**Towards a Collective Layer in the Big Data Stack**

Authors Name/s per 1st Affiliation *(Author)*

line 1 (of *Affiliation*): dept. name of organization

line 2: name of organization, acronyms acceptable

line 3: City, Country

line 4: e-mail: name@xyz.com

Authors Name/s per 2nd Affiliation *(Author)*

line 1 (of *Affiliation*): dept. name of organization

line 2: name of organization, acronyms acceptable

line 3: City, Country

line 4: e-mail: name@xyz.com

*Abstract*—

In this paper we present our ongoing efforts to identify and implement collective communication patterns for optimization of data communication and data reductions in iterative MapReduce applications. So far we have identified MapReduceMerge, AllGather, AllReduce and Scatter patterns. These patterns have the ability to improve performance while facilitating ease of use for the users. Our goal is to identify, define and implement a set of commonly agreed set of collective communications primitives that can be used across many different MapReduce or iterative MapReduce frameworks.

We present prototype implementations of Map-AllGather and Map-AllReduce primitives for Twister4Azure and Hadoop (called H-Collectives). We achieved up to 33% improvement for KMeansClustering and up to 50% improvement with Multi-Dimensional Scaling in addition to the improved user friendliness. In some case, collective communication operations virtually eliminated almost all the overheads of the computations.

Keywords-component; formatting; style; styling; insert (key words)

#  Introduction

During the last decade three interconnected disruptions mainly driven by the industry altered the landscape of scalable parallel computing, which has long been dominated by the HPC applications. These disruptions are the emergence of data intensive computing (aca big data), the new generation of programming, commodity cluster based execution and storage frameworks such as MapReduce and the utility computing model introduced by Cloud computing environments. Oftentimes MapReduce is used to process the “Big Data” in cloud or cluster environments. While these disruptions have fueled major innovations and have caused a massive progress in the area of scalable parallel computing, there still exists many opportunities to improve these toolsets and technologies to extract the maximum out of the compute, data and human resources. In this paper we present a set of collective communications primitives that extend the MapReduce model to improve the efficiency and usability of large scale parallel data intensive computations on cloud computing environments as well as traditional clusters.

When performing distributed computations, often times the data needs to be shared and/or consolidated among the different nodes of the computations. Collective communication primitives effectively facilitate these data communications by providing operations that involve a group of nodes simultaneously[[1](#_ENREF_1), [2](#_ENREF_2)], rather than one by one. Collective communication primitives are very popular in the HPC community and used heavily in MPI type of HPC applications. A lot of research [[1](#_ENREF_1)] had been done to optimize the performance of these collective communication operations, as they have a large impact on the performance of HPC applications.

In this paper we identify several collective communication primitives to support and optimize common computation and communication patterns that we notice in iterative MapReduce computations. we present the applicability of collective communication operations to Iterative MapReduce without sacrificing the desirable properties of MapReduce such as fault tolerance, scalability, familiar API’s and data model, etc. Addition of collective communication operations enriches the iterative MapReduce model by providing many performance and ease of use advantages such as providing efficient data communication operations optimized for the particular execution environment and use case, providing programming models that fit naturally with application patterns and allowing users to avoid overheads by skipping unnecessary steps of the execution flow.

We present these patterns as high level constructs that can be adopted by any MapReduce and iterative MapReduce run time. We present proof-of-concept implementation of the primitives on Hadoop and Twister4Azure as the first steps and envision a future where all the Mapreduce and iterative MapReduce runtimes would support a commonly agreed set of collective communication primitives.

The solutions presented in this paper focus on mapping All-to-All type of collective communication operations, AllGather and AllReduce to the MapReduce model as Map-AllGather and Map-AllReduce patterns. Map-AllGather gathers the outputs from all the map tasks and distributes the gathered data to all the workers after a combine operation. Map-AllReduce primitive combines the results of the Map Tasks based on a reduction operation and delivers the result to all the workers. We also present the MapReduceMergeBroadcast as a canonical model representative of the most of the iterative MapReduce frameworks.

# MapReduce-MergeBroadcast (MR-MB)

In this section we introduce the MapReduce-MergeBroadcast[[3](#_ENREF_3)], called MR-MB here onwards, as a generic programming model to represent data-intensive iterative MapReduce applications. Programming models of most of the current iterative MapReduce frameworks can be specified as the MR-MB.

## API

MR-MB programming model extends the *map* and *reduce* functions of traditional MapReduce to include the loop variant delta values as an input parameter. MR-MB provides the loop variant data (*dynamicData*), including broadcast data, to the *Map* and *Reduce* tasks as a list of key-value pairs using this additional input parameter.

 *Map(<key>, <value>, list\_of <key,value> dynamicData)*

 *Reduce(<key>,list\_of<value>,list\_of<key,value> dynamicData)*

## Merge Task

The *Merge[*[*3*](#_ENREF_3)*]* was defined as a new step to the MapReduce programming model to support iterative applications. It is a single task, or the convergence point, which executes after the *Reduce* step that can be used to perform summarization or aggregation of the results of a single MapReduce iteration. The *Merge* step can also serve as the “loop-test” that evaluates the loops condition in the iterative MapReduce programming model.

*Merge* Task receives all the *Reduce* outputs and the broadcast data for the current iteration as the inputs. With *merge*, the overall flow of the iterative MapReduce computation and data flow would appear as follows:

 *Map →Combine→Shuffle→Sort→Reduce→Merge→Broadcast*

Following is the programming APIs of the Merge task.

*Merge(list\_of <key,list\_of<value>> reduceOutputs,*

*list\_of <key,value> dynamicData)*

## Broadcast

Boradcast operation broadcasts the loop variant data to all the tasks in an iteration. In typical data-intensive iterative computations, the loop-variant data is orders of magnitude smaller than the loop-invariant data. Broadcast operation typically broadcasts the output data of the Merge tasks to the tasks of the next iteration. Broadcast operation of MR-MB can also be thought of as executing at the beginning of the iterative MapReduce computation. This would make the model Broadcast-MapReduce-Merge, which is essentially similar to the MapReduce-Merge-Broadcast when iterations are present (eg: …MRn-> Mergen-> Broadcastn-> MRn+1-> Merge n+1->...). Broadcast can be implemented efficiently based on the environment as well as the data sizes. Well known algorithms for data broadcasting include flat-tree, minimum spanning tree (MST), pipeline and chaining[[4](#_ENREF_4)]. It’s possible to share broadcast data between multiple Map and/or reduce tasks executing on the same node.

## Current iterative MapReduce Frameworks and MR-MB

Twister4Azure[[3](#_ENREF_3)] supports the MR-MB natively. In Twister, the combine step is part of the driver program and is executed after the MapReduce computation of every iteration. Twister is a MapReduce-Combine model, where the Combine step is similar to the Merge step. Twister[[5](#_ENREF_5)] MapReduce computations broadcast the loop variant data products at the beginning of each iteration, effectively making the model Broadcast-MapReduce-Combine, which is semantically similar to the MR-MB. HaLoop[[6](#_ENREF_6)] performs an additional MapReduce computation to do the fixed point evaluation for each iteration, effectively making this MapReduce computation equivalent to the Merge task. Data broadcast is achieved through a MapReduce computation to join the loop variant and loop invariant data.

# Collective Communications Primitives for Iterative MapReduce

While implementing iterative MapReduce applications using the MR-MB model, we started to notice several common execution flow patterns across the different applications. Some of these applications had very trivial Reduce and Merge tasks while some of the applications needed extra effort to map to the MR-MB model owing to the execution patterns being slightly different than the iterative MapReduce pattern. In order to solve such issues, we introduce Map-Collectives communications primitives to the iterative MapReduce programming model, inspired by the MPI collective communications primitives[[2](#_ENREF_2)].



1. Collective Communication primitives

These primitives support higher-level communication patterns that occur frequently in data-intensive iterative applications by substituting certain steps of the MR-MB computation. As depicted in figure 1, these collective primitives can be thought of as a Map phase followed by a series of framework-defined communication and computation operations leading to the next iteration.

In this paper we propose two collective communication primitive implementations: Map-AllGather and Map-AllReduce. You can also identify MR-MB as another collective communication primitive as well.

## Requirements

When designing collective communication primitives for iterative MapReduce, we should make sure those fit with the MapReduce data model and the MapReduce computational model, which supports multiple Map task waves, large overheads, significant variations and inhomogeneous tasks. Also the primitives should retain the scalability, while keeping the programming model simple and easy to understand. These primitives should be designed to maintain the same type of framework-managed excellent fault tolerance supported by the MapReduce frameworks.

## Advantages

### Performance improvement

Introduction of collective communication primitives provides several types of performance improvements to the iterative MapReduce applications.

Collective primitives can skip or overlap certain steps (eg: shuffle, reduce, merge) of the iterative MapReduce computational flow reducing the overheads. Collective communication patterns would also fit more naturally with the application patterns, avoiding the need for unnecessary steps.

Another advantage is the ability of the frameworks to optimize these operations transparently for the users, even allowing the possibility of different optimizations (poly-algorithm) for different use cases and environments. For an example, a communication algorithm that’s best for smaller data sizes may not be the best for larger data sizes. In such cases, the collective communication operations can opt to have multiple algorithm implementations to be used for different data sizes. These primitives also have the capability to make the applications more efficient by overlapping communication with computation. Frameworks can start the execution of collectives as soon as the first results are produced from the Map tasks. For example, in the Map-AllGather primitive we present in section 4, partial Map results are broadcasted to all the nodes as soon as they become available. It is also possible to perform some of the computations in the data transfer layer, like the hierarchical reduction in Map-AllReduce primitive using a reduction tree.

### Ease of use

These primitive operations make life easier for the application developers by presenting them with patterns and APIs that fit more naturally with their applications. This simplifies the porting of new applications to the iterative MapReduce model.

In addition, by using the Map-Collective operations, the developers can avoid manually implementing the logic of some of the operations, such as reduce and merge tasks, for each application and can rely on optimized operations provided by the framework.

### Scheduling with iterative primitives

Collective primitives also give us the ability to propagate the scheduling information for the next iteration along with the collective communication data. These primitives can schedule the tasks of a new iteration or application through the primitives by delivering information about the new tasks to the worker nodes. This reduces the scheduling overheads for the tasks of the new iteration. For an example, Twister4Azure successfully employs this strategy, together with the caching of task metadata, to schedule new iterations with minimal overhead.

## Programming model

Iterative MapReduce collective communication primitives can be specified as an outside configuration option without changing the MapReduce programming model. This permits the applications developed with Map-Collectives to be backward compatible with frameworks that don’t support the collectives. This also makes it easy for developers who are already familiar with MapReduce programming to use collectives.

## Implementations

Map-Collectives can be add-on improvements to MR frameworks. The simplest implementation would be implementing the primitives using the current MapReduce API and communication model on the user level, then providing the implementation as a library. This will achieve ease of use for the users and in addition help with the performance.

More optimized implementations can implement these primitives as part of the MapReduce framework (or as a separate library) with the ability to optimize the data transfers based on environment and use case using collective communications across the worker nodes.

It is not our objective in this paper to find the most optimal implementations for each of the environments, rather to present a sufficiently optimal implementation for each of the primitive and for each of the environments, to show the performance efficiencies that can be gained through using even a modest implementation of these operations. We leave finding of the most optimized methods for each environment as a future work.

In the case of cloud executions, we should also note that finding most optimal implementations for cloud environments might end up being a moving target as cloud environments evolve very rapidly. Also it is harder for the outsiders due to the black box nature of cloud environments. This presents an interesting opportunity for cloud providers to provide optimized implementations of these primitives as cloud infrastructure services that can be used by the framework developers.

# Map-AllGather Collective

AllGather is an all-to-all collective communication operation that gathers data from all the workers and distribute the gathered data to all the workers [[1](#_ENREF_1)]. We can notice the AllGather pattern in data-intensive iterative MapReduce applications where the “reduce” step is a simple aggregation operation that simply aligns the outputs of the Map Tasks together in order, followed by “merge” and broadcast steps that transmit the assembled output to all the workers. An example would be a MapReduce computation that generates a matrix as the loop variant data, where each map task outputs several rows of the resultant matrix. In this computation we would use the Reduce and Merge tasks to assemble the matrix together and then to broadcast the assembled matrix.

Example data-intensive iterative applications that have the AllGather pattern include MultiDimensionalScaling,…

## Model

We developed a Map-AllGather iterative MapReduce primitive similar to the MPI AllGather[[1](#_ENREF_1)] collective communication primitive to support applications with communication patterns similar to the above in a more efficient manner.

|  |  |  |  |
| --- | --- | --- | --- |
| **Pattern** | **Execution and communication flow** | **Frameworks** | **Sample applications** |
| **MapReduce** | Map->Combine->Shuffle->Sort->Reduce | Hadoop, Twister, Twister4Azure | WordCount, Grep, etc. |
| **MapReduce-MergeBroadcast** | Map->Combine->Shuffle->Sort->Reduce->Merge->Broadcast | Twister, Haloop, Twister4Azure | KMeansClustering, PageRank, |
| **Map-AllGather** | Map->AllGather Communication->AllGather Combine | H-Collectives, Twister4Azure | MDS-BCCalc |
| **Map-AllReduce** | Map->AllReduce (communication + computation) | H-Collectives, Twister4Azure | KMeansClustering, MDS-StressCalc |

### Execution model

Map-AllGather primitive broadcasts the Map Task outputs to all computational nodes (all-to-all communication) of the current computation, and then assembles them together in the recipient nodes as depicted in figure 2. Each Map worker will deliver its result to all other workers of the computation once the Map task is completed.

The computation and communication pattern of a Map-AllGather computation is Map phase followed by AllGather communication (all-to-all) followed by the AllGather combine phase. As we can notice, this model substitute the shuffle->sort->reduce->merge->broadcast steps of the MR-MB with all-to-all broadcast and Allgather combine.



1. Map-AllGather Collective

### Data Model

For Map-AllGather the map output key should be an integer specifying the location of the output value in the resultant gathered data product. Map output values can be vectors, set of vectors (partial matrix) or single values. Final output value of the Map-AllGather operation is an assembled array of Map output values in the order of their corresponding keys. The result of AllGather-Combine will be provided to the Map tasks of the next iteration as the loop variant data using the API’s and mechanisms suggested in section 2.2.1.

The final assembly of AllGather data can be performed by implementing a custom combiner or using the default combiner of AllGather-combine. Custom combiner allows the user to specify a custom assembling function. In this case, the input to the assembling function is a list of Map outputs key-value pairs, ordered by the key. This assembling function gets executed in each worker node after all the data is received.

The default combiner should work for most of the use cases as the combining of AllGather data is oftentimes a trivial process. The default combiner expect the Map outputs to be in <int, double[]> format. The map output key represents the location of the corresponding value in the assembled value. In the above mentioned Matrix example the key would represent the row index of the output matrix and the value would contain the corresponding row vector. Map outputs with duplicate keys (same key for multiple output values) are not supported and ignored.

Users can use their Mapper implementations as it is with Map-AllGather primitive. User just have to specify the collective operation and then the shuffle and reduce phases of MapReduce would be replaced by the AllGather communication and computations.

### Cost Model

An optimized implementation of AllGather, such as a by-directional exchange based implementation[[1](#_ENREF_1)], we can estimate the cost of the AllGather component as following (using the Hockney model[[4](#_ENREF_4), [7](#_ENREF_7)] where α is the latency and β is the transmission time per data item (1/bandwidth)), where *m* is the number of map tasks.

It’s also possible to further reduce this cost by performing local aggregation in the Map worker nodes. The variation of Map task completion times will also help to avoid the network congestion in these implementations.

Map-Allgather substitute the Map output processing (collect, spill, merge), Reduce task (shuffle, merge, execute, write), Merge task (shuffle, execute) and broadcast overheads with a less costly AllGather operation. The information contained in the AllGather transfers can be used to aid in scheduling the tasks of the next iteration.

## Fault tolerance

All-Gather partial data product transfers from Map tasks can fail due to communication and other failures. When task level fault tolerance (typical MapReduce fault tolerance) is enabled, it’s possible for the workers to read the missing map output data from the persistent storage (eg:HDFS) to successfully perform the All-Gather computation.

The fault tolerance model and the speculative execution model of MapReduce enables duplicate execution of tasks. Duplicate data detection can be performed before the final assembly of the data at the recipient nodes.

## Benefits

Use of the AllGather primitive in an iterative MapReduce computation eliminates the need for reduce, merge and the broadcasting steps in that particular computation. Additionally the smaller sized multiple broadcasts of our Map-AllGather primitive originating from multiple servers of the cluster would be able to use the network more effectively than a single larger broadcast originating from a single server.

In typical MapReduce computations, often times the Map task execution times are inhomogeneous[[8](#_ENREF_8)]. Implementations of AllGather primitive can start broadcasting the map task result values as soon as the first map task is completed. This mechanism ensures that almost all the data is broadcasted by the time the last map task completes its execution, resulting in overlap of computations with communication. This benefit will be even more significant when we have multiple waves of map tasks.

In addition to improving the performance, this primitive also enhances the usability as it eliminates the overhead of implementing reduce and/or merge functions. Map-AllGather can be used to efficiently schedule the next iteration or the next application of the computational flow as well.

## Implementations

In this section we present two implementations of the Map-AllGather primitive. These implementations are proof of concepts to show the advantages achieved by using the Map-AllGather primitive. It’s possible to further optimize them using more advanced communication algorithms, based on the environment they will be executing, the scale of the computations, and the data sizes as shown in MPI collective communications literature[[1](#_ENREF_1)]. One of the main advantages of these primitives is the ability to improve primitive implementations without the need to change the user application, making it possible to optimize these implementations in the future.

### H-Collectives Map-AllGather

H-Collectives Map-AllGather primitive is implemented using the Netty NIO library on top of Apache Hadoop. This implementation performs simple TCP-based best effort broadcasts for each Map task output. If a data product is not received through the TCP broadcasts, then it would be fetched from the HDFS. The tasks for the next iteration are already speculatively scheduled and waiting to start as soon as all the AllGather data is received, getting rid of most of the Hadoop job startup/cleanup and task scheduling overheads. We can’t do sepculative scheduling with Hadoop MapReduce as we need to add the loop variant data (only available after the previous iteration is finished) to the Hadoop DistributedCache when scheduling the Job.

### Twister4Azure Map-AllGather

We implemented Map-AllGather in Twister4Azure using the Windows Communication Foundation (WCF)-based Azure TCP inter-role communication mechanism, while using the Azure table storage as a persistent backup. It performs simple TCP-based broadcasts for each Map task output, which is an all-to-all linear implementation. Workers start transmitting the data as soon as a task is completed taking advantage of the inhomogeneous Map task completion times. More sophisticated implementations can utilize minimum spanning tree based broadcast algorithms or a pairwise exchange based algorithm. However, the performance of these optimized algorithms may be hindered by the fact that all data will not be available at the same time.

# Map-AllReduce Collective

AllReduce is a collective pattern which combines a set of values emitted by all the workers based on a given operation and makes the results available to all the workers[[1](#_ENREF_1)]. This pattern can be seen in many iterative data mining and graph processing algorithms. Example data-intensive iterative applications that have the Map-AllReduce pattern include KMeansClustering, Multi-dimensional Scaling StressCalc computation and PageRank using out links matrix.

## Model

We propose Map-AllReduce iterative MapReduce primitive similar to the MPI AllReduce[[1](#_ENREF_1)] collective communication operation, to efficiently aggregate and reduce the results of the Map Tasks.

### Execution Model

The computation and communication pattern of a Map-AllReduce computation is Map phase followed by the AllReduce communication and computation (reduction) as depicted in figure 3. As we can notice, this model allows us to substitute the shuffle->sort->reduce->merge->broadcast steps of the MR-MB with AllReduce communication in the communication layer. AllReduce phase can be implemented efficiently using algorithms such as bidirectional exchange (BDE) [[1](#_ENREF_1)] or hierarchical tree based reduction.



1. Map-AllReduce collective

Map-AllReduce also allows the implementations to perform local aggregation on the worker nodes across multiple map tasks and to perform hierarchical reduction of the Map Task outputs while delivering them to all the workers.

### Data Model

For Map-AllReduce, the map output values should be vectors or single values of numbers. The values belonging to each distinct map output key is processed as a separate data reduction operation. Output of the Map-AllReduce operation is a list of key value pairs where each key corresponds to a map output key and the value is the combined value of the map output values that were associated with that map output key. As shown in figure 4 the number of records in the Map-AllReduce output is equal to the number of unique map output keys. For example, if the Map tasks output 10 distinct keys, then the Map-AllReduce would result in 10 combined vectors or values. Map output value type should be a number. In addition to the summation, any commutative and associative operation can be performed using this primitive. Example operations include sum, max, min, count, and product operations. Operations such as average can be performed by using Sum operation together with an additional element (dimension) to count the number of data products. Due to the associative and commutative nature of the operations, AllReduce has the ability to start combining the values as soon as at least one mapper completes the execution. It also allows the AllReduce implementations to use reduction trees or bidirectional exchanges to optimize the AllReduce computation and communication.



1. Example Map-AllReduce with Sum operation

It is also possible to allow users to specify a post process function that executes after the AllReduce communication. This function can be used to perform a simple operation on the AllReduce result or to check for the iteration termination condition. This function would get executed in each worker node after all the AllReduce data has been received by them.

*list<Key, IOpRedValue> postOpRedProcess(*

*list<Key, IOpRedValue> opRedResult);*

### Cost Model

An optimized implementation of AllReduce, such as a by-directional exchange based implementation[[1](#_ENREF_1)], will reduce the cost of the AllReduce component to,

It’s also possible to further reduce this cost by performing local aggregation and reduction in the Map worker nodes as the compute cost of AllReduce is very small. Map-AllReduce substitute the Map output processing (collect, spill, merge), Reduce task (shuffle, merge, execute, write), Merge task (shuffle, execute) and broadcast overheads with a less costly AllReduce operation.

## Fault Tolerance

If the All-Reduce communication step fails for some reason, it’s possible for the workers to read map output data from the persistent storage to perform the All-Reduce computation.

The fault tolerance model and the speculative execution model of MapReduce make it possible to have duplicate execution of tasks. Duplicate executions can result in incorrect Map-AllReduce results due to the possibility of aggregating the output of the same task twice as Map-AllReduce starts the data reduction as soon as the first Map task output is present. The most trivial fault tolerance model for AllReduce would be a best-effort mechanism, where the AllReduce implementation would fall back to using the Map output results from the persistent storage (eg: HDFS) in case duplicate results are detected. Duplicate detection can be performed by maintaining a set of map ID’s with each combined data product.

It’s possible for the frameworks to implement richer fault tolerance mechanisms, such as identifying the duplicated values in localized areas of the reduction tree.

## Benefits

This primitive will reduce the work each user has to perform in implementing Reduce and Merge tasks. It also removes the overhead of Reduce and Merge tasks from the computations and allows the framework to perform the combine operation in the communication layer itself.

Map-AllReduce semantics allow the implementations to perform hierarchical reductions, reducing the size and the number of intermediate data communications and optimizing the computation. The hierarchical reduction can be performed in as many levels as needed based on the size of the computation and the scale of the environment. For example, first level in mappers, second level in the node and nth level in rack level, etc. The mapper level would be similar to the “combine” operation of vanilla map reduce. The local node aggregation can combine the values emitted by multiple mappers running in a single physical node. All-Reduce combine processing can be performed in real time when the data is received.

## Implementations

In this section we present two implementations of the Map-AllReduce primitive. These implementations are proof of concepts to show the advantages achieved by using the Map-Reduce primitive. One of the main advantages of these primitives is the ability to improve the implementations in the future without any need to change the user application implementations. Current implementations use n'ary tree based hierarchical reductions. Other algorithms to implement AllReduce include ﬂat-tree/linear, pipeline, binomial tree, binary tree, and k-chain trees[[4](#_ENREF_4)].

### H-Collectives Map-AllReduce

H-Collectives Map-AllReduce primitive is implemented on top of Apache Hadoop using node-level local aggregation and using the Netty NIO library to broadcast the locally aggregated values to the other worker nodes of the computation. The final reduce combine operation is performed in each of the worker nodes and is done after all the Map tasks are completed and the data is transferred. A single worker node may run several Map workers and many more map tasks belonging to the computation. The Hadoop Map-AllReduce implementation maintains a node-level cache to serve the AllReduce results to these tasks.

### Twister4Azure Map-AllReduce

Current implementation uses a hierarchical processing approach where the results are first aggregated in the local node and then final assembly is performed in the destination nodes. Twister4Azure Map-AllReduce implementation maintains a worker node-level cache of the AllReduce result values to server the results to multiple Map workers and map tasks running on a single server.

The iteration check happens in the destination nodes and can be specified as a custom function or as a limit on the number of iterations.

# Evaluation

All the Hadoop and H-Collectives experiments were conducted in the FutureGrid Alamo cluster which has Dual Intel Xeon X5550 (8 total cores) per node, 12GB Ram per node and a 1Gbps network. All the Twister4Azure tests were performed in Windows Azure cloud using Azure extra-large instances. Azure extra-large instances provide 8 compute cores and 14GB memory per instance.

In the figures, Map overhead is the start and cleanup overhead for each map task. Scheduling is the per iteration (per MapReduce job) startup and task scheduling time. Cleanup is the per iteration (per job) overhead from reduce task execution completion to the iteration (job) end. Map Variation includes the time due to variation of data load, compute and map overhead times. "Comm+Red+Merge" includes the time for map to reduce data shuffle, reduce execution, merge and broadcast. "Compute" and "data load" times are calculated using the average compute only and data load times across all the tasks of the computation. We plot the common components (data load, compute) at the bottom of the graph to highlight variable components.

## Multi-Dimensional Scaling using Map-AllGather

The objective of Multi-Dimensional Scaling (MDS) is to map a data set in high-dimensional space to a user-defined lower dimensional space with respect to the pairwise proximity of the data points[[9](#_ENREF_9)]. In this paper, we use parallel[[10](#_ENREF_10)] Scaling by MAjorizing a COmplicated Function (SMACOF)[[11](#_ENREF_11)], an iterative majorization algorithm. The input for MDS is an N\*N matrix of pairwise proximity values. The resultant lower dimensional mapping in D dimensions, called the X values, is an N\*D matrix.

Unweighted MDS results in two MapReduce jobs per iteration, BCCalc and StressCalc. Each BCCalc Map task generates a portion of the total X matrix. The reduce step of MDS BCCalc computation is an aggregation operation, which simply assembles the outputs of the Map tasks together in order. This X value matrix is then broadcasted to be used by the StressCalc step of the current iterations as well as in the BCCalc step of the next iteration. MDS performs relatively smaller amount of computations for a unit of input data. Hence MDS has larger data loading and memory overheads.

Usage of the Map-AllGather primitive in MDS BCCalc computation eliminates the need for reduce, merge and the broadcasting steps in that particular computation.

### H-Collectives MDS Map-AllGather

We implemented the MDS for Hadoop using vanilla MapReduce and using H-Collectives Map-AllGather primitive. Vanilla MapReduce implementation uses the Hadoop DistributedCache to broadcast loop variant data to the Map tasks.

Figure 5 shows the MDS (BC calculation only) strong scaling performance results highlighting the overhead of different phases on the computation. We only used the BC Calculation step of the MDS in each iteration and skipped the stress calculation step to further highlight the AllGather component. This test case scales a 51200\*51200 matrix and in to a 51200\*3 matrix.



1. MDS Hadoop using only the BC Calculation MapReduce job per iteration to highlight the overheads. 20 iterations, 51200 data points

As we can see, the H-Collectives implementation gets rid of the communication, reduce, merge, task scheduling and job cleanup overheads of the vanilla MapReduce computation. However, we notice a slight increase of Map task overhead and Map variation in the case H-Collectives Map-AllReduce based implementation. We believe these increases are due to the rapid scheduling of Map tasks across successive iterations in H-Collectives, whereas in the case of vanilla MapReduce the map tasks of successive iterations have few seconds between the scheduling.

### Twister4Azure MDS Map-AllGather



1. MDS application implemented using Twister4Azure. 20 iterations. 51200 data points (~5GB).

We implemented MDS for Twister4Azure using Map-AllGather primitive and MR-MB with optimized broadcasting. Twister4Azure optimized broadcast version is an improvement over vanilla MapReduce where it uses an optimized tree-based algorithm to perform TCP broadcasts of in-memory data. Figure 6 shows the MDS (with both BCCalc and StressCalc steps) strong scaling performance results comparing the Twister4Azure Map-AllGather based implementation with the MR-MB implementation. The number of map tasks per computation is equal to the number of total cores of the computation. The Map-AllGather based implementation improves the performance of Twister4Azure MDS over MapReduce with optimized broadcast by 13% up to 42% for the current test cases.

### Detailed analysis of overheads

In this section we perform detailed analysis of overheads of the Hadoop MDS BCalc calculation using a histogram of number of executing Map Tasks. In this test, we use only the BCCalc MapReduce job. MDS computations in this section use 51200 \*51200 data points, 6 Iterations on 64 cores using 64 Map tasks per iteration. The total AllGather data size of this computation is 51200\*3 data points. Average data load time is 10.61 seconds per map task. Average actual MDS BCCalc compute time is 1.5 seconds per map task.

Figure 7 and 8 plot the total number of executing Map tasks at a given moment of the computation. Number of executing Map tasks approximately represent the amount of useful work done in the cluster at that given moment. The resultant graphs comprise of blue bars that represent an iteration of the computation. The width of each blue bar represents the time spent by Map tasks in that particular iteration. This includes the time spent loading Map input data, Map calculation time and time to process and store Map output data. The space between the blue bars represents the overheads of the computation.

Figure 7 presents the MDS using Hadoop MapReduce. Figure 8 presents MDS using H-Collectives AllGather implementation. In H-Collectives, Hadoop driver program performs speculative (overlap) scheduling of iterations by scheduling the tasks for the next iteration while the previous iteration is still executing. Stripped section on each blue bar represent the data loading time (time it takes to read input data from HDFS). Overheads of this computation include Allgather communication and task scheduling. As we can notice, the overheads between the iterations virtually disappear with the use of AllGather primitive.



1. Hadoop MapReduce MDS-BCCalc histogram



1. H-Collectives AllGather MDS-BCCalc histogram

### Twister4Azure vs Hadoop

Twister4Azure is already optimized for iterative MapReduce[[3](#_ENREF_3)] and contains very low scheduling, data loading and data communication overheads compared to Hadoop. Hence, the overhead reduction we achieve by using collective communication is comparatively less in Twister4Azure compared to Hadoop. Also a major component of Hadoop MDS Map task cost is due to the data loading, as you can notice in figure 8. Twister4Azure avoids this cost by using data caching and cache aware scheduling.

## K-meansClustering using Map-AllReduce

K-means clustering[[12](#_ENREF_12)] is often implemented using an iterative refinement technique where each iteration performs two main steps: the cluster assignment step and the centroids update step. In a typical MapReduce implementation, the assignment step is performed in the Map task and the update step is performed in the Reduce task, while centroid data is broadcasted at the beginning of each iteration.

K-means Clustering centroid update step is an AllReduce computation. In this step all the values (data points assigned to a certain centroid) belonging to each key (centroid) needs to be combined independently and the resultant key-value pairs (new centroids) are distributed to all the Map tasks of the next iteration. KMeansClustering has relatively smaller data loading and memory overheads vs the number of computations compared to the MDS application discussed above.

### H-Collectives KMeansClustering-AllReduce

We implemented the K-means Clustering application for Hadoop using the Map-AllReduce primitive and using vanilla MapReduce. The vanilla MapReduce implementation uses in-map combiners to perform aggregation of the values to minimize the size of map-to-reduce data transfers.



1. Hadoop K-means Clustering comparison with H-Collectives Map-AllReduce Weak scaling. 500 Centroids,20 Dimensions. 10 iterations.

Figure 9 illustrates the Hadoop K-means Clustering weak scaling performance results in which we scale the computations while keeping the workload per core constant. Figure 10 is the Hadoop K-means Clustering strong scaling performance with the number of cores scaled while keeping the data size constant. Strong scaling test cases with smaller number of nodes use more map task waves optimizing the intermediate communication, resulting in relatively smaller overheads for the computation. (HDFS replication factor of 6 was used to increase the data locality).

As we can see, the H-Collectives implementation gets rid of the communication, reduce, merge, task scheduling and job cleanup overheads of the vanilla MapReduce computation. However, we notice a slight increase of Map task overhead and Map variation in the case H-Collectives Map-AllReduce based implementation. We believe these increases are due to the rapid scheduling of Map tasks across successive iterations in H-Collectives, whereas in the case of vanilla MapReduce the map tasks of successive iterations have few seconds between the scheduling.



1. Hadoop K-means Clustering comparison with H-Collectives Map-AllReduce Strong scaling. 500 Centroids,20 Dimensions,10 iterations.

### Twister4Azure KmeansClustering-AllReduce



1. Twister4Azure K-means weak scaling with Map-AllReduce. 500 Centroids, 20 Dimensions. 10 iterations. 32 to 256 Million data points.



1. K-means Clustering strong scaling. 500 Centroids, 20 Dimensions, 10 iterations. 128Million data points.

We implemented the K-means Clustering application for Twister4Azure using the Map-AllReduce primitive and vanilla MR-MB. The vanilla MapReduce implementation uses in-map combiners to perform local aggregation of the values to minimize the size of map-to-reduce data transfers. Figure 11 shows the K-means Clustering weak scaling performance results, where we scale the computations while keeping the workload per core constant. Figure 12 presents the K-means Clustering strong scaling performance, where we scaled the number of cores while keeping the data size constant. As we can see in these figures, the Map-AllReduce implementation gets rid of the communication, reduce and merge overheads of the vanilla MapReduce computation.

# Background and Related Works

## Collective Communication Primitives

Collective communication operations[[2](#_ENREF_2)] facilitate the optimized communication and coordination between groups of nodes of a distributed computations and are used heavily in MPI type of HPC applications. These powerful operations make it much easier and efficient to perform complex data communications inside the distributed parallel applications. Collective communication also implicitly provides some form of synchronization across the participating tasks. There exist many different implementations of HPC collective communication primitives supporting many different algorithms and topologies suiting the different environments and different use cases. The best collective implementation for a given scenario depends on many factors including message size, number of workers, topology of the system, the computational capabilities/capacity of the nodes and etc. Oftentimes collective communication implementations follow a poly-algorithm approach to automatically select the best algorithm and topology for the given scenario.

Data redistribution communication primitives can be used to distribute and share data across the worker processors. Examples of these include broadcast, scatter, gather, and allgather operations. Data consolidation communication primitives can be used to collect and consolidate data contributions from different workers. Examples of these include reduce, reduce-scatter and allreduce.

We can also categorize collective communication primitives based on the communication patterns as well.

* All-to-One: gather, reduce
* One-to-All : broadcast, scatter
* All-to-All : allgather, allreduce, reduce-scatter
* Synchronization : Barrier

MapReduce model supports the All-to-One type communications through the Reduce step. Broadcast operation introduced in MapReduce-MergeBroadcast model (section II) serves as an alternative to the One-to-All type collective communication operations. MapReduce model contains a barrier between the Map and Reduce phases and the iterative MapReduce model introduces a barrier between the iterations as well. The solutions presented in this paper focus on introducing All-to-All type collective communication operations to the MapReduce model.

We can implement All-to-All communications using pairs of existing All-to-One and One-to-All type operations present in the MapReduce-MergeBroadcast model. For an example, AllGather operation can be implemented as Reduce-Merge followed by Broadcast. However, these types of implementations would be inefficient and would be harder to use compared to dedicated optimized implementations of All-to-All operations.

## MapReduce

MapReduce consist of a programming model, storage architecture and an associated execution framework for distributed processing of very large data sets introduced by Google MapReduce[[13](#_ENREF_13)]. MapReduce frameworks take care of data partitioning, task parallelization, task scheduling, fault tolerance, intermediate data communication and many other aspects of the computations for the users, while providing an easy to use programming model allowing users with no background or experience in distributed and parallel computing to utilize MapReduce and the distributed infrastructures to easily process large volumes of data.

MapReduce frameworks are typically not optimized for the best performance or parallel efficiency of small scale applications. Main goals of MapReduce frameworks include framework managed fault tolerance, ability run on commodity hardware, ability to process very large amounts of data and horizontal scalability of compute resources.

## Iterative MapReduce

Data intensive iterative computations, where individual iterations can be specified as MapReduce computations, are a subset of distributed parallel computations suited for execution in cloud environments. Examples of such applications that are implemented using iterative MapReduce include PageRank, Multi-Dimensional Scaling[[14](#_ENREF_14), [15](#_ENREF_15)], K-means Clustering, Descendent query[[6](#_ENREF_6)], LDA, and Collaborative Filtering with ALS-WR.

These data-intensive iterative computations can be performed using traditional MapReduce frameworks like Hadoop by taking care of the iterative aspects in the job client driver, albeit in an un-optimized manner. However, many optimizations and programming model improvements are available for better performance and usability of the iterative MapReduce programs. Such optimization opportunities are highlighted by the development of many iterative MapReduce and similar frameworks such as Twister[[5](#_ENREF_5)], HaLoop[[6](#_ENREF_6)], Twister4Azure[[15](#_ENREF_15)],Daytona[[16](#_ENREF_16)], i-mapreduce, and spark[[17](#_ENREF_17)]. Current optimizations for iterative MapReduce exploited by these frameworks include caching of the loop-invariant data, cache aware scheduling of tasks, iterative aware programming models, direct memory streaming of intermediate data, iteration-level fault tolerance, caching of intermediate data (HaLoop reducer input cache), dynamic modifications to cached data (eg: genetic algorithm), and caching of output data (in HaLoop for fixed point evaluation).

## Apache Hadoop

Apache Hadoop[[18](#_ENREF_18)] MapReduce together with Hadoop distributed parallel file system(HDFS) [[19](#_ENREF_19)] provides a widely used open-source implementation of MapReduce. Hadoop supports data locality based scheduling and reduces the data transfer overheads by overlapping data communication with computations when reduce steps are involved. Hadoop performs duplicate executions of slower tasks and handles failures by rerunning the failed tasks using different workers. MapReduce frameworks like Hadoop tradeoff costs such as large startup overheads, task scheduling overheads and intermediate data persistence overheads for better scalability and reliability

## Twister4Azure

Twister4Azure is a distributed decentralized iterative MapReduce runtime for Windows Azure Cloud that was developed utilizing Azure cloud infrastructure services. Twister4Azure optimizes the iterative MapReduce computations by multi-level caching of loop invariant data, by performing cache aware scheduling, optimizing intermediate data transfers, optimizing data broadcasts and many other optimizations described in Gunarathne, et al[[3](#_ENREF_3)].

# Conclusions and Future Work

##### Acknowledgment

##### References

[1] E. Chan, M. Heimlich, A. Purkayastha et al., “Collective communication: theory, practice, and experience,” Concurrency and Computation: Practice and Experience, vol. 19, no. 13, pp. 1749-1783, 2007.

[2] "MPI Collective Communications," <http://www.ncsa.illinois.edu/UserInfo/Resources/Hardware/CommonDoc/MessPass/MPIColl.html>.

[3] T. Gunarathne, B. Zhang, T.-L. Wu et al., “Scalable parallel computing on clouds using Twister4Azure iterative MapReduce,” Future Generation Computer Systems, vol. 29, no. 4, pp. 1035-1048, 6//, 2013.

[4] J. Pjesivac-Grbovic, T. Angskun, G. Bosilca et al., "Performance analysis of MPI collective operations." p. 8 pp.

[5] J.Ekanayake, H.Li, B.Zhang et al., “Twister: A Runtime for iterative MapReduce,” in Proceedings of the First International Workshop on MapReduce and its Applications of ACM HPDC 2010 conference June 20-25, 2010, Chicago, Illinois, 2010.

[6] Y. Bu, B. Howe, M. Balazinska et al., “HaLoop: Efficient Iterative Data Processing on Large Clusters,” in The 36th International Conference on Very Large Data Bases, Singapore, 2010.

[7] R. W. Hockney, “The communication challenge for MPP: Intel Paragon and Meiko CS-2,” Parallel Computing, vol. 20, no. 3, pp. 389-398, 3//, 1994.

[8] J. Ekanayake, T. Gunarathne, and J. Qiu, “Cloud Technologies for Bioinformatics Applications,” Parallel and Distributed Systems, IEEE Transactions on, vol. 22, no. 6, pp. 998-1011, 2011.

[9] J. B. Kruskal, and M. Wish, Multidimensional Scaling: Sage Publications Inc., 1978.

[10] S.-H. Bae, J. Y. Choi, J. Qiu et al., “Dimension reduction and visualization of large high-dimensional data via interpolation,” in Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing, Chicago, Illinois, 2010, pp. 203-214.

[11] J. de Leeuw, “Convergence of the majorization method for multidimensional scaling,” Journal of Classification, vol. 5, pp. 163-180, 1988.

[12] S. Lloyd, “Least squares quantization in PCM,” Information Theory, IEEE Transactions on, vol. 28, no. 2, pp. 129-137, 1982.

[13] J. Dean, and S. Ghemawat, “MapReduce: simplified data processing on large clusters,” Commun. ACM, vol. 51, no. 1, pp. 107-113, 2008.

[14] Z. Bingjing, R. Yang, W. Tak-Lon et al., "Applying Twister to Scientific Applications." pp. 25-32.

[15] T. Gunarathne, B. Zhang, T.-L. Wu et al., “Portable Parallel Programming on Cloud and HPC: Scientific Applications of Twister4Azure,” in 2011 Fourth IEEE International Conference on Utility and Cloud Computing (UCC), Melbourne, Australia, 2011.

[16] Microsoft Daytona, Retrieved Feb 1, 2012, from : <http://research.microsoft.com/en-us/projects/daytona/>.

[17] M. Zaharia, M. Chowdhury, M. J. Franklin et al., “Spark: Cluster Computing with Working Sets,” in 2nd USENIX Workshop on Hot Topics in Cloud Computing (HotCloud '10), Boston, 2010.

[18] Apache Hadoop, Retrieved Mar. 20, 2010, from ASF: <http://hadoop.apache.org/core/>.

[19] "Hadoop Distributed File System HDFS," December, 2009; <http://hadoop.apache.org/hdfs/>.