduce Model, MPI is the main tool used to processing big data problem. It is deployed on expensive hardware such as HPC or supercomputers. But MapReduce model tries to use commodity machines to solve big data problems. MapReduce model defines data abstraction as KeyValue pairs and computation flow as “map, shuffle and then reduce”. “Shuffle” operation is a local disk based communication in order to “regroup” and “sort” intermediate data. One advantage of this tool is that it doesn’t relies on memory to load and process all the data but using local disks. So MapReduce model can solve problems where the data size is too large to fit into the memory. Besides, MapReduce model also provides fault tolerance which is important to big data processing. The open source implementation of this model is Hadoop, which largely used nowadays in industry and academia.

Through MapReduce becomes popular due to its simplicity and scalability, it is still slow when running iterative algorithms. Iterative algorithms require a chain of MapReduce jobs which result in repeat data loading in each iteration. Then several frameworks such as Twister [3], HaLoop [4] and Spark [5] solve this problem by caching intermediate data.

Another model used for iterative computation is graph model. Google released Pregel [6] to run iterative graph algorithms. Pregel abstracts data as vertices and edges. Each worker caches vertices and related out-edges as graph partitions. Computation happens vertices. In each iteration, the results are sent as messages along out-edges to neighbor vertices and processed by them at the subsequent iteration. The whole parallelization is BSP style. There are two open source projects following Pregel’s design. One is Giraph [7] and another is Hama [8]. Giraph also follows iterative BSP graph computation pattern but Hama tries to build a general BSP computation model.

Meanwhile GraphLab [17] [18] does graph processing differently. It abstracts graph data as data graph and uses consistency models to control vertex value update. And later GraphLab is enhanced with PowerGraph [19] abstraction to reduce the communication overhead. It uses “vertex-cut” graph partitioning with GAS (Gather-Apply-Scatter) computation model. Later, this is also learned by GraphX [20].

Other than tools for iterative computation, there are also big data tools designed for query and stream processing. These tools mainly use DAG model. DAG model abstracts computation flow as a directed acyclic graph. Each vertex in the graph is a process, and each edge is a communication channel between two processes. As a result, DAG model is helpful to those applications which has complicated parallel workflow. Tools using DAG model often use other tools in MapReduce model as lower level parallel engine. For example, Hive [21] is a tool designed to do SQL-like query on big data. It uses Hadoop as parallel engine and builds DAGs on MapReduce jobs. For other tools in DAG model, some are like Hive and are used for query, such as Dryad [22], Pig [23], Tez [24], Shark [25], MRQL [26]. Others are for stream processing, such as S4 [27], Storm [28], Samza [29], and Spark Streaming [30].

Because each tool is designed in different models and for different purposes, it is difficult for users to pick up the correct tool for their applications. As a result, model composition becomes the current trend in the design of big data tools. For example, Spark’s RDD data abstraction and the transformation operations on RDD are very similar to MapReduce model. But it organizes computation tasks as DAGs. Stratosphere [31] and REEF [32], these two projects also try to include different models in one framework.



Figure 2. Hierarchical Data Abstraction and Collective Communication Operations

However, no matter what kind of tools you use, communication is hidden in all these tools and coupled with the computation flow. Then the only way to improve the communication performance is to reduce the size of intermediate data [19]. But based on the research in MPI, we know that some communication operations can be optimized through changing the communication algorithm. We already give one example in Introduction. Here we take “broadcast” as another example. “Broadcast” is done through simple algorithm (one-by-one sending) in Hadoop. But it is optimized using BitTorrent technology in Spark [33] or using pipeline based chain algorithm in Twister [12] [13].

Though these research work [12][13][33][34] try to add or improve collective communication operations, they are still limited in types and constrained by the computation flow. As a result, it is necessary to build a separated communication layer abstraction. With this layer of abstraction, we can provide a rich set of operations and give users flexibility in choosing suitable operations to their applications.

A common question is that why we don’t use MPI directly which already provided collective communication abstraction. There are many reasons. The collective communication in MPI is still limited in abstraction. MPI provides a low level data abstraction on arrays and objects so that many collective communication operations used in other big data tools are not provided in MPI. Besides, MPI doesn’t provide computation abstraction so that writing MPI applications is difficult compared with writing applications on other big data tools. Thirdly, MPI is deployed on HPC or supercomputers. Despite of projects like [16], it is not well integrated with cloud environment such as Hadoop ecosystems.

# Collective Communication Abstraction

To support different types of communication patterns in big data tools, we abstract data types in a hierarchical way. Then we define collective communication operations on top of the data abstractions. To improve the efficiency of communication, we also add memory management support for data caching and reuse.

## Hierarchical Data Abstraction

The data abstraction has 3 categories in horizontal direction and 3 levels in vertical direction (Figure 1). Horizontally, data are abstracted as arrays, Key-Values or vertices, edges and messages in graphs. In vertical direction, we build abstraction from basic types, to partitions and tables.

Firstly, any data which can be sent or received is an implementation of interface Commutable. At the lowest level, there are two basic types under this interface: arrays and objects. Based on the component type of an array, currently we have byte array, int array, long array and double array. From object type, to describe graph data, we have vertex object, edge object and message object; to describe Key-Vale pairs, we have key object and value object.

Next, at the middle level, basic types are wrapped as partitions such as array partitions, Key-Value partitions and graph data partitions: edge partition, vertex partition and message partition. Noticing that we follow the design from Giraph, edge partition and message partition are built with byte arrays not on edge object or message object directly. When reading, bytes are converted to edge object or message object. When writing, edge object or message object are serialized and written to byte arrays.

At the top level, partitions are put into tables. Table is an abstraction which contains several partitions. Each partition in a table has a unique partition ID. If two partitions with the same ID are added to the table. It will solve the ID conflict, either combine or merge two partitions into one partition. Tables are associated with each other through table IDs. Partitions inside of tables on different workers which share the same table ID are considered as one dataset participating in the collective communication. Collective communication is defined as redistribution or consolidation of partitions between tables sharing the same table ID on several workers. For example, in Figure 2, a set of tables associated with ID 0 is defined on workers from 0 to N. Partitions from 0 to M are distributed among these tables. A collective communication operation on table 0 is to move these partitions between these tables. We will talk about more in details about the behavior of partition movement in different collective communication operations.

Table 1. Collective Communication Operations and the Data Abstractions Supported (“√” means “supported” and “implemented” and “○” means “supported” but “not implemented)

|  |  |  |  |
| --- | --- | --- | --- |
| Operation Name | Array  Table | Key-Value Table | Graph  Table |
| Broadcast | √ | ○ | √ (Vertex) |
| Allgather | √ | ○ | √ (Vertex) |
| Allreduce | √ | ○ | ○ (Vertex) |
| Regroup | √ | √ | √ (Edge) |
| Send all messages to vertices |  |  | √ |
| Send all edges to vertices |  |  | √ |

## Collective Communication Operations

Collective communication operations are defined on top of the data abstractions. Currently three categories of collective communication operations are supported.

1. Collective communication inherited from MPI collective communication operations such as broadcast, allgather, allreduce.
2. Collective communication inherited from MapReduce “shuffle-reduce” operation, e.g. “regroup” operation with “combine or reduce” support.
3. Collective communications abstracted from graph communication, such as regrouping vertices or edges, moving edges to vertices and sending messages to vertices.

Some collective communication operations tie to certain data abstractions. For example, graph collective communication operations from the third category have to be done on graph data. But for other operations, the boundary is blur. For example, “allgather” operation can be used on array tables, key-value tables, and vertex tables. But currently we only implemented it on array tables and vertex tables. We haven’t seen applications which use Key-Value abstraction also need “allgather” operation. The following is a table which summarizes all the operations identified from applications and related data abstractions (See Table 1). We continue adding other collective communication operations not shown in this table in future.

Here we take another look of Figure 2 and use “regroup” as an example to see how it works on array tables. Similar to MPI, for N + 1 workers, workers are ranked from 0 to N. Here Worker 0 is selected as master worker which collects the partition distribution information on all the workers. Each worker reports the current table ID and the partition IDs it owns. Table ID is used to identify if the collective communication is on the same data set. Once all the partition IDs are received, master worker decides the destination worker IDs of partitions. Usually the decision is done through modulo operation. Once the decision is made on master, the result is broadcasted to all workers. After that each worker starts to send partitions out and receive partition from other workers (See Figure 3).

Each collective communication can be implemented in many different algorithms. This has been talked in many papers [12][13][14]. For example, we have two implementations of “allreduce”. One is “bidirectional-exchange algorithm” [14] and another is “regroup-allgather algorithm”. When the data size is large and each table has many partitions, as what we discussed in Section I, “regroup-allgather” is more suitable because it has less data sending and more balanced workload on each worker. But if the table on each worker only has one or a few partitions, “regroup” cannot help much while “bidirectional-exchange” is more effective.

Besides, we also optimize the implementation of several collective communication operations when the partition distribution is known in the application context. Normally just like what we show in Figure 3, master worker has to collect the partition distribution on each worker and broadcast this information to each worker and let them know what partition to send and what partition to receive. But when the partition distribution is known, the step of information collection can be skipped. For example, we provide an implementation of “allgather” when the total number of partition is known. In general, we enrich Harp collective communication library with providing different implementations for each operation so that users can choose the proper one based on the application requirement.



Figure 4. The abstraction of tables and partitions

## Implementation



Figure 3. The process of regrouping array tables

To make the collective communication abstraction work, we design and implement several components on each worker to send and receive data. These components are resource pool, receiver and data queue. Resource pool is a very important component in collective communication and iterative computation. In iterative algorithms, the collective communication operations are called repeatedly in iterations. The intermediate data between iterations are similar in size but just with different contents. Resource pool caches the data used in last iteration to enable reuse in next iteration so that the program can avoid repeating memory allocation and lower down the time used on garbage collection.

The process of sending is as follow: the worker firstly serialize the data to a byte array fetched from a resource pool and then send it through the socket. Receiving is managed by a component named receiver. Receiver starts a thread to listen to the socket requests. For each request, receiver spawns another handler thread to process it. We use “producer-consumer” model to process the data received. Handler threads send the data received to the data queue. Then the main thread of the worker which controls the collective communication fetches data from the queue and examine if it is the data belongs to this round of communication. If yes, the data is removed from the queue otherwise it will be put back to the queue again (See Figure 4).

# Map-Collective Model



Figure 5. The mechanism of receiving data

The collective communication abstraction we proposed is designed to run in a general environment with a set of parallel Java processes. Each worker only needs a list of all workers’ locations to start the communication. As a result, this work can be used to improve collective communication operations in existing big data tools. But since communication is hidden in these tools, the applications still cannot be benefited from the expressiveness of collective communication. Here we propose Map-Collective model which is transformed from MapReduce model to enable using collective communications in map tasks. In this section, we are going to talk about several features of Map-Collective model.

## Hadoop Plugin

Harp is designed as a plugin in Hadoop. Currently it supports Hadoop-1.2.1 and Hadoop-2.2.0. To install Harp library, users only need to put Harp jar package to the Hadoop directory. For Hadoop 1, user need to configure the job scheduler to the scheduler designed for Map-collective jobs. But in Hadoop 2.0, because YARN resource management layer and MapReduce framework are separated, users are not required to change the scheduler. Instead, user just need to set "mapreduce.framework.name" to "map-collective" in client job configuration. Harp will launch a specialized application master to request resources and schedule Map tasks.

## MAP-COLLECTIVE INTERFACE

Table 2. “mapCollecitve” interface

|  |
| --- |
| protected void mapCollective(  KeyValReader reader,   Context context)   throws IOException,   InterruptedException {  // Put user code here…  } |

In Map-Collective model, user-defined mapper classes are extended from class CollectiveMapper which is extended from original class Mapper. In CollectiveMapper, user can override a method “mapcollective” with application code. This mechanism is just like the override of “map” method in Class Mapper. But “mapcollective” method uses KeyValReader as parameter but not a single pair of key and value. KeyValReader provides the flexibility to users so that they can either read all key-values into the memory and cache them or read them part by part to fit the memory constraint (See Table 2).

Table 3 Allgather code example

|  |
| --- |
| // Generate array partitions  List<ArrPartition<DoubleArray>>  arrParList = new ArrayList<  ArrPartition<DoubleArray>>();  Random rand = new Random();  for (int i = workerID;   i < numPartitions; i += numMappers){  DoubleArray array = new DoubleArray();  double[] doubles =   pool.getDoubleArrayPool().  getArray(arrSize);  array.setArray(doubles);  array.setSize(arrSize);  for (int j = 0; j < arrSize; j++) {  doubles[j] = rand.nextDouble();  }  arrParList.add(  new ArrPartition<DoubleArray>(  array, i)); }  // Define array table  ArrTable<DoubleArray, DoubleArrPlus>   arrTable =   new ArrTable<  DoubleArray, DoubleArrPlus>(  0, DoubleArray.class,   DoubleArrPlus.class);  for (ArrPartition<DoubleArray> arrPar :  arrParList) {  arrTable.addPartition(arrPar);  }  allgatherTotalKnown(  arrTable, numPartitions); |

CollectiveMapper initializes all the components required in collective communication. User can invoke collective communication calls directly in “mapcollective” method. We also expose current worker ID and resource pool to users. Here is an example of how to do “allgather” in “mapcollective” method.

Firstly we generate several array partitions with arrays fetched from resource pool and add these partitions into an array list. The total number of partitions on all the workers is specified by numPartitions. Each worker has numPartition / numMappers partition (we assume numPartitions % numMappers = 0). Then we add these partitions in an array table and then invoke “allgather”. DoubleArrPlus is the combiner class used in array table in case of partition ID conflict in partition receiving. The “allgather” method used here is called “allgatherTotalKnown”. This version of “allgather” allows all the workers send out all the partitions it owns to neighbors directly with bucket algorithm. Because the total number of partitions in provided as a parameter. Workers don’t need to negotiate the number of partitions to receive on each worker (See Table 3).

## BSP Style Parallelism

To enable in-memory collective communication between workers, we need to make every worker alive simultaneously. As a result, instead of dynamic scheduling, we use static scheduling. Workers are separated to different nodes and collective communication are done between workers iteratively. Then the whole parallelism follows BSP (Bulk Synchronous Parallel) pattern.



Figure 6. Parallelism and Architecture

Here we use our Harp implementation in Hadoop-2.2.0 as an example to talk about the scheduling mechanism and initialization of the environment. The whole process is similar to the process of launching MapReduce jobs in Hadoop-2.2.0. We mentioned that in job configuration at client side, users need to set "mapreduce.framework.name" to "map-collective". This means the system chooses MapCollecitveRunner as job client instead of default YARNRunner for MapReduce jobs. Then MapCollecitveRunner launches MapCollctiveAppMaster to the cluster. MapCollectiveAppMaster is similar to MRAppMaster because both of them are responsible for requesting resources and launching tasks. When MapCollectiveAppMaster request resources, it specifies the location of each task. We try to distribute tasks to different nodes because we use multi-threading in each task so that it is unnecessary to put multiple tasks on a single node.

In launching stage, MapCollectiveAppMaster records the location of each task and generate two lists. One contains the locations of all the workers and another contains the mapping between map task IDs and worker IDs. These files currently are stored on HDFS and shared to all the workers. To ensure every worker has started, we use a “handshake”-like mechanism to synchronize these workers. In the first step, the master worker tries to ping each worker with sending a message. It will try several times if sending fails. In the second step, slave workers who received the message will send a response message back to let the master knows its existence. In the third step, once the master gets all the responses, it broadcasts a small message to all the workers and notify them the success of the initialization.

When the initialization is done, each worker invokes “mapCollective” method to do computation. We use hybrid model in parallelism and allows multi-threading in each worker. The interface designed to launch multithread tasks is called “doTasks”. With providing a Task object with user-defined “run” method and related input partitions. Harp can automatically do multi-threading parallelization and return the outputs.

## Fault Tolerance

We separate fault tolerance issue to two. One is fault detection and another is fault recovery. Currently our effort is to ensure every worker can report exception or fault correctly without hanging there. With careful implementation and based on the results of testing, this issue is solved.

However, fault recovery is very difficult because the execution flow in each worker is very flexible. Currently we do job level fault recovery. Based on the scale of time length of execution, a jobs with large number of iterations are separated into a small number of jobs each of which contains several iterations. This can naturally form check-pointing between iterations. Because Map-Collective jobs are very efficient on performance, this method is feasible without generating large overhead. At the same time, we are also investigating task-level recovery by re-synchronizing new launched tasks with other old live tasks after fault.

# Applications

We give 3 application examples here to show how these applications are implemented in Harp. These applications are K-Means clustering, Force-directed Graph Drawing Algorithm, and Weighted Deterministic Annealing SMACOF. The first two algorithms are very simple. Both of them use a single collective communication operations during iterations. The first algorithm uses array abstraction but the second one uses graph abstraction. The third application also uses array abstraction but is much more complicated than the other two. It has nested iterations and we use two different collective communication operations alternately. For applications with KeyValue abstraction, we only implemented WordCount. Because it is very simple, we don’t spend paragraphs here to introduce it.

## K-Means Clustering

K-Means Clustering is an algorithm to cluster large number of data points to a predefined set of clusters. We use Lloyd's algorithm [10] to implement K-Means Clustering in Map-Collective model.

In Hadoop-Harp, each worker loads a part of the data points and caches them into memory as array partitions. The master worker loads the initial centroids file and broadcast it to all the workers. Later, in each iterations, each worker calculates its own local centroids and then use “allreduce” operation at the end of the iteration to produce the global centroids of this iteration on each worker. After several iterations, the master work will write the final version of centroids to HDFS.

We use pipeline based method to do broadcasting for initial centroids distribution [12]. For “allreduce” in each iteration, due to the large size of intermediate data, we split “allreduce” to three stages. Firstly we do “regroup” on intermediate data based on partition IDs. Next, on each worker we reduce the partitions with the same ID into a partition of new centroids. Finally, we do “allgather” on new generated data to let every worker has all the new centroids.

## Force-directed Graph Drawing Algoritm

We implement Hadoop-Harp version of Fruchterman-Reingold algorithm which produces aesthetically-pleasing, two-dimensional pictures of graphs by doing simplified simulations of physical systems [35].

Vertices of the graph are considered as atomic particles. At the beginning, vertices are randomly placed in a 2D space. The displacement of each vertex is generated based on the calculation of attractive and repulsive forces on each other. In each iteration, the algorithm calculates the effect of repulsive forces to push them away from each other, and then calculates attractive forces to pull them close to each other, and finally limit the total displacement by the temperature. Both attractive and repulsive forces are defined as functions of distances between vertices following Hook’s law.

In Hadoop-Harp implementation, graph data are stored as partitions of adjacency lists in files and then are loaded into edge tables and partitioned based on the hash values of source vertex ID. We use “regroupEdges” operation to move edge partitions with the same partition ID to the same worker. We create vertex partitions based on edge partitions. These vertex partitions are used to store displacement of vertices calculated in one iteration.

The initial vertex positions are generated randomly. We store them in another set of tables and broadcast them to all workers before starting iterations. Then in each iteration, once displacement of vertices are calculated, new vertex positions are generated. Because the algorithm requires to calculate the repulsive forces between every two vertices, we use “allgather” to redistribute the current positions of all the vertices to all the workers instead of sending messages along edges as what Pregel/Giraph does. By combining multiple collective communication operations, we show the flexibility of Hadoop-Harp in implementing different applications.

## Weighted Deterministic Annealing SMACOF

Generally, Scaling by MAjorizing a COmplicated Function (SMACOF) is a gradient decent type of algorithm which is widely used for large-scale Multi-dimensional Scaling (MDS) problems [36]. The main purpose of this algorithm is to project points from high dimensional space to 2D or 3D space for visualization by providing pair-wise distances of the points in original space. Through iterative stress majorization, the algorithm tries to minimize the difference between distances of points in original space and their distances in the new space.

The algorithm we implement in Hadoop-Harp is call Weighted Deterministic Annealing SMACOF (WDA-SMACOF). Here “weight” is assigned for every pair of point distances to avoid low quality of dissimilarity values and “deterministic annealing” is used to find global optima of an optimization process instead of local optima by adding a computational temperature to the target object function [36].

The algorithm has nested iterations. In every iteration of outer iterations, we firstly calculate matrix B, and then calculate X (the coordination values of points on the target dimension space) through conjugate gradient process. Finally we calculate the stress value of this iteration. Inner iterations are in conjugate gradient process to solve X.

Originally the algorithm is implemented in Twister [36]. The three different computations in outer iterations are separated into three MapReduce jobs and ran alternatively. There are two flaws in this implementation. One is that the static data cached in jobs cannot shared to each other, therefore there is duplication in caching and it causes high memory usage. Another is that, the results from the last job has to be collected back to the client and broadcast to the next job. This process is inefficient and can be replaced by collective communication calls.

In Hadoop-Harp, we improve the parallel implementation use “allgather” and “allreduce” two collective communication operations. Conjugate gradient process uses “allgather” to collect the results from matrix multiplication and “allreduce” for the result from inner product calculation. In outer iterations, “allreduce” is used to sum the result of stress value calculation. We use bucket algorithm in “allgather” and bi-directional exchange algorithm in “allreduce”.

# Experiments

K-Means Clustering

On Madrid 8 nodes (The results in January, v.s. Hadoop, v.s. MPI)

Performance comparison between Harp and original Hadoop is done on Madrid cluster which contains 8 nodes (Figure 2). 10 iteration K-means clustering under different problem sizes are used as test examples. Execution time is reduce by about 40% (10 million 3D points and 500 centroids) ~ 84% (1 million 3D points and 50 thousand centroids).

Graph layout algorithm

On Madrid (1, 2, 4, 8)

WDA-MDS

On BigRed II [37] (8, 16, 32, 64, 128, 100k, 200k, 300k, 400k)

PERFORMANCE

Application performance

Benchmark performance

# CONCLUSION

In this paper, after analyzing communication in different big data tools, we abstract collective communication layer from execution model. Then with this abstraction, we build Map-Collective model to improve the performance and expressiveness of big data applications.

We implement collective communication abstraction and Map-Collective model in as Harp plugin in Hadoop. With Hadoop-Harp, we present three different big data applications: K-Means Clustering, Force-directed Graph Drawing and WDA-SMACOF. We show how different application can be expressed in different abstractions and how the collective communication operations can be combined together to run application efficiently. Through experiments, we show the excellent performance on 8 nodes Madrid HPC and 100+ nodes on BigRed II supercomputer.

# REFERENCES

1. J. Dean and S. Ghemawat. “Mapreduce: Simplified data processing on large clusters.” OSDI, 2004.
2. Apache Hadoop. <http://hadoop.apache.org>
3. J. Ekanayake, H. Li, B. Zhang, T. Gunarathne, S.-H, Bae, J. Qiu, G. Fox. “Twister: A Runtime for iterative MapReduce.” Workshop on MapReduce and its Applications, HPDC, 2010.
4. Y. Bu, B. Howe, M. Balazinska, and M. Ernst. “Haloop: Efficient Iterative Data Processing on Large Clusters”. VLDB, 2010.
5. M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. “Spark: Cluster Computing with Working Sets”. HotCloud, 2010.
6. Grzegorz Malewicz, et al. “Pregel: A System for Large-scale Graph Processing”. SIGMOD. 2010.
7. Apache Giraph. <https://giraph.apache.org/>
8. Apache Hama. <https://hama.apache.org/>
9. J. MacQueen, “Some Methods for Classification and Analysis of MultiVariate Observations.” Berkeley Symp. on Mathematical Statistics and Probability, 1967.
10. S. Lloyd. “Least Squares Quantization in PCM”. IEEE Transactions on Information Theory 28 (2): 129–137, 1982.
11. Apache Mahout. <https://mahout.apache.org/>
12. J. Qiu, B. Zhang, “Mammoth Data in the Cloud: Clustering Social Images.” In Clouds, Grids and Big Data, IOS Press, 2013.
13. B. Zhang, J. Qiu. “High Performance Clustering of Social Images in a Map-Collective Programming Model”. Poster in proceedings of ACM Symposium On Cloud Computing, 2013.
14. E. Chan, M. Heimlich, A. Purkayastha, and R. Geijn. “Collective communication: theory, practice, and experience”. Concurrency and Computation: Practice and Experience (19), 2007.
15. MPI Forum. “MPI: A Message Passing Interface”. In Proceedings of Supercomputing, 1993.
16. MPICH2-YARN. <https://github.com/clarkyzl/mpich2-yarn>
17. Y. Low, et al. “GraphLab: A New Parallel Framework for Machine Learning”. Conference on Uncertainty in Artificial Intelligence, 2010.
18. Y. Low, et al. “Distributed GraphLab: A Framework for Machine Learning and Data Mining in the Cloud”. PVLDB, 2012.
19. J. Gonzalez, et al. “PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs”. OSDI, 2012.
20. R. Xin, et al. “GraphX: A Resilient Distributed Graph System on Spark”. GRADES, SIGMOD workshop, 2013.
21. Apache Hive. <http://hive.apache.org/>
22. M. Isard et al. “Dryad: Distributed Data-Parallel Programs from Sequential Building Blocks”. EuroSys, 2007.
23. Apache Pig. <http://pig.apache.org>
24. Apache Tez. <http://tez.incubator.apache.org>
25. C. Engle, et al. “Shark: Fast Data Analysis Using Coarse-grained Distributed Memory”. SIGMOD, 2012.
26. Apache MRQL. <http://mrql.incubator.apache.org>
27. Apache S4. <http://incubator.apache.org/s4/>
28. Apache Storm. <http://storm.incubator.apache.org/>
29. Apache Samza. <http://samza.incubator.apache.org/>
30. M. Zaharia, et al. “Discretized Streams: An Efficient and Fault-Tolerant Model for Stream Processing on Large Clusters”. HotCloud, 2012.
31. Stratosphere. <http://stratosphere.eu/>
32. REEF. <http://www.reef-project.org/>
33. M. Chowdhury et al. “Managing Data Transfers in Computer Clusters with Orchestra”. ACM SIGCOM, 2011.
34. T. Gunarathne et al. “Towards a Collective Layer in the Big Data Stack”. CCGrid, 2014
35. T. Fruchterman, and M. Reingold. “Graph Drawing by Force-Directed Placement”, Software – Practice & Experience 21 (11): 1129–1164, 1991.
36. Y. Ruan. “A Robust and Scalable Solution for Interpolative Multidimensional Scaling With Weighting”. E-Science, 2013.
37. Big Red II. <https://kb.iu.edu/data/bcqt.html>