Harp: Collective Communication on Hadoop

Bingjing Zhang, Yang Ruan, Judy Qiu
Computer Science Department
Indiana University
Bloomington, IN, USA
zhangbj, yangruan, xqiu@indiana.edu

***Abstract*—Big data processing tools have evolved rapidly in recent years. MapReduce has proven very successful but is not optimized for many important analytics, especially those involving iteration. In this regard, Iterative MapReduce frameworks improve performance of MapReduce job chains through caching. Further, Pregel, Giraph and GraphLab abstract data as a graph and process it in iterations. But all these tools are designed with a fixed data abstraction and have limited collective communication support to synchronize application data and algorithm control states among parallel processes. In this paper, we introduce a collective communication abstraction layer which provides efficient collective communication operations on several common data abstractions such as arrays, key-values and graphs, and define a MapCollective programming model which serves the diverse collective communication demands in different parallel algorithms. We implement a library called Harp to provide the features above and plug it into Hadoop so that applications abstracted in MapCollective model can be easily developed on top of MapReduce framework and conveniently integrated with other tools in Apache Big Data Stack. With improved expressiveness in the abstraction and excellent performance on the implementation, we can simultaneously support various applications from HPC to Cloud systems together with high performance.**

***Keywords*-Collective Communication; Big Data Processing; Hadoop**

I. Introduction

Beginning with the publication of Google's MapReduce paper [1], the last decade saw a huge shift in the evolution of big data processing tools. Since then Hadoop [2], the open source version of Google MapReduce, has become the mainstream of big data processing, with many other tools emerging to handle big data problems. Extending the original MapReduce model to include iterative MapReduce, tools such as Twister [3] and HaLoop [4] can cache loop invariant data in iterative algorithms locally to avoid repeat input data loading in a MapReduce job chain. Spark [5] also uses caching to accelerate iterative algorithms by abstracting computations as transformations on RDDs instead of restricting computations to a chain of MapReduce jobs. To process graph data, Google unveiled Pregel [6] and soon Giraph [7] emerged as its open source version.

Regardless of their differencess, all such tools are based on a kind of "top-down" design. Each has a fixed programming model which includes a data abstraction, a computation model and a communication pattern. The catch is that individual tools with a fixed pattern cannot adapt to a variety of applications, which could cause performance inefficiency.

For example, in k-means clustering with Lloyd's algorithm [8], every parallel task in the successive iteration needs all the centroids generated in the previous iteration. Mahout [9] on Hadoop chooses to reduce the outputs from all the map tasks in one reduce task, store the new centroids data on HDFS, and read the data back to memory for the next iteration. The whole process is commonly applied in many big data tools and can be summarized as "reduce-gather-broadcast". But "gather-broadcast" is not an efficient way to redistribute new centroids generated in the previous iteration, especially when the size of centroids data grows large. The time complexity of "gather" is at least $kd\beta$ 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 startup time α is neglected). Also the time complexity of "broadcast" is at least $kd\beta$ [10][11]. Therefore the time complexity of "gather-broadcast" is about $2kd\beta$. This can be reduced to $kd\beta$ if "allgather" operation is used instead [12], however none of these tools provides this pattern of data movement.

Many parallel iterative algorithms use such methods of data movement to synchronize the dependent application data and the algorithm execution states between all the parallel processes. These operations can be executed only once or multiple times per iteration, therefore their performance is crucial to the efficiency of the whole application. We call this type of data movement "Collective Communication". Iterative algorithms which were previously expressed as a chain of MapReduce jobs can now be re-abstracted as iterations of high performance collective communication operations. Such algorithms include k-means clustering, logistic regression, neural network, principal component analysis, expectation maximization and support vector machine, all of which follow the statistical query model [13].

Rather than fixing communication patterns, we decided to separate this layer out and build a collective communication abstraction layer. We studied a broad range of communication patterns including "allgather", "allreduce", "broadcast" in MPI collective communication operations, "shuffle" in MapReduce, "group-by" in database applications, and "send

messages along edges to neighbor vertices" in Pregel. Our contributions in this paper are as follows: (a) We provide a common set of data abstractions and related collective communication operation abstractions. On top of this abstraction layer we define the MapCollective programming model, which allows users to invoke collective communication operations to synchronize a set of parallel processes. (b) We implement these ideas in the Harp open source library [14] as a Hadoop plugin. The word "harp" symbolizes how parallel processes coordinate through collective communication operations for efficient data processing, just as strings in harps can make concordant sound. By plugging Harp into Hadoop, we can express the MapCollective model in a MapReduce framework and enable efficient in-memory collective communication between map tasks across a variety of important data analysis applications.

From here on, Section 2 discusses related work. Section 3 shows some application examples as expressed with collective communication operations. Section 4 describes the collective communication abstraction layer. Section 5 explains Map-Collective model works. Section 6 presents the Harp library implementation, and Section 7 shows Harp's performance through benchmarking on the applications.

II. Related Work

MapReduce became popular thanks to its simplicity and scalability, yet is still slow when running iterative algorithms. Frameworks like Twister, HaLoop and Spark solved this issue by caching intermediate data and developed the iterative MapReduce model. Another iterative computation model is the graph model, which abstracts data as vertices and edges and executes in BSP (Bulk Synchronous Parallel) style. Pregel and its open source version Giraph follow this design. By contrast, GraphLab [15] abstracts data as a "data graph" and uses consistency models to control vertex value updates. GraphLab was later enhanced with PowerGraph [16] abstraction to reduce the communication overhead. This was also used by GraphX [17]. For all these tools, collective communication is still hidden and coupled with the computation flow. Although some research works [10] [11] [18] [19] try to add or improve collective communication operations, they are still limited in operation types and constrained by the computation flow. As a result, it is necessary to build a separate communication abstraction layer. With this we can fashion a programming model that provides a rich set of communication operations and grants users flexibility in choosing operations suitable to their applications.

III. Application Scenarios

In this section we use k-means clustering, force-directed graph drawing algorithm, and weighted deterministic annealing SMACOF (WDA-SMACOF), to express the applications using collective communication operations.

A. K-means Clustering

At the start of k-means clustering with Lloyd's algorithm, each task loads and caches a part of the data points while a single task needs to prepare initial centroids and use "broadcast" operation to send the data to all other tasks. Later in every iteration, the tasks do their own calculations and then use "allreduce" operation to produce the global centroids of this iteration.

B. Force-directed Graph Drawing Algorithm

Fruchterman-Reingold algorithm produces aesthetically pleasing, two-dimensional pictures of graphs by crafting simplified simulations of physical systems [20]. Vertices of the graph act as atomic particles. Initially vertices are randomly placed in a 2D space. The displacement of each vertex is generated based on the calculation of attractive and repulsive forces. Every iteration, the algorithm calculates the effect of repulsive forces to push them away from each other, then determines attractive forces to pull them close, limiting the total displacement by temperature. Both attractive and repulsive forces are defined as functions of distance between vertices following Hooke's Law. The input data of this algorithm is abstracted as graph data. Since the algorithm requires calculation of the repulsive forces between every two vertices in the graph, the communication is more than just sending messages between neighbor vertices. Instead we use "allgather" to redistribute the current positions of the vertices to all the tasks between iterations.

C. WDA-SMACOF

SMACOF (Scaling by MAjorizing a COmplicated Function) is a gradient descent type of algorithm used for large-scale multi-dimensional scaling problems. Through iterative stress majorization, the algorithm minimizes the difference between distances from points in the original space and their distances in the new space. WDA-SMACOF improves on the original SMACOF [21]. It uses deterministic annealing techniques to avoid local optima during stress majorization, and employs conjugate gradient for the equation solving with a non-trivial matrix to keep the time complexity of the algorithm in $O(N^2)$. WDA-SMACOF has nested iterations. In every outer iteration, the algorithm firstly does an update on an order N matrix, then performs a matrix multiplication; the coordination values of points on the target dimension space is calculated through conjugate gradient process in inner iterations; the stress value of this iteration is determined as the final step. We express WDA-SMACOF with "allgather" and "allreduce", two operations. In outer iterations, "allreduce" sums the results from the stress value calculation. For inner iterations the conjugate gradient process uses "allgather" to collect the results from matrix multiplication and "allreduce" for those from inner product calculations.

IV. Collective Communication Abstractions

A. Hierarchical Data Abstractions

Various collective communication patterns have been observed in existing big data processing tools and the application examples. To support them, we first abstract data types in a hierarchy. In Fig. 1, we abstract data horizontally as arrays, key-values, or vertices, edges and messages in graphs. Vertically we construct abstractions from basic types to partitions and tables. Firstly, any data which can be sent or received is an implementation of interface `Transferrable`. At the lowest level, there are two basic types under this interface: arrays and objects. Based on the component type of an array, we now have byte array, int array, long array and double array. To describe graph data for object type there is vertex object, edge object and message object; for key-value pairs we use key object and value object. Next at the middle level, basic types are wrapped as array partition, key-value partition and graph data partition (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 deserialized to an edge object or a message object. When writing, either the edge object or the message object is serialized back to byte arrays. At the top level are tables containing several partitions, each with 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 them into one. Tables on different processes are associated with each other through table IDs. Tables sharing the same table ID are considered as one dataset and a collective communication operation is defined as redistribution or consolidation of partitions in this dataset. For example, in Fig. 2, a set of tables associated with ID 0 is defined on processes 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.

B. 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 adapted from MPI [22] collective communication operations: e.g. "broadcast", "allgather", and "allreduce".

2) Collective communication derived from MapReduce "shuffle-reduce" operation: e.g. "regroup" operation with "combine or reduce" support.

3) Collective communication abstracted from graph communication: e.g. "regroup vertices or edges", "send messages to vertices" and "send edges to vertices".

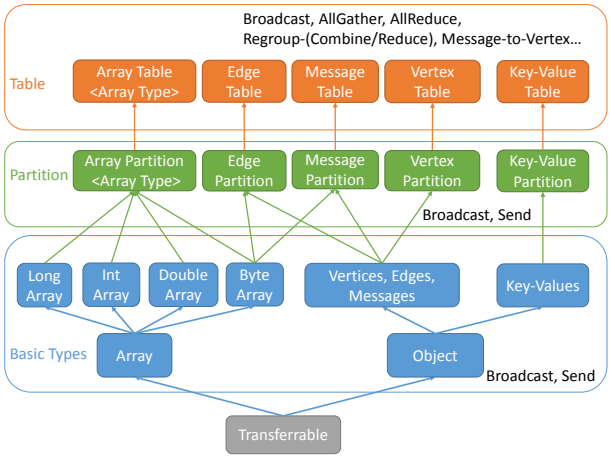


Figure 1. Hierarchical data abstractions

We list several defined operations in Table I. Some collective communication operations tie to certain data abstractions. For example, "send messages to vertices" has to be done on graph data. But for other operations, the boundary is blurred. From application examples, "allgather" operation is used both on array and vertex tables. Additionally, each collective communication can be implemented in a rich set of algorithms. We choose candidate algorithms for optimization based on two criteria: the frequency of the collective communication and the total data size in the collective communication. For the operation which most frequently occurs in the application, we choose the algorithm with high performance to reduce the cost on application data synchronization. With different data sizes, some algorithms are good for small data while others favor large data. For example, we have two versions of "allreduce". One is "bidirectional-exchange" algorithm [12] and another is "regroup-allgather" algorithm. When the data size is large

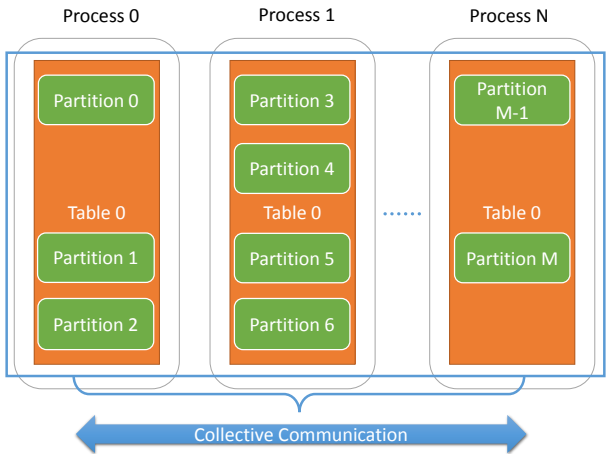


Figure 2. Tables and partitions in collective communication operations

Table I
COLLECTIVE COMMUNICATION OPERATIONS

Operation Name	Data Abstraction	Algorithm	Time Complexity
broadcast	arrays, key-value pairs & vertices	chain	$n\beta$
allgather	arrays, key-value pairs & vertices	bucket	$pn\beta$
allreduce	arrays, key-value pairs	bi-directional exchange	$(\log_2 p)n\beta$
		regroup-allgather	$2n\beta$
regroup	arrays, key-value pairs & vertices	point-to-point direct sending	$n\beta$
send messages to vertices	messages, vertices	point-to-point direct sending	$n\beta$
send edges to vertices	edges, vertices	point-to-point direct sending	$n\beta$

Notice that in Column "Time Complexity", p is the number of processes, n is the number of input data items per process, β is the per data item transmission time, communication startup time α is neglected and the time complexity of the "point-to-point direct sending" algorithm is estimated regardless of potential network conflicts.

and each table has many partitions, "regroup-allgather" is more suitable because it has less data sending and more balanced workload per process. But if the table on each process only has one or a few partitions, "bidirectional-exchange" is more effective.

V. MAPCOLLECTIVE PROGRAMMING MODEL

Since communication is hidden in many existing big data processing tools, even with a collective communication abstraction layer, the applications still cannot benefit from the expressiveness of this abstraction. As a solution we define a MapCollective programming model to enable using collective communication operations.

A. BSP Style Parallelism

MapCollective model follows the BSP style. We consider two levels of parallelism. At the first level, each parallel component is a process where the collective communication operations happen. The second is the thread level for parallel processing inside of each process. This is not mandatory in the model but it can maximize memory sharing and multi-threading in each process and save the data size in collective communication. To enable in-memory collective communication, we need to make every process alive simultaneously. As a result, instead of dynamic scheduling, we use static scheduling. When processes are scheduled and launched, their locations are synchronized between all the processes for future collective communications.

B. Fault Tolerance

When it comes to fault tolerance, failure detection and recovery are crucial system features. Currently we have focused our efforts on failure detection to ensure every process can report exceptions or faults correctly without getting hung up. Failure recovery poses a challenge because the execution flow in the MapCollective model is very flexible. Currently we do job level failure recovery. Based on the execution time length of scale, an algorithm with a large number of iterations can be separated into a small number of jobs, each of which contains several iterations. This naturally forms checkpointing between iterations. Since MapCollective 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 execution states between new launched tasks and other old live tasks.

VI. HARP IMPLEMENTATION

We implemented the collective communication abstraction layer and MapCollective model in the Harp library. By plugging it into Hadoop, users can write a MapCollective job with the support of MapReduce frameworks. Collective communication is enabled between map tasks.

A. Layered Architecture

The current Harp implementation targets Hadoop 2. Fig. 3 shows how different layers interface with each other in the architecture. At the bottom level is the MapReduce framework. We extend the original MapReduce framework to expose the network location of map tasks. In the upper level, Harp builds collective communication abstractions which provide collective communication operators, hierarchical data types of tables and partitions, and the memory allocation management pool for data caching and reuse. All these components interface with the MapCollective programming model. After wrapping, the MapCollective programming model provides three components to the application level: a MapCollective programming interface, a set of collective communication APIs, and data abstractions which can be used in the programming interface.

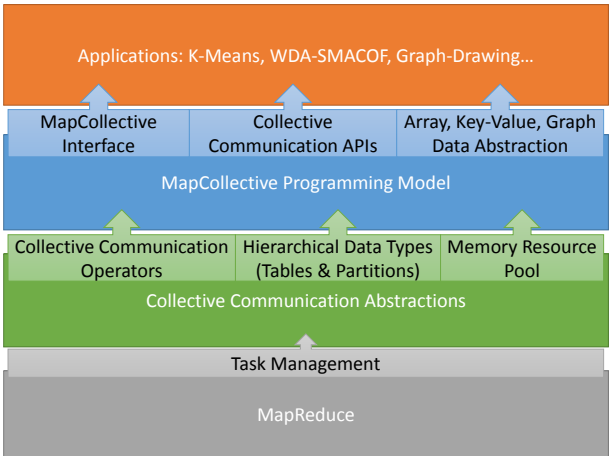


Figure 3. Architecture layers in Harp implementation

B. MapCollective Programming Interface

To program in MapCollective model, users need to override a method called `mapCollective` in class `CollectiveMapper` which is extended from class `Mapper` in the original MapReduce framework. While similar, `mapCollective` method differs from `map` method in class `Mapper` in that it employs `KeyValReader` to provide flexibility to users; therefore users can either read all key-values into the memory and cache them, or read them part by part to fit the memory constraint. `CollectiveMapper` not only provides collective communication operation APIs but also an API called `doTasks` to enable users to launch multithread tasks. Given an input partition list and a `Task` object with user-defined `run` method, the `doTasks` method can automatically perform thread level parallelization and return the outputs. See the code example below:

```
protected void mapCollective(
    KeyValReader reader,Context
    context) throws IOException,
    InterruptedException {
    // Put user code here...
    // doTasks(...)
    // allreduce(...)
}
```

VII. Experiments

A. Test Environment

We evaluate the performance of Hadoop-Harp on the Big Red II supercomputer [23]. The tests use the nodes in "cpu" queue where the maximum number of nodes allowed for job submission is 128. Each node has 32 processors and 64GB memory. The nodes are running in Cluster Compatibility Mode and connected with Cray Gemini interconnect. But the implementation of communication in Harp is based on Java socket without optimizations aimed at Cray Gemini interconnect. Hadoop-2.2.0 and JDK 1.7.0_45 are installed. Because there is only a small 32GB memory mapped local /tmp directory on each node, we choose Data Capacitor II (DC2) to store the data. We group file paths on DC2 into partition files on HDFS and let each map task read file paths as key-value pairs.

In all the tests, we deploy one map task on each node and utilize all 32 CPUs to do multi-threading inside. To reflect the scalability and the communication overhead, we calculate the speedup based on the number of nodes but not the number of CPUs. In JVM options of each map task, we set both Xmx and Xms to 54000M, NewRatio to 1 and SurvivorRatio to 98. Because most memory allocation is cached and reused through the memory resource pool, we can increase the size of the young generation and leave most of its space to Eden space.

B. Results on K-means Clustering

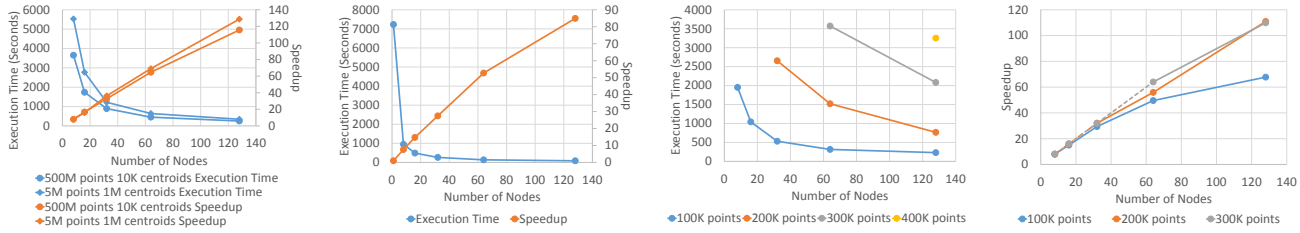
We run k-means clustering with two different random generated data sets. One is clustering 500 million 3D points into ten thousand clusters, while another is clustering 5 million 3D points into 1 million clusters. In the former, the input data is about 12GB and the ratio of points to clusters is 50000:1. In the latter case, the input data size is only about 120MB but the ratio is 5:1. Such a ratio is commonly high in clustering; the low ratio is used in a scenario where the algorithm tries to do fine-grained clustering as classification [24]. The baseline test uses 8 nodes, then scales up to 128 nodes. The execution time and speedup are shown in Fig. 4a. Since each point is required to calculate distance with all the cluster centers, total workload of the two tests is similar. But due to the cache effect, we see "5 million points and 1 million centroids" is slower than "500 million points and 10 thousand centroids" when the number of nodes is small. As the number of nodes increases, however, they draw closer to one another. We assume we have the linear speedup on the smallest number of nodes that we test. So we consider the speedup on 8 nodes is 8. The experiments show the speedup comparison in both test cases is close to linear.

C. Results on Force-directed Graph Drawing Algorithm

This algorithm runs with a graph of 477,111 vertices and 665,599 undirected edges. The graph represents a retweet network about the U.S. presidential election in 2012 from Twitter [25]. Although the size of input data is fairly small, the algorithm is computation intensive. We test the algorithm on 1 node as the base case and then scale up to 128 nodes. Execution time of 20 iterations and speedup are shown in Fig. 4b. From 1 node to 16 nodes, we observe almost linear speedup. The speedup drops smoothly after 32 nodes and then plummets sharply on 128 nodes because the computation time per iteration slows to around 3 seconds.

D. Results on WDA-SMACOF

The WDA-SMACOF algorithm runs with different problem sizes including 100K points, 200K, 300K and 400K. Each point represents a gene sequence in a dataset of representative 454 pyrosequences from spores of known AM fungal species [26]. 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. It is about 140GB for the 100K problem, about 560GB for 200K, 1.3TB for 300K and 2.2TB for 400K. Due to memory limitations, the minimum number of nodes required to run the application is 8 for the 100K problem, 32 for the 200K, 64 for 300K and 128 for 400K. The execution time and speedup are seen in Fig. 4c and Fig. 4d. Since 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



(a) (b) (c) (d)

Figure 4. The performance results of the applications (a) execution time and speedup of k-means clustering (b) execution time and speedup of force-directed graph drawing algorithm (c) execution time of WDA-SMACOF (d) speedup of WDA-SMACOF

is 100K problems on 128 nodes. This is a standard effect in parallel computing where the small problem size reduces compute time compared to communication, which in this case has an overhead of about 40% of total execution time.

VIII. Conclusion

We propose to abstract a collective communication layer in the existing big data processing tools to support communication optimizations required by the applications. We build a MapCollective programming model on top of collective communication abstractions to improve the expressiveness and performance of big data processing. Harp is an implementation designed in a pluggable way to bridge the differences between Hadoop ecosystem and HPC system and bring high performance to the Apache Big Data Stack through a clear communication abstraction, which did not exist before in the Hadoop ecosystem. Note that these ideas will allow simple modifications of Mahout library that drastically improve its low parallel performance; this demonstrates the value of building new abstractions into Hadoop rather than developing a totally new infrastructure as we did in our prototype Twister system. With three applications, the experiments show that with Harp we can scale these applications to 128 nodes with 4096 CPUs on the Big Red II supercomputer, where the speedup in most tests is close to linear. Future work will include the high performance communication libraries developed for simulation (exascale). We will extend the work on fault tolerance to evaluate the current best practices in MPI, Spark and Hadoop. We are working with several application groups and will extend the data abstractions to include those needed in pixel and spatial problems.

Acknowledgment

We appreciate the system support offered by Big Red II. We gratefully acknowledge support from National Science Foundation CAREER grant OCI-1149432.

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 et al. “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 et al. “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] S. Lloyd. “Least Squares Quantization in PCM”. IEEE Transactions on Information Theory 28 (2), 1982.
[9] Apache Mahout. https://mahout.apache.org/
[10] J. Qiu, B. Zhang. “Mammoth Data in the Cloud: Clustering Social Images”. In Clouds, Grids and Big Data, IOS Press, 2013.
[11] B. Zhang, J. Qiu. “High Performance Clustering of Social Images in a Map-Collective Programming Model. Poster in SoCC, 2013.
[12] E. Chan, M. Heimlich, A. Purkayastha, and R. Geijn. “Collective communication: theory, practice, and experience”. Concurrency and Computation: Practice and Experience 19 (13), 2007.
[13] C.-T. Chu et al. “Map-Reduce for Machine Learning on Multicore”. NIPS, 2006.
[14] Harp. http://salsaproj.indiana.edu/harp/index.html
[15] Y. Low et al. “Distributed GraphLab: A Framework for Machine Learning and Data Mining in the Cloud”. PVLDB, 2012.
[16] J. Gonzalez et al. “PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs”. OSDI, 2012.
[17] R. Xin et al. “GraphX: A Resilient Distributed Graph System on Spark”. GRADES, SIGMOD workshop, 2013.
[18] M. Chowdhury et al. “Managing Data Transfers in Computer Clusters with Orchestra”. SIGCOM, 2011.
[19] T. Gunarathne, J. Qiu and D. Gannon. “Towards a Collective Layer in the Big Data Stack”. CCGrid, 2014.
[20] T. Fruchterman, M. Reingold. “Graph Drawing by Force-Directed Placement”, Software Practice & Experience 21 (11), 1991.
[21] Y. Ruan et al. “A Robust and Scalable Solution for Interpolative Multidimensional Scaling With Weighting”. E-Science, 2013.
[22] MPI Forum. “MPI: A Message Passing Interface”. SC, 1993.
[23] Big Red II. https://kb.iu.edu/data/bcqt.html
[24] G. Fox. “Robust Scalable Visualized Clustering in Vector and non Vector Semimetric Spaces”. Parallel Processing Letters 23, 2013.
[25] X. Gao and J. Qiu. “Social Media Data Analysis with IndexedHBase and Iterative MapReduce”. Workshop on Many-Task Computing on Clouds, Grids, and Supercomputers (MTAGS), SC, 2013.
[26] Y. Ruan et al. “Integration of Clustering and Multidimensional Scaling to Determine Phylogenetic Trees as Spherical Phylograms Visualized in 3 Dimensions”. C4Bio, CCGrid worksop, 2014.