**Harp: Collective Communication on Hadoop**

Bingjing Zhang, Yang Ruan

Indiana University

*Abstract— Big data tools have evolved rapidly in recent years. In the beginning, there was only MapReduce, but later tools in other different models have also emerged. 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 none of them can adapt to various kinds of problems and applications. In this paper, we abstract a collective communication layer which provides optimized communication operations on different data abstractions such as arrays, key-values and graphs, and define Map-Collective model which serve the diverse communication demands in different parallel applications*. *In addition, we design and implement the collective communication abstraction and Map-Collective model in Harp library and plug it into Hadoop. Then Hadoop can do in-memory communication between Map tasks without writing intermediate data to HDFS. With improved expressiveness and acceptable performance on collective communication, we can simultaneously support various applications from HPC to Cloud systems together.*

# Introduction

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

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

To process graph data, Google announced Pregel [6]. Later its open source version Giraph [7] and Hama [8] came out. These tools also use caching to accelerate iterative algorithms, only abstracting input data not as key-value pairs but as a graph, as well as defining communication in each iteration as messaging between vertices along edges.

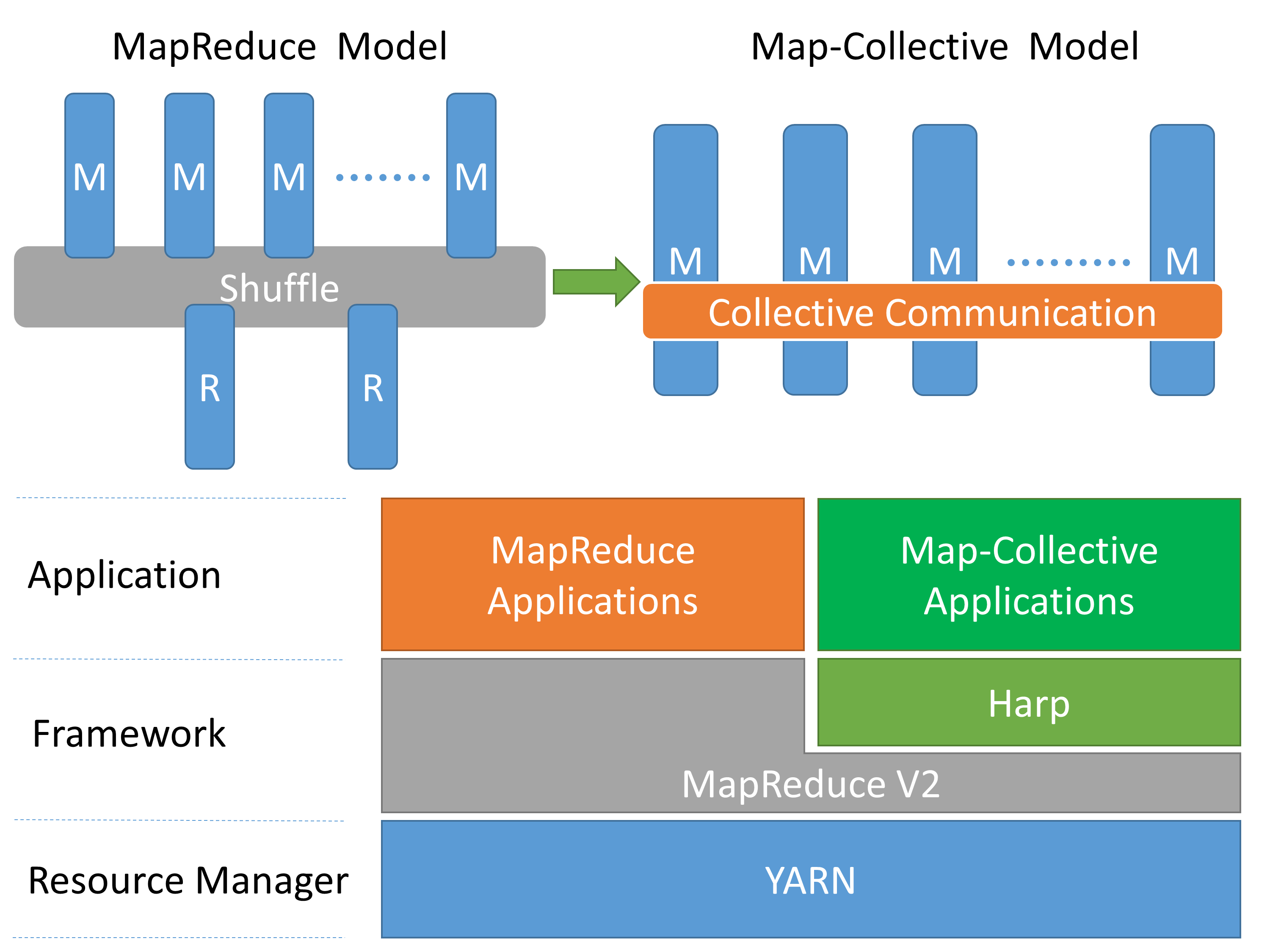


Figure 1. Parallelism and Architecture of Harp

However, all these tools are based on a kind of “top-down” design. The whole programming model, from data abstraction to computation and communication pattern, is fixed. Users have to put their applications into the programming model, which 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. This data is then 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 compute each part of them in parallel. This kind of “(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 between workers with all-to-all communication directly.

Regardless, “gather-broadcast” is not an efficient way to relocate the new centroids generated by reduce tasks, especially when the centroids data grows 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 neglected). 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 conflict in communication is not considered).

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

In the 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, 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 focus on every detail of a communication call. For example, users have to calculate the buffer size for data receiving, but this is hard to obtain in many applications as the amount of sending data may be very dynamic and unknown to the receivers.

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. It provides data abstractions and related communication abstractions with optimized implementation. By plugging Harp into Hadoop, we convert MapReduce model to Map-Collective model and enables efficient in-memory communication between map tasks to adapt various communication needs in different applications. The word “harp” symbolizes the effort to make parallel processes cooperate together through collective communication for efficient data processing just as strings in harps can make concordant sound (See Figure 1). With these two important contributions, collective communication abstraction and Map-Collective model, Harp is neither a replication of MPI nor some other research work which might try to transplant MPI into Hadoop system [16].

Table 1. Computation Model, Data Abstraction and Communication in Different Tools

|  |  |  |  |
| --- | --- | --- | --- |
| Tool | Computation Model | Data  Abstraction | Communication |
| MPI | Loosely  Synchronous | N/A | Arrays and objects sending/receiving between processes, or collective communication operations. |
| Hadoop | MapReduce | Key-Values | Shuffle (disk-based) between Map stage and Reduce stage. |
| Twister | MapReduce | Key-Values | Regroup data (in-memory) between Map stage and Reduce stage and a few collective communication operations such as “broadcast” and “aggregate”. |
| Spark | DAG/ MapReduce | RDD | Communication between workers in transformations on RDD and a few collective communication operations such as “broadcast” and “aggregate”. |
| Giraph | Graph/BSP | Graph | Graph-based communication following Pregel style. Communication happens at the end of each iteration. Data is abstracted as messages and sent along out-edges to target vertices. |
| Hama | Graph/BSP | Graph/ Messages | Graph-based communication following Pregel style. Besides it can do general message sending/receiving between workers at the end of each iteration. |
| GraphLab | Graph/BSP | Graph | Graph-based communication, but communication is hidden and viewed as caching and fetching of ghost vertices and edges. In PowerGraph model, the communication is hidden in GAS model as communication between master vertex and its replicas. |
| GraphX | Graph/BSP | Graph | Graph-based communication supports Pregel model and PowerGraph model. |
| Dryad | DAG | N/A | Communication happens between two connected vertex processes in execution DAG. |

In the rest 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-Harp. Section 5 gives examples of several applications implemented in Harp. Section 6 shows the performance of Harp through benchmarking on the applications.

# Related Work

Previously we mentioned many different big data tools. After years of development and evolution, this world is getting bigger and more complicated. Each tool has its own computation model with related data abstraction and communication abstraction in order to achieve optimization for different applications (See Table 1).

Before the MapReduce model, MPI was the main tool used to process big data. It is deployed on expensive hardware such as HPC or supercomputers. But MapReduce tries to use commodity machines to solve big data problems. This model defines data abstraction as key-value pairs and computation flow as “map, shuffle and then reduce”. “Shuffle” operation is a local disk based communication which can “regroup” and “sort” intermediate data. One advantage of this tool is that it doesn’t rely on memory to load and process all the data, instead using local disks. So MapReduce model can solve problems where the data size is too large to fit into the memory. Furthermore, it also provides fault tolerance which is important to big data processing. The open source implementation of this model is Hadoop [2], which is largely used nowadays in industry and academia.

Though MapReduce became popular for its simplicity and scalability, it is still slow when running iterative algorithms. Iterative algorithms require a chain of MapReduce jobs which result in repeat input data loading in each iteration. 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 the 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 on vertices. In each iteration, the results are sent as messages along out-edges to neighboring vertices and processed by them at the subsequent iteration. The whole parallelization is BSP (Bulk Synchronous Parallel) style. There are two open source projects following Pregel’s design. One is Giraph [7] and another is Hama [8]. Giraph exactly follows iterative BSP graph computation pattern while Hama tries to build a general BSP computation model.

At the same time, GraphLab [17] [18] does graph processing in a different way. It abstracts graph data as “a data graph” and uses consistency models to control vertex value update. GraphLab was later enhanced with PowerGraph [19] abstraction to reduce the communication overhead by using “vertex-cut” graph partitioning and GAS (Gather-Apply-Scatter) computation model. This was also learned by GraphX [20].

The third model is 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. DAG model is helpful to those applications which have complicated parallel workflows. Dryad [21] is an example of parallel engine using DAG model. Tools using this model are often used for query and stream data processing.

Because each tool is designed in different models and aimed at different categories of algorithms, it is difficult for users to pick up the correct tool for their applications. As a result, model composition becomes a trend in the current design of big data tools. For example, Spark’s RDD data abstraction and the transformation operations on RDDs are very similar to MapReduce model. But it organizes computation tasks as DAGs. Stratosphere [22] and REEF [23] also try to include several different models in one framework.

However, for all these tools, communication is still hidden 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 gave one example “allreduce” in the Introduction. Here we take “broadcast” as another example. “Broadcast” is done through simple algorithm (one-by-one sending) in Hadoop. But it is optimized by using BitTorrent technology in Spark [24] or using a pipeline-based chain algorithm in Twister [12] [13].

Though these research work [12] [13] [24] [25] 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 build a computation model which provides a rich set of communication operations and gives users flexibility in choosing suitable operations to their applications.

A common question is why we don’t use MPI directly since it already offers collective communication abstraction. There are many reasons. The collective communication in MPI is still limited in abstraction. It 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 directly 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 commonly deployed on HPC or supercomputers. Despite projects like [16], it is not as well integrated with cloud environments as Hadoop ecosystems.

# Collective Communication Abstraction

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

## Hierarchical Data Abstraction

The data abstraction has 3 categories horizontally and 3 levels vertically (Figure 2). Horizontally, data is abstracted as arrays, key-values or vertices, edges and messages in graphs. Vertically we build abstractions 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. For object type, to describe graph data, there is vertex object, edge object and message object; to describe key-value pairs, we use key object and value object.

Next, at the middle level, basic types are wrapped as array partitions, key-value partitions and graph partitions (edge partition, vertex partition and message partition). Notice that we follow the design of Giraph, edge partition and message partition are built from byte arrays but not from edge objects or message objects directly. When reading, bytes are converted to an edge object or a message object. When writing, the edge object or the message object is serialized and written back to byte arrays.

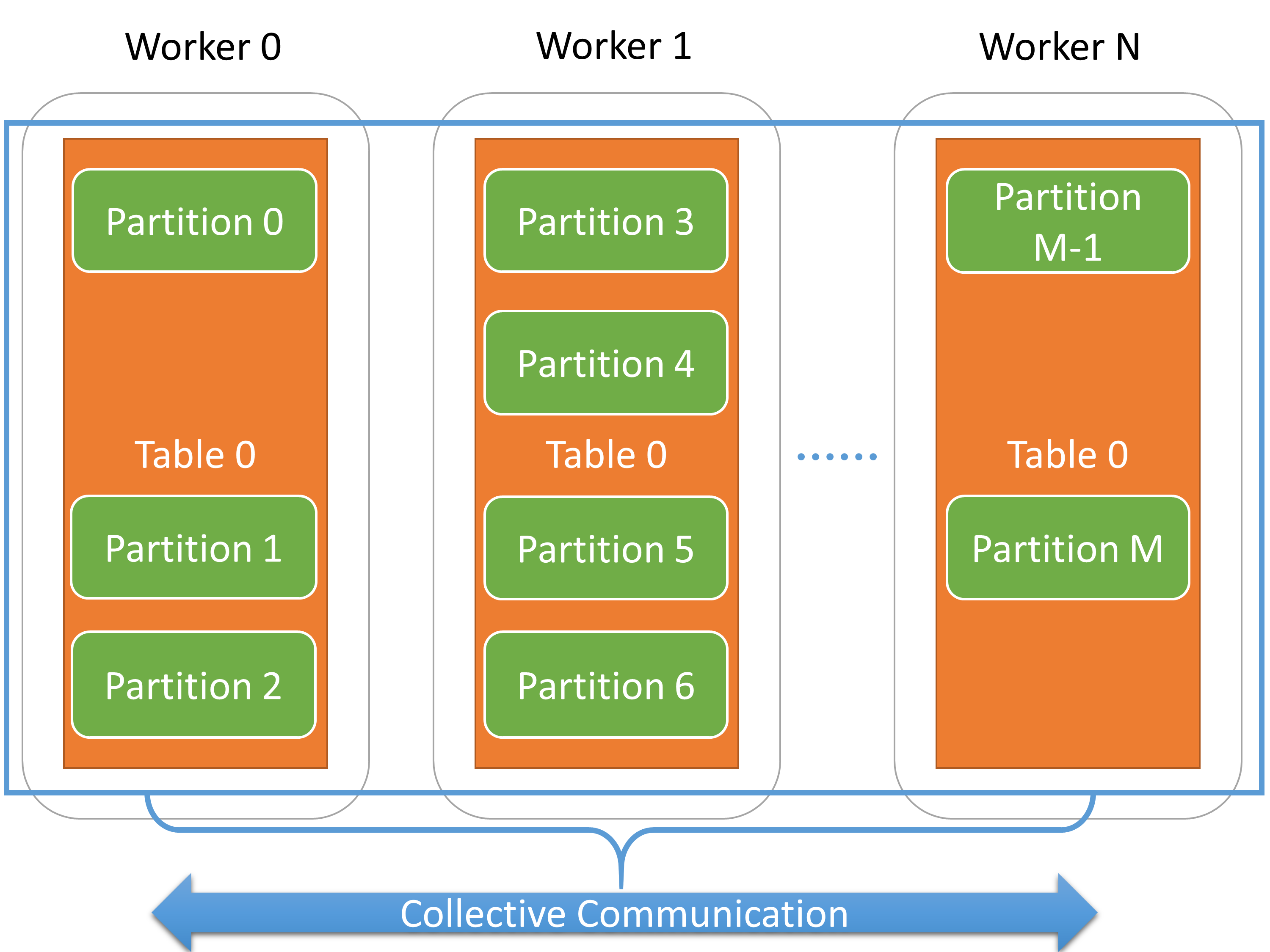


Figure 3. Abstraction of Tables and Partitions

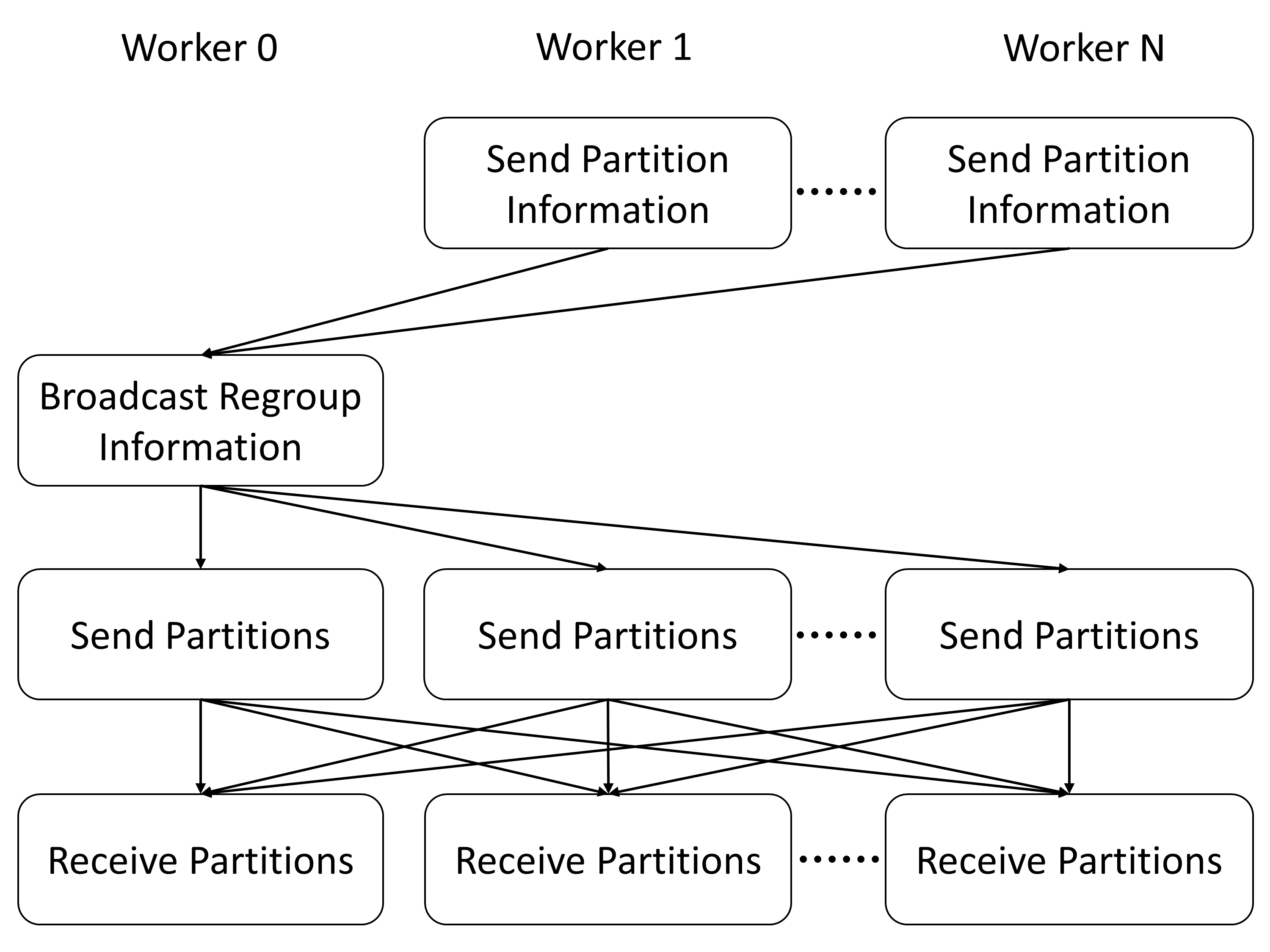


Figure 4. The Process of Regrouping Array Tables

At the top level, partitions are put into tables. A 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 by either combining or merging two partitions into one. Tables on different workers are associated with each other through table IDs. Tables which share the same table ID are considered as one dataset and “collective communication” is defined as redistribution or consolidation of partitions in this dataset. For example, in Figure 3, 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 the partitions between these tables. We will talk more in detail about the behavior of partition movement in collective communication operations.

## 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”, and “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 “regroup vertices or edges”, “move edges to vertices” and “send messages to vertices”.

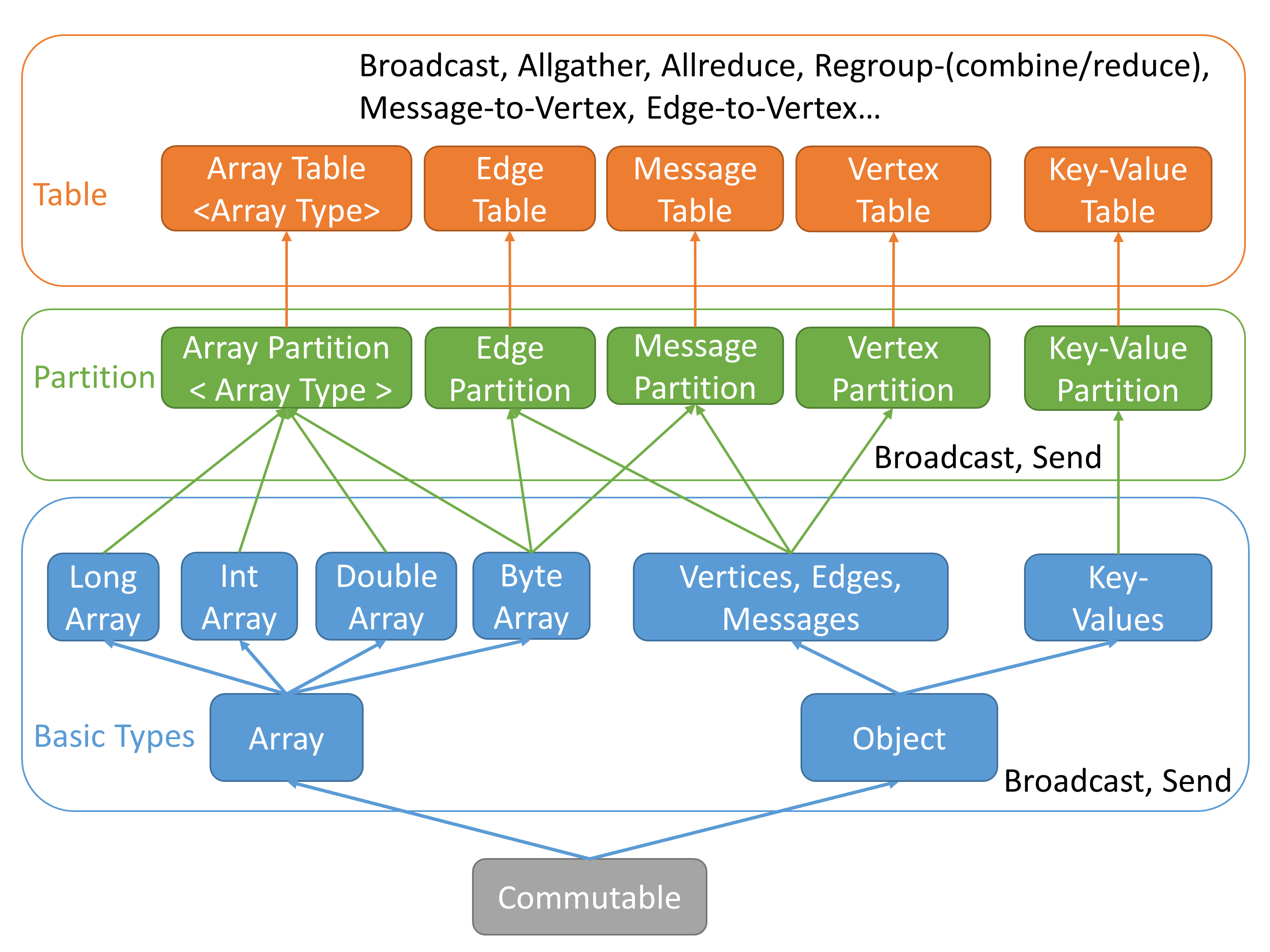


Figure 2. Hierarchical Data Abstraction and Collective Communication Operations

Some collective communication operations tie to certain data abstractions. For example, graph collective communication operations have to be done on graph data. But for other operations, the boundary is blurred. For example, “allgather” operation can be used on array tables, key-value tables, and vertex tables. But currently we only implement it on array tables and vertex tables. The following is a table which summarizes all the operations identified from applications and related data abstractions (See Table 2). We continue adding other collective communication operations not shown on this table in the future.

Here we take another look at Figure 3 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 the 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 dataset. Once all the partition IDs are received, master worker decides the destination worker IDs of each partition. Usually the decision is done through modulo operation. Once the master’s decision is made, the result is broadcasted to all workers. After that each worker starts to send out and receive partitions from other workers (See Figure 4).

Each collective communication can be implemented in many different algorithms. This has been discussed 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 like what we discussed in the Introduction, “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, though “regroup” cannot help much, “bidirectional-exchange” is more effective. Currently different algorithm are provided in different operation calls but we tend to provide automatic algorithm selection in future.

Table 2. 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 |  |  | √ |

In addition, we also optimize the “decision making” stages of several collective communication operations when the partition distribution is known in the application context. Normally just like we show in Figure 4, the master worker has to collect the partition distribution on each worker and broadcast the “regroup” decision to let them know which partition to send and which to receive. But when the partition distribution is known, the step of information “gather-and-broadcast” can be skipped. For example, we provide an implementation of “allgather” when the total number of partitions is known. In general, we enrich Harp collective communication library by providing different implementations for each operation so that users can choose the proper one based on the application requirement.

## Implementation

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 crucial in computation and collective communication of iterative algorithms. In these algorithms, the collective communication operations are called repeatedly and the intermediate data between iterations is similar in size, just with different content. Resource pool caches the data used in the last iteration to enable it to reuse them in the next. Therefore the application can avoid repeating allocating memory and lower the time used on garbage collection.

The process of sending is as follows: the worker first serializes the data to a byte array fetched from the resource pool and then send it through the socket. Receiving is managed by the receiver component. It starts a thread to listen to the socket requests. For each request, receiver spawns a handler thread to process it. We use “producer-consumer” model to process the data received. For efficiency, we don’t maintain the order of data in sending and receiving. Each data is identified by its related metadata information. Handler threads add the data received to the data queue. The main thread of the worker fetches data from the queue and examines if it belongs to this round of communication. If yes, the data is removed from the queue; otherwise it will be put back into the queue again (See Figure 5).

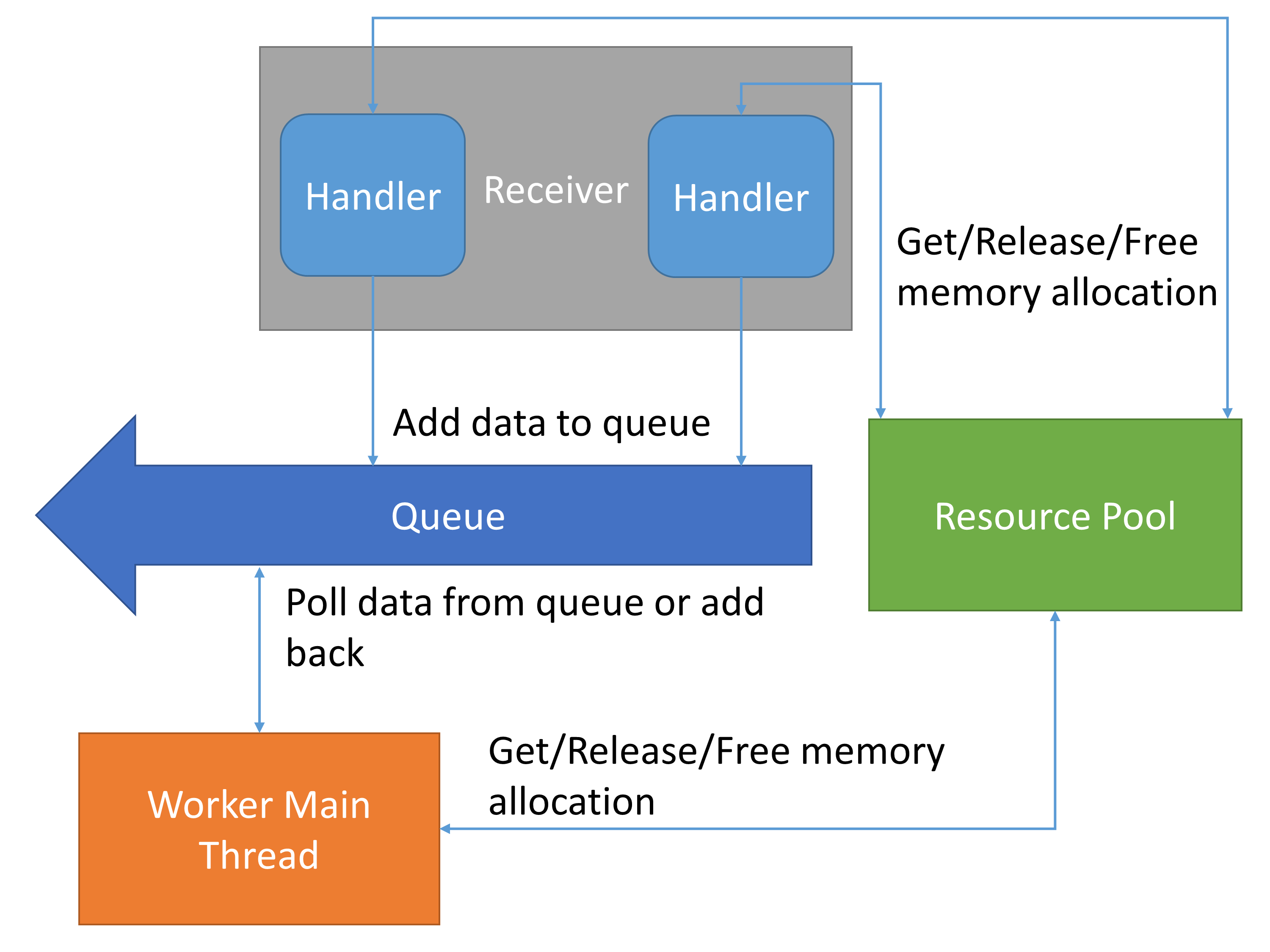


Figure 5. The Mechanism of Receiving Data

# Map-Collective Model

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. Therefore this work can be used to improve collective communication operations in any existing big data tool. But since communication is hidden in these tools, the applications still cannot be benefited from the expressiveness of collective communication abstraction. 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 and Harp Installation

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 the Harp jar package into the Hadoop library directory. For Hadoop 1, user need to configure the job scheduler to the scheduler designed for Map-collective jobs. But in Hadoop 2.0, since YARN resource management layer and MapReduce framework are separated, users are not required to change the scheduler. Instead, they 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.

Table 3. “mapCollecitve” interface

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

## MAP-COLLECTIVE INTERFACE

In Map-Collective model, user-defined mapper classes are extended from the class CollectiveMapper which is extended from the class Mapper in the original MapReduce framework. In CollectiveMapper, users need to override a method “mapCollective” with application code. This mechanism is just like the override of “map” method in Class Mapper. But “mapCollective” method doesn’t use a single pair of key and value as parameters but instead uses KeyValReader. KeyValReader provides flexibility to users; therefore 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 3).

CollectiveMapper initializes all the components required in collective communication. Users can invoke collective communication calls directly in the “mapCollective” method. We also expose the current worker ID and resource pool to users. Here is an example of how to do “allgather” in “mapCollective” method (See Table 4).

Table 4. “Allgather” code example

|  |
| --- |
| // Generate array partitions  List<ArrPartition<DoubleArray>>  arrParList = new ArrayList<  ArrPartition<DoubleArray>>();  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] = j;  }  arrParList.add(  new ArrPartition<DoubleArray>(  array, i)); } |
| // Define array table  ArrTable<DoubleArray, DoubleArrPlus>  arrTable =  new ArrTable<  DoubleArray, DoubleArrPlus>(  0, DoubleArray.class,  DoubleArrPlus.class);  // Add partitions to the table  for (ArrPartition<DoubleArray> arrPar :  arrParList) {  arrTable.addPartition(arrPar);  } |
| // Allgather  **allgatherTotalKnown**(  arrTable, numPartitions); |

Firstly we generate several array partitions with arrays fetched from the 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 invoke “allgather”. DoubleArrPlus is the combiner class used in these array tables to solve partition ID conflict in partition receiving. The “allgather” method used here is called “allgatherTotalKnown”. Because the total number of partitions in provided as a parameter in this version of “allgather”, workers don’t need to negotiate the number of partitions to receive from each worker but send out all the partitions they own to their neighbor directly with the bucket algorithm.

## 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 into different nodes and do collective communication iteratively. The whole parallelism follows the BSP pattern.

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 applications in Hadoop-2.2.0. In job configuration at client side, users need to set "mapreduce.framework.name" to "map-collective". Then the system chooses MapCollectiveRunner as job client instead of default YARNRunner for MapReduce jobs. MapCollectiveRunner launches MapCollectiveAppMaster to the cluster. MapCollectiveAppMaster is similar to MRAppMaster because both of them are responsible for requesting resources and launching tasks. When MapCollectiveAppMaster requests resources, it schedules the tasks to different nodes. This can maximize memory sharing and multi-threading on each node and save the intermediate data size in collective communication.

In launching stage, MapCollectiveAppMaster records the location of each task and generates 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 among all the workers. To ensure every worker has started, we use a “handshake”-like mechanism to synchronize them. In the first step, the master worker tries to ping its subordinates by sending a message. In the second step, slave workers who received the ping message will send a response message back to let the master know they are alive. In the third step, once the master gets all the responses, it broadcasts a small message to all workers to notify them of the initialization’ success.

When the initialization is done, each worker invokes “mapCollective” method to do computation and communication. We design the interface “doTasks” to enable users to launch multithread tasks. Given an input partition list and a Task object with user-defined “run” method and, the “doTasks” method can automatically do multi-threading parallelization and return the outputs.

## Fault Tolerance

We separate fault tolerance issue into two sections. One is fault detection and another is fault recovery. Currently our effort is to ensure every worker can report exceptions or faults correctly without getting hung up. 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, jobs with a large number of iterations can be 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.

# Applications

We give 3 application examples here to show how they 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 per iteration. But the third one is much more complicated. It has nested iterations and two different collective communication operations are used alternately. In data abstraction, the first and the third algorithm uses array abstraction but the second one utilizes graph abstraction. For key-value abstraction, we only implemented Word Count. We don’t introduce it here because it is very simple, with only one “regroup” operation and without iterations.

## 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 broadcasts it to all the workers. Later, in each iteration, a worker calculates its own local centroids and then uses “allreduce” operation at the end of the iteration to produce the global centroids of this iteration on each worker. After several iterations, the master worker 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 use “regroup-allgather” to do “allreduce”. Each local intermediate data is chunked to partitions. We firstly “regroup” them based on partition IDs. Next, on each worker we reduce the partitions with the same ID to obtain one partition of the new centroids. Finally, we do “allgather” on new generated data to let every worker have all the new centroids.

## Force-directed Graph Drawing Algoritm

We implement a Hadoop-Harp version of the Fruchterman-Reingold algorithm which produces aesthetically-pleasing, two-dimensional pictures of graphs by doing simplified simulations of physical systems [26].

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, then calculates attractive forces to pull them close, 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 is 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 is calculated, new vertex positions are generated. Because the algorithm requires calculation of the repulsive forces between every two vertices, we use “allgather” to redistribute the current positions of the vertices to all the workers. By combining multiple collective communication operations from different categories, 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 descent-type of algorithm which is widely used for large-scale Multi-dimensional Scaling (MDS) problems [27]. 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.

Weighted Deterministic Annealing SMACOF (WDA-MSACOF) is an algorithm which optimize the original SMACOF. It uses deterministic annealing technique to avoid local optima during stress majorization, and it uses conjugated gradient for a weighting function in order to keep the time complexity of the algorithm in *O(N2).* Originally the algorithm is commonly used in a data clustering and visualization pipeline called DACIDR [28]. In the past, the workflow uses both Hadoop and Twister in order to achieve maximum performance [29]. With the help of Harp, this pipeline could be directly implemented on it instead of using the hybrid MapReduce model.

WDA-SMACOF has nested iterations. In every outer iteration, we firstly do an update on an order *N* matrix, then do a matrix multiplication; we calculate the coordination values of points on the target dimension space through conjugate gradient process; the stress value of this iteration is calculated as the final step. Inner iterations are the conjugate gradient process which is to solve the equation similar to *Ax=b* in iterations of matrix multiplications.

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

In Hadoop-Harp, we improve the parallel implementation using “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

We test the 3 applications described above to examine the performance of Hadoop-Harp. We use Big Red II supercomputer [30] as the test environment.

## Test Environment

We use the nodes in “cpu” queue on Big Red II. Each node has 32 processors and 64 GB memory. The nodes are connected with Cray Gemini interconnect.

We deploy Hadoop-2.2.0 on Big Red II. The JDK version we use is 1.7.0\_45. Hadoop is not naturally adopted by supercomputers like Big Red II, so we need to do some adjustment. Firstly, we have to submit a job in Cluster Compatibility Mode (CCM) but not Extreme Scalability Mode (ESM). In addition, because there is no local disk on each node and /tmp directory is mapped to part of the memory (about 32GB), we cannot hold large data on local disks in HDFS. For small input data, we still use HDFS, but for large data, we choose to use Data Capacitor II (DC2), the file system connected to compute nodes. We create partition files in Hadoop job client, each of which contains several file paths on DC2. The number of partition files is matched with the number of map tasks. Each map task reads all file paths in a partition file as key-value pairs and then reads the real file contents from DC2. In addition, the implementation of communication in Harp is based on Java socket, we didn’t do any optimization aimed at Cray Gemini interconnect.

In all the tests, we deploy one worker on each node, and utilize 32 processors to do multi-threading inside. Generally we test on 8 nodes, 16 nodes, 32 nodes, 64 nodes and 128 nodes (which is the maximum number of nodes allowed for job submission on Big Red II). This means 256 processors, 512 processors, 1024 processors, 2048 processors and 4096 processors. But to reflect the scalability and the communication overhead, we calculate efficiency based on the number of nodes but not the number of processors.

In JVM execution command of each worker, we set both “Xmx” and “Xms” to 54000M, “NewRatio” to 1 and “SurvivorRatio” to 98. Because most memory allocation is cached and reused, it is not necessary to keep large survivor spaces. We increase SurvivorRatio and lower down survivor spaces to a minimum so we can leave most of the young generation to Eden space.

## Results on K-Means Clustering

We test K-Means clustering with two different generated random data sets. One is clustering 500 million 3D points into 10 thousand clusters and another is clustering 5 million 3D points into 1 million clusters. In the former case, the input data is about 12 GB and the ratio of points to clusters is 50000:1. In the latter case, the input data size is only about 120 MB but the ratio is 5:1. The ratio is commonly high in clustering but the low ratio is used in a different scenario where the algorithm tries to do fine-grained clustering as classification [31] [32]. Because each point is required to calculate distance with all the cluster centers, total workloads of the two tests are similar.

We use 8 nodes as the base case and then scale to 16, 32, 64 and 128 nodes. The execution time and speedup are shown in Figure 6. Due to the cache effect, we see “5 million points and 1 million centroids” is slower than “500 million points and 10 thousands centroids” when the number of nodes is small. But as the number of nodes increases, they draw closer to one another. For speedup, we assume we have the linear speedup on the smallest number of nodes we test. So we consider the speedup on 8 nodes is 8. The experiments show the speedups in both test cases is close to linear.

Figure 8. Execution Time of WDA-SMACOF

Figure 9. Parallel Efficiency of WDA-SMACOF

Figure 10. Speedup of WDA-SMACOF

Figure 6. Execution Time and Speedup of K-Means Clustering

Figure 7. Execution Time and Speedup of Force-Directed Graph Drawing Algorithm

## Results on Force-directed Graph Drawing Algorithm

We test this algorithm with a graph of 477111 vertices and 665599 undirected edges. The graph represents a retweet network about the presidential election in 2012 from Twitter [33].

The size of input data is pretty small but the algorithm is computation intensive. We load vertex ID as int and initial random coordination values as float. The total size is about 16M. We test the algorithm on 1 node as the base case and then scale to 8, 16, 32, 64 and 128 nodes. We present the execution time of 20 iterations and the speedup in Figure 7. From 1 node to 16 nodes, we observe almost linear speedup. It drops smoothly after 32 nodes. On 128 nodes, because the computation time per iteration gets very short to around 3 seconds, the speedup drops a lot.

## Results on WDA-SMACOF

We test WDA-SMACOF with different problem sizes including 100K points, 200K points, 300K points and 400k points. Each point represents a gene sequence in a dataset of representative 454 pyrosequences from spores of known AM fungal species [34]. Because the input data is the distance matrix of points and related weight matrix and V matrix, the total size of input data is in quadratic growth. We cache distance matrix in short arrays, weight matrix in double arrays and V matrix in int arrays. Then the total size of input data is about 140 GB for 100K problem, about 560 GB for 200K problem, 1.3 TB for 300K problem and 2.2 TB for 400K problem.

The input data is stored in DC2 and each matrix is split into 4096 files. They are loaded from there to workers. Due to memory limitation, the minimum number of nodes required to run the 100K problem is 8. Then we scale 100K problem on 8, 16, 32, 64 and 128 nodes. But for the 200K problem, the minimum number of nodes required is 32. So we scale the 200K problem on 32, 64 and 128 nodes. With the 300K problem, the minimum node requirement is 64. Then we scale 300K problem from 64 to 128 nodes. For 400K problem, we only run it on 128 nodes because this is the minimum node requirement for that amount.

Here we give the execution time, parallel efficiency and speedup. Because we cannot run each input on a single machine, we choose the minimum number of nodes to run the job as the base to calculate parallel efficiency and speedup. In most cases, the efficiency values are very good. The only point that has low efficiency is 100K problem on 128 nodes. In this test, the communication overhead doesn’t change much. But due to low computation on each node, communication overhead takes about 40% of total execution time, therefore the overall efficiency drops.

# CONCLUSION

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

We implement collective communication abstraction and Map-Collective model in a 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 with the collective communication abstraction and the Map-Collective model, these applications can be simply expressed with the combination of collective communication operations. Through experiments on the Big Red II supercomputer, we show that we can scale these applications to 128 nodes with 4096 processors. The speedups in most of tests are close to linear on 64 nodes and in some of tests the speedup can even keep linear on 128 nodes.

# 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. M. Isard et al. “Dryad: Distributed Data-Parallel Programs from Sequential Building Blocks”. EuroSys, 2007.
22. Stratosphere. <http://stratosphere.eu/>
23. REEF. <http://www.reef-project.org/>
24. M. Chowdhury et al. “Managing Data Transfers in Computer Clusters with Orchestra”. ACM SIGCOM, 2011.
25. T. Gunarathne et al. “Towards a Collective Layer in the Big Data Stack”. CCGrid, 2014
26. T. Fruchterman, and M. Reingold. “Graph Drawing by Force-Directed Placement”, Software – Practice & Experience 21 (11): 1129–1164, 1991.
27. Y. Ruan. “A Robust and Scalable Solution for Interpolative Multidimensional Scaling With Weighting”. E-Science, 2013.
28. Y. Ruan, et al. “DACIDR: Deterministic Annealed Clustering with Interpolative Dimension Reduction using a Large Collection of 16S rRNA Sequences”. Proceedings of ACM-BCB, 2012.
29. Y. Ruan, et al. “HyMR: a Hybrid MapReduce Workflow System”. Proceedings of ECMLS’12 of ACM HPDC, 2012
30. Big Red II. <https://kb.iu.edu/data/bcqt.html>
31. G. Fox. “Robust Scalable Visualized Clustering in Vector and non Vector Semimetric Spaces”. Parallel Processing Letters 23, 2013.
32. G. Fox, D. Mani. “Parallel Deterministic Annealing Clustering and Its Application to LC-MS Data Analysis”. Big Data, 2013.
33. X. Gao and J. Qiu. “Social Media Data Analysis with IndexedHBase and Iterative MapReduce,” Proc. Workshop on Many-Task Computing on Clouds, Grids, and Supercomputers (MTAGS) at Super Computing 2013.
34. Y. Ruan, G. House, S. Ekanayake, U. Schütte, J. Bever, H. Tang, G. Fox. “Integration of Clustering and Multidimensional Scaling to Determine Phylogenetic Trees as Spherical Phylograms Visualized in 3 Dimensions”. Proceedings of C4Bio of IEEE/ACM CCGrid, 2014.