Collective Communication Patterns for Iterative MapReduce

Thilina Gunarathne, Thomas Wiggins, Judy Qiu

October 11, 2013

SALSA Group, Pervasive Technology Institute, Indiana University
http://salsahpc.indiana.edu

CONTENTS

1 Introduction

Data-intensive iterative computations are a subset of distributed parallel computations suited for execution in cloud environments. Examples of such applications include dimensional scaling, clustering algorithms, machine learning algorithms, and expectation maximization applications. The growth in number as well as importance of such data-intensive iterative computations is driven partly by the need to process massive amounts of data, along with the emergence of data-intensive computational fields such as bioinformatics, chemical informatics and web mining. Examples of iterative data-intensive applications implemented with iterative MapReduce include PageRank[1], Multi-Dimensional Scaling[2, 3], K-means Clustering, Descendent query[1], 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 opportunities are highlighted by the development of frameworks such as Twister[4], HaLoop[1], Twister4Azure[3], Daytona[5], i-mapreduce, and Spark[6]. 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 (e.g. genetic algorithm), and caching of output data (HaLoop for fixed point evaluation).

When performing distributed computations, oftentimes the data needs to be shared and/or consolidated among different nodes of the computations. Collective communication primitives are the communication operations that involve a group of nodes simultaneously [7], rather than exchanging data between just a pair of nodes. Collective communication operations facilitate the optimized communication and coordination between groups of nodes of distributed computations, making it easier and more efficient to perform complex data communications inside the distributed parallel applications. Collective communication primitives are very popular in the HPC community and heavily used in the MPI type of HPC applications. A lot of research [7] has been done to optimize the performance of these collective communication operations, as they have a large impact on the performance of HPC applications. There exist many different implementations of collective communication primitives supporting numerous algorithms and topologies to suit the different environments and use cases.

In addition to the common characteristics of data-intensive iterative computations that we mentioned above, we noticed several common communication and computation patterns among some of the data-intensive iterative MapReduce computations. In this work, we present the applicability of collective communication operations to Iterative MapReduce without sacrificing the desirable properties of MapReduce programming model and execution framework, such as fault tolerance, scalability, familiar APIs and data model, etc. The addition of collective communication operations enriches the iterative MapReduce model by providing many advantages in performance and ease of use, such as providing efficient data communication operations optimized for the particular execution environment and use case, programming models that fit naturally with application patterns, and allowing users to avoid overhead by skipping unnecessary steps of the execution flow. These patterns can be implemented on any of the current Iterative MapReduce frameworks as well as on traditional Hadoop. In this paper we present the *Map-AllGather* and *Map-AllReduce* primitives and initial performance results for them.

The solutions presented focus on mapping All-to-All type collective communication operations, AllGather and AllReduce to the MapReduce model as Map-AllGather and Map-AllReduce patterns. *Map-AllGather* collects 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 most 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 K-means Clustering and up to 50% improvement with Multi-Dimensional Scaling in addition to the improved user friendliness. In some cases (ref. section 5.3), collective communication operations virtually eliminated the overhead of implementing reduce and/or merge functions.

2 BACKGROUND

2.1 COLLECTIVE COMMUNICATION

When performing distributed computations, most often the data needs to be shared and/or consolidated among the different nodes of the computations. Collective communication primitives are the communication operations that involve a group of nodes simultaneously[7], rather than exchanging data between just a pair of nodes. These powerful operations improve ease and efficiency in performing complex data communications. Some of the collective communication operations also provide synchronization capabilities to the applications as well. Collective communication operations are used heavily in MPI type of HPC applications. Much research[7] has gone into optimizing the performance of these collective communication operations, as they have a large impact on the performance of HPC applications. There exist numerous implementations of collective communication primitives supporting many algorithms and topologies to suit the different environments and use cases. The best collective implementation for a given scenario depends on many factors, including message size, number of workers, topology of the system, and the computational capabilities/capacity of the nodes[3]. Oftentimes collective communication implementations follow a poly-algorithm approach to automatically select the best algorithm and topology for the given scenario.

There are two main categories of collective communication primitives.

- Data redistribution operations
    - These operations can be used to distribute and share data across the worker processors. Examples of these include broadcast, scatter, gather, and allgather operations.
- Data consolidation operation
    - This type of operation can be used to collect and consolidate data contributions from different worker processors. 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
- Barrier

MapReduce model supports the All-to-One type communications through the Reduce step. MapReduce-MergeBroadcast model we introduce in section 3 further extends this support through the Merge step. Broadcast operation introduced in MapReduce-MergeBroadcast model 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 does the same for the iterations (or between the MapReduce jobs corresponding to iterations). 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 mode. For example, AllGather operation can be implemented as Reduce-Merge followed by Broadcast. However, these types of implementations would be inefficient and harder to use compared to dedicated optimized implementations of All-to-All operations.

2.2 MAPREDUCE

MapReduce consists of a programming model and an associated execution framework for distributed processing of very large data sets. MapReduce partitions the processing of very large input data into a set of independent tasks. Its programming model consists of *map(key$_1$ value$_1$)* function and *reduce(key$_2$, list<value$_2$>)* function. MapReduce programs written as Map and Reduce functions will be parallelized by the framework and executed in a distributed manner. The framework takes care of data partitioning, task scheduling, fault tolerance, intermediate data communication, and many other aspects. These features and the simplicity of the programming model allow 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. Their main goals include framework-managed fault tolerance, the ability to run on commodity hardware and process very large amounts of data, and horizontal scalability of compute resources. MapReduce frameworks like Hadoop trade off overhead costs such as large startup, task scheduling and intermediate data persistence for better scalability and reliability.

When running a computation, MapReduce frameworks first logically split the input data into partitions, where each partition would be processed by a single Map task. When a MapReduce computation has more map tasks than the Map slots available in the cluster, the tasks will be scheduled in waves. For example, a computation with 1000 map tasks executing in a cluster of 200 Map slots will execute as approximately 5 map task waves. Tasks in MapReduce frameworks are oftentimes dynamically scheduled, taking data locality into consideration. Map tasks read the data from the assigned logical data partition and process them as key-value pairs using the provided *map* function. The output key-value pairs of a map function are collected, partitioned, merged and transferred to the corresponding reduce tasks. Most of the MapReduce frameworks persist the Map output data in the Map nodes.

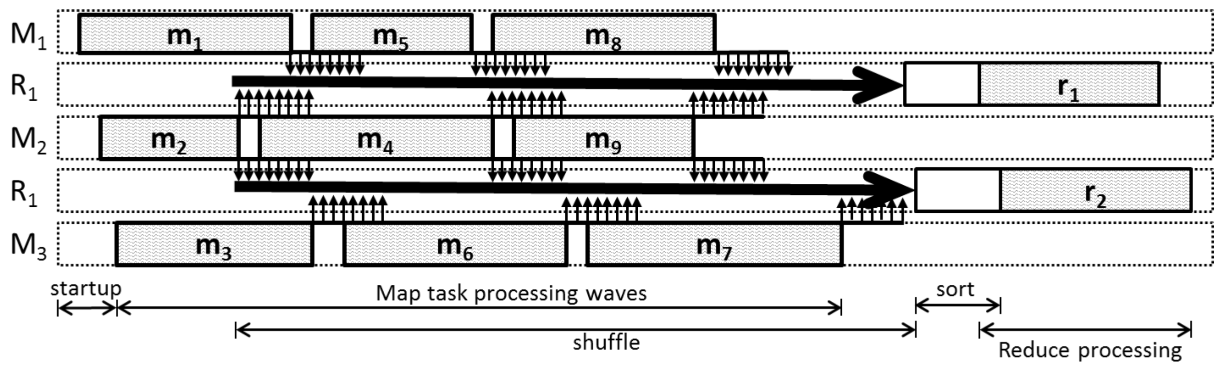

Figure 1. A sample MapReduce execution flow showing information transferred from Map to Reduce tasks with only those from M2 showing the transfer to reduce tasks r1 and r2 in reduce slots R1 and R2. Shown are three map tasks in each of three Map slots M_1, M_2, & M_3.

Reduce tasks fetch the data from the Map nodes and perform an external-merge sort on it. This starts as soon as the first map task completes the execution. Reduce task starts the reduce function processing once all the Map tasks are finished and after the intermediate data is shuffled and sorted.

	Map Task					Reduce Task			
Task Scheduling	**Data read**	**Map execution**	**Collect**	**Spill**	**Merge**	**Shuffle**	**Merge**	**Reduce Execution**	**Write output**

2.3 ITERATIVE MAPREDUCE

Many important data-intensive applications and algorithms can be implemented as iterative computation and communication steps, where computations inside an iteration are independent and synchronized at the end of each iteration through reduce and communication steps. Oftentimes each iteration is also amenable to parallelization. Many statistical applications fall into this category, including graph processing, clustering algorithms, data mining applications, machine learning algorithms, data visualization algorithms, and most of the expectation maximization algorithms. Their preeminence is a result of scientists relying on clustering, mining, and dimension reduction to interpret the data. Emergence of computational fields such as bioinformatics and machine learning has also contributed to increased interest in this class of applications.

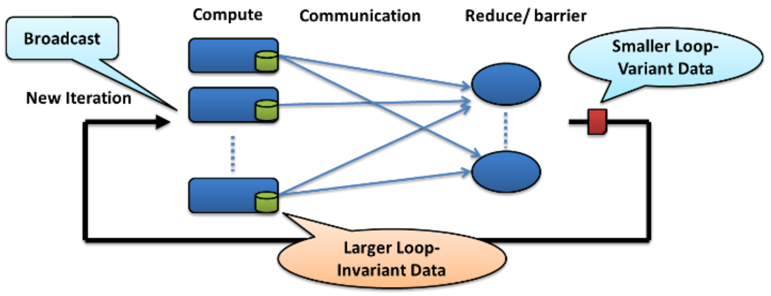

Figure 2. Structure of a typical data-intensive iterative application

As mentioned in the section above, there exists a significant amount of computation algorithms that rely on iterative computations with which we can easily specify each iterative step as a MapReduce computation. Typical data-intensive iterative computations follow the structure depicted in Figure 1.

We can identify two main types of data in these computations: the very large loop invariant input data and the smaller loop variant delta values. The loop invariant would be the set of input data points. Single iterations of these computations are easy to parallelize by processing the data points (or blocks of data points) independently in parallel while performing synchronization between the iterations through communication steps. In a K-means Clustering computation, the loop invariant input data would be the set input data vectors, while in a PageRank calculation it would be a representation of the link graph. The nature of these input data points gives rise to several optimization possibilities.

Delta values are the result of processing the input data in each iteration. Oftentimes these delta values are needed for the computation of the next iteration. In K-means Clustering computation, the loop variant delta values are the centroid values. In PageRank calculations the delta values conform to the page rank vector. Other general properties of the data-intensive iterative MapReduce calculations include relatively finer-grained tasks resulting in more prominent intermediate I/O overhead and a very large number of tasks due to multiple iterations giving more significance to the scheduling overhead.

Fault Tolerance for iterative MapReduce can be implemented either in the iteration level or in the task level. In the case of iteration-level fault tolerance, the checkpointing will happen on a per iteration basis and the frameworks can avoid checkpointing the individual task outputs. Due to the finer-grained nature of the tasks along with a high number of iterations, some users may opt for higher performance by selecting iteration-level fault tolerance. When this is used, the whole iteration would need to be re-executed in case of a task failure. Task-level fault tolerance is similar to the typical MapReduce fault tolerance, and the fault-recovery is performed by execution of failed Map or Reduce tasks.

3 MAPREDUCE-MERGEBROADCAST

Many iterative MapReduce frameworks can be specified as MapReduce-MergeBroadcast. In this section we introduce MapReduce-MergeBroadcast as a generic programming model for the data-intensive iterative computations.

3.1 API

MapReduce-MergeBroadcast programming model extends the *map* and *reduce* functions of traditional MapReduce to include the loop variant delta values as an input parameter. MapReduce-MergeBroadcast provides the loop variant data (*dynamicData*) 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)

This additional input can be used to provide the broadcast data to the Map and Reduce tasks. As we show in later sections, this additional input parameter can be used to provide the loop variant data distributed using other mechanisms to the map tasks. This extra input parameter can also be used to implement additional functionalities, such as performing map side joins.

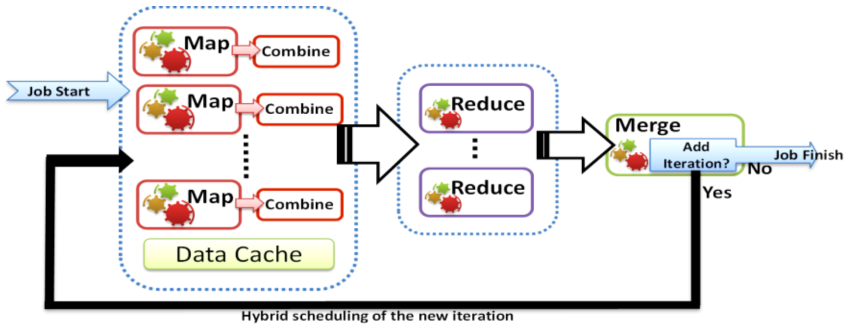

Figure 3. Twister4Azure MapReduce-MergeBroadcast compute/data flow

3.2 MERGE TASK

We define *Merge* as a new step to the MapReduce programming model to support iterative applications. It is a single task, or the convergence point, that executes after the *Reduce* step. It 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 tasks can be used to add a new iteration, finish the job, or schedule a new job. These decisions can be made based on the number of iterations or by comparison of the results from previous and current iterations, such as the k-value difference between iterations for K-means Clustering. Users can use the results of the current iteration and the broadcast data to make these decisions. Usually the output of the merge task needs to be broadcasted to tasks of the next iteration.

Merge Task receives all the *Reduce* outputs and the broadcast data for the current iteration as the inputs. There can only be one *merge* task for a MapReduce job. With *merge*, the overall flow of the iterative MapReduce computation flow would appear as follows:

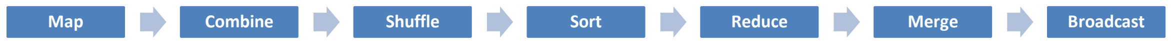

Figure 4. MapReduce-MergeBroadcast computation flow

The programming APIs of the Merge task can be where the “reduceOutputs” are the outputs of the reduce tasks and the “broadcastData” is the loop variant broadcast data for the current iteration.

Merge(list_of <key,list_of<value>> reduceOutputs, list_of <key,value> dynamicData)

3.3 BROADCAST

Broadcast operation broadcasts the loop variant data to all tasks in the iteration. In typical data-intensive iterative computations, the loop-variant data is orders of magnitude smaller than the loop-invariant data. In the MapReduce-MergeBroadcast model, the broadcast operation typically transmits the output data of the Merge tasks to the tasks of the next iteration. Broadcast operation of MapReduce-MergeBroadcast 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.

…MapReduce$_n$-> Merge$_n$-> Broadcast$_n$-> MapReduce$_{n+1}$-> Merge $_{n+1}$-> Broadcast$_{n+1}$-> MapReduce $_{n+2}$-> Merge...

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[8]. It's possible to share broadcast data between multiple tasks executing on a single node, as MapReduce computations typically have more than one map/reduce/merge worker per worker node.

3.4 MapReduce Merge-Broadcast Cost Model

There exist several models that are frequently used by the message passing community to model data communication performance[8]. We use the Hockney model [8, 9] for its simplicity. Hockney assumes the time to send a data set with n data items among two nodes is $\alpha+n\beta$, where α is the latency and β is the transmission time per data item (1/bandwidth). However it cannot model the network congestion.

Merge is a single task that receives the outputs of all the reduce tasks. The cost of this transfer would be $r\alpha + n_r\beta$, where n_r is the total number of reduce outputs and r is the number of reduce tasks. The execution time of the Merge task would be relatively small, as typically the merge would be performing a computationally trivial task such as aggregation or summarization. The output of the Merge task would need to be broadcasted to all the workers of the next iteration. A minimal spanning tree-based broadcast cost[7] can be modeled as follows, where n_v is the total number of merge outputs (broadcast data).

$$T_{MST-BCast} = (\alpha + \beta n_v)\log(p)$$

Based on these costs, the total cost of a MapReduce-MergeBroadcast can be approximated as shown in the proceeding diagram. The broadcast needs to be done only once per worker node, as the map tasks executing in a single worker node can share the broadcasted data among the tasks.

$$T_{MR-MB} = T_{MR} + r\alpha + n_r\beta + f(n_r) + (\alpha + \beta n_v)\log(p)$$

3.5 Current iterative MapReduce Frameworks and MapReduce-MergeBroadcast

Twister4Azure supports the MapReduce-MergeBroadcast 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 MapReduce computations broadcast the loop variant data products at the beginning of each iteration, effectively making the model Broadcast-MapReduce-Combine. This is semantically similar to the MapReduce-MergeBroadcast, as a broadcast at the beginning of iteration is similar to a broadcast at the end of the previous iteration.

HaLoop performs an additional MapReduce computation to do the fixed point evaluation on each iteration, effectively making this MapReduce computation equivalent to the Merge task. Data broadcast is achieved through a MapReduce computation to perform a join operation on the loop variant and loop invariant data.

All the above models can be generalized as Map->Reduce->Merge->Broadcast.

4 Collective Communications Primitives for Iterative MapReduce

There are several reasons why this is not guaranteed:

- Has to fit with the MapReduce data model
- Has to fit with the MapReduce computational model, which is more non-deterministic and has multiple waves, large overhead and inhomogeneous tasks
- Has to retain scalability and framework-managed fault tolerance of MapReduce
- Has to keep the programming model simple and easy to understand

4.1 Motivation

While implementing some of the iterative MapReduce applications, we noticed several common execution flow patterns. Another point of interest is that some of these applications have very trivial Reduce and Merge tasks when implemented using the MapReduce-MergeBroadcast model, while other applications needed extra effort to map to the MapReduce-MergeBroadcast model. This is owing to the execution patterns being slightly different than the iterative MapReduce pattern. In order to solve such issues, we introduce the Map-Collectives communications primitives, inspired by the MPI collective communications primitives, to the iterative MapReduce programming model.

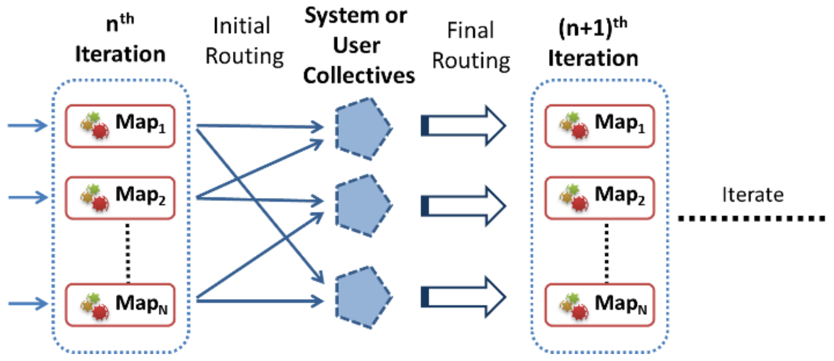

Figure 5. Collective Communication primitives

These primitives support higher-level communication patterns that occur frequently in data-intensive iterative applications, in addition to substituting certain steps of the computation. 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 three collective communication primitive implementations: *Map-AllGather*, *Map-AllReduce* and *Map-Scatter*. You can also identify MapReduce-MergeBroadcast as another collective communication primitive. Map-AllGather, Map-AllReduce and Map-Scatter also exist as special cases of the MapReduce-MergeBroadcast. These Map-Collective primitives provide many improvements over the traditional MapReduce-MergeBroadcast model. They are designed to maintain the same type of framework-managed excellent fault tolerance supported by the MapReduce frameworks.

In this work, our intention is to present a sufficiently optimal implementation for both the primitives and the environments in order to prove the performance efficiencies that can be gained through using even a modest implementation of these operations. We reserve locating the most optimized methods as a future work. Also we should note that finding most optimal implementations for cloud environments might end up being a moving target, as cloud environments evolve very rapidly and the cloud providers release new services and other features frequently.

4.1.1 Performance

These primitives allow us to skip or overlap certain steps of the typical iterative MapReduce computational flow. Having patterns that are more natural avoids unnecessary steps in traditional MR and iterative MR. Another advantage is the ability of the frameworks to optimize these operations transparently for the users, even affording the possibility of different optimizations (poly-algorithm) for different use cases and environments. For example, a communication algorithm that's best for smaller data sizes might not be the best for larger ones. In such cases, the collective communication operations can opt to implement multiple algorithm implementations to be used for different data sizes.

Finding the most optimal communication pattern implementations for a cloud environment is harder for the outsiders due to the black box nature of cloud environments. As such we don't have any information about the actual topology or the interconnects of public clouds. This presents an interesting opportunity for cloud providers to offer optimized implementations of these primitives as cloud infrastructure services, which can be used by the framework developers.

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, 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 in the Map-AllReduce primitive. The data reduction can be performed hierarchically using a reduction tree.

4.1.2 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 case when porting new applications to the iterative MapReduce model.

In addition, by using the Map-Collective operations, the developers can avoid manually implementing the logic of these operations (e.g. Reduce and Merge tasks) for each application and instead rely on optimized operations provided by the framework.

4.1.3 Scheduling with iterative primitives

Iterative primitives also give us the ability to propagate the scheduling information for the next iteration. These primitives can schedule the tasks of a new iteration or application through the primitives by taking advantage of their ability to deliver information about the new tasks. This can reduce scheduling overhead for the tasks of the new iteration. Twister4Azure successfully employs this strategy, together with the caching of task metadata, to schedule new iterations with minimal overhead.

4.1.4 *PROGRAMMING MODEL*

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

4.1.5 *IMPLEMENTATIONS*

Map-Collectives can be add-on improvements to MR frameworks. The simplest implementation would be applying these primitives using the current MapReduce model on the user level, then providing them as a library. This will achieve ease of use for the users and help with the performance. More optimized implementations can implement these primitives as part of the MapReduce framework, as well as providing the ability to optimize the data transfers based on environment and use case.

Table 1. Summary of patterns

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	K-means Clustering, PageRank
Map-AllGather	Map->AllGather Communication->AllGather Combine	H-Collectives, Twister4Azure	MDS-BCCalc
Map-AllReduce	Map->AllReduce (communication + computation)	H-Collectives, Twister4Azure	K-means Clustering, MDS-StressCalc
Map-ReduceScatter	Map->Scatter (communication + computation)	H-Collectives, Twister4Azure	PageRank, Belief Propagation

5 MAP-ALLGATHER COLLECTIVE

AllGather is an all-to-all collective communication operation that gathers data from all the workers and distributes the gathered data to them [7]. We can see the AllGather pattern in data-intensive iterative 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. An example would be a MapReduce computation that generates a matrix as the loop variant delta, 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 broadcast the result. An example of data-intensive iterative applications that have the AllGather pattern is MultiDimensionalScaling.

5.1 MODEL

We developed a Map-AllGather iterative MapReduce primitive similar to the MPI AllGather [7] collective communication primitive. Our intention was to support applications with communication patterns similar to those previously discussed in a more efficient manner.

5.1.1 Execution model

Map-AllGather primitive broadcasts the Map Task outputs to all computational nodes (all-to-all communication) of the current computation, then assembles them together in the recipient nodes. Each Map worker will deliver its result to all other workers of the computation once the Map task finishes. In the Matrix example mentioned above, each Map task broadcasts the rows of the matrix it generates. The resultant matrix would be assembled at each worker after the receipt of all the Map outputs.

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. This model substitutes the shuffle->sort->reduce->merge->broadcast steps of the MapReduce-MergeBroadcast with all-to-all broadcast and Allgather combine.

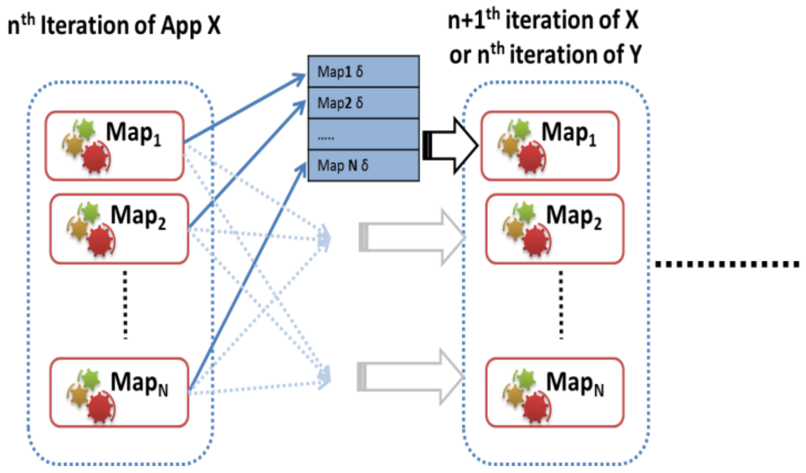

Figure 6. Map-AllGather Collective

5.1.2 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, sets of vectors (partial matrix), or single values. 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 values using the APIs and mechanisms suggested in section 2.2.1.

The final assembly of AllGather data can be performed by implementing a custom combiner or using a default 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 output key-value pairs, ordered by the key. This assembling function is executed in each worker node.

The default combiner should work for most of the use cases, as the combining of AllGather data is oftentimes a trivial process. It expects 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 abovementioned 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 thus are ignored in the default Map-AllGather combine step. AllGather operates with arrays of the basic output of Map tasks.

Users can use their Mapper implementation as it is with Map-AllGather primitive. All it requires is to specify the collective operation, and then the shuffle and reduce phases of MapReduce will be replaced by the AllGather communication and computations.

5.1.3 *Cost Model*

With an optimized implementation of AllGather, such as a bi-directional exchange-based implementation[7], we can estimate the cost of the AllGather shown in the following equation, where *m* is the number of map tasks.

$$T_{AllGather} = \log(m)\,\alpha + \frac{m-1}{m} n_v \beta$$

It's also possible to further reduce this cost by performing local aggregation in the Map worker nodes. In the case of AllGather, summation of size of all map output would be approximately equal to the loop variant data size of the next iteration ($n_m \approx n_v$). The variation of Map task completion times will also help to avoid network congestion in these implementations.

$$T_{AllGather} = \log(p)\,\alpha + \frac{p-1}{p} n_v \beta$$

Map-AllGather substitutes the Map output processing (collect, spill, merge), Reduce task (shuffle, merge, execute, write), Merge task (shuffle, execute) and broadcast overhead with a less costly AllGather operation. The MapReduce job startup overhead can also be significantly reduced by utilizing the information contained in the AllGather transfers to aid in scheduling the tasks of the next iteration. Hence Map-AllReduce per iteration overhead is significantly smaller than the traditional MapReduce job startup overhead.

5.2 Fault tolerance

When task level fault tolerance (typical map/reduce fault tolerance) is used and some of the AllGather data products are missing due to communication or other failures, it's possible for the workers to read map output data from the persistent storage (e.g. HDFS) to perform the AllGather computation. The fault tolerance model and the speculative execution model of MapReduce make it possible to have duplicate execution of tasks. Duplicate detection and removal can be performed before the final assembly of the data at the recipient nodes.

5.3 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 would be able to use the network more effectively than a larger broadcast originating from a single point.

Implementations of AllGather primitive can start broadcasting the map task result values as soon as the first map task is completed. In a typical MapReduce computation, the Map task execution times are inhomogeneous. 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 more significant when we have multiple waves of map tasks.

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

5.4 IMPLEMENTATIONS

In this section we present two implementations of the Map-AllGather primitive. These implementations are proof of concept to show the advantages achieved by using the Map-AllGather primitive. It's possible to further optimize them using more advanced 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[7]. One of the main advantages of these primitives is the ability for improvement without the need to change the user application implementations, leaving us open for optimization of these implementations in the future.

5.4.1 TWISTER4AZURE MAP-ALLGATHER

The Map-AllGather primitive is implemented in Twister4Azure using the Windows Communication Foundation (WCF)-based Azure TCP inter-role communication mechanism of Azure platform, with 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 don't wait for the other workers to complete and start transmitting the data as soon as a task is completed, taking advantage of the inhomogeneous finishing times of Map tasks to avoid communication scheduling and network congestion. More sophisticated implementations could be a tree-based algorithm 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.

5.4.2 H-COLLECTIVES MAP-ALLGATHER

H-Collectives Map-AllGather primitive is implemented using the Netty NIO library on top of Apache Hadoop. It 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. As shown in Fig. 7, tasks for the next iteration are already scheduled and waiting to start as soon as all the AllGather data is received, removing most of the MapReduce application startup and task scheduling overhead.

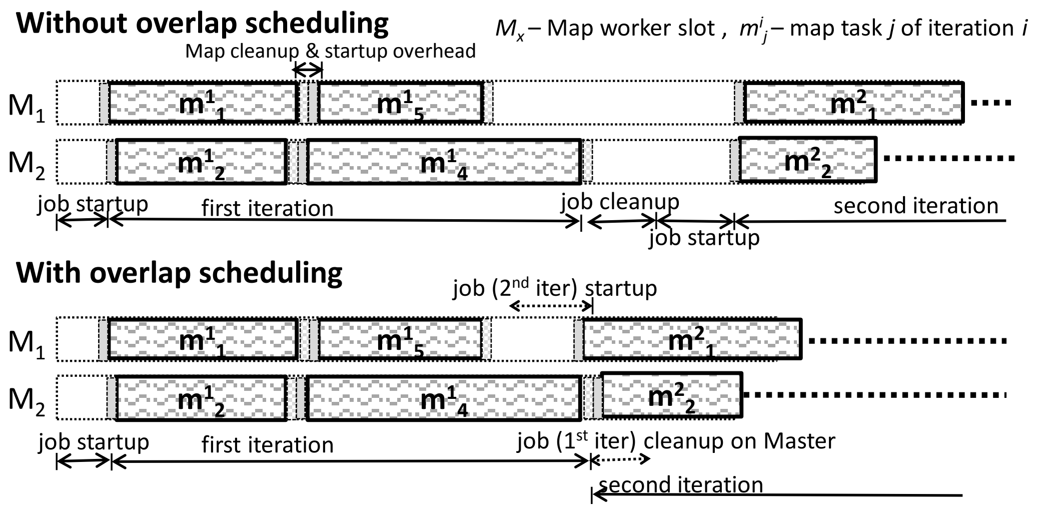

Figure 7. Scheduling map tasks m^k_i within map slots M_j. k labels iterations and i map tasks. The figure shows 4 map tasks and 2 map slots

6 MAP-ALLREDUCE COLLECTIVE

AllReduce is a collective pattern which combines a set of values emitted by all the workers based on a specified operation and makes the results available to all the workers[7]. 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 K-means Clustering, Multi-dimensional Scaling StressCalc computation and PageRank using out links matrix.

6.1 MODEL

We propose a Map-AllReduce iterative MapReduce primitive that will aggregate and reduce the results of the Map Tasks, similar to the MPI AllReduce [7] collective communication operation.

6.1.1 EXECUTION MODEL

The computation and communication pattern of a Map-AllReduce computation is Map phase followed by the AllReduce communication and computation (reduction). We noticed this model allows us to substitute the shuffle->sort->reduce->merge->broadcast steps of the MapReduce-MergeBroadcast with AllReduce communication in the communication layer. AllReduce phase can be implemented efficiently using algorithms such as bidirectional exchange (BDE) [7] or hierarchical tree-based reduction.

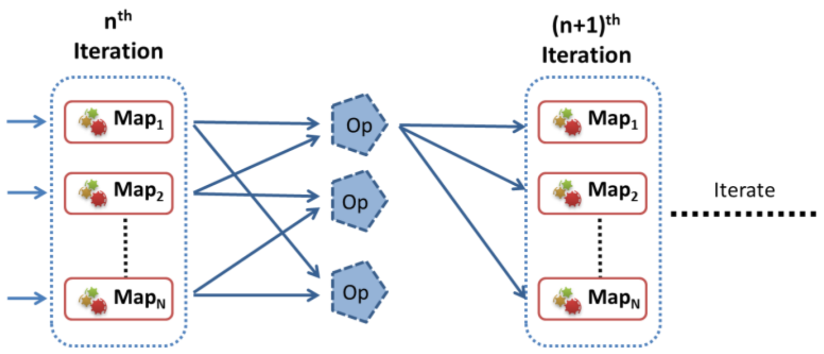

Figure 7. Map-AllReduce collective

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

6.1.2 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 are processed as separate data reduction operations. Output of the Map-AllReduce operation is a list of key value pairs, where each key is a map output key and corresponding value is the combined value of the map output values that were associated with that particular map output key. 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.

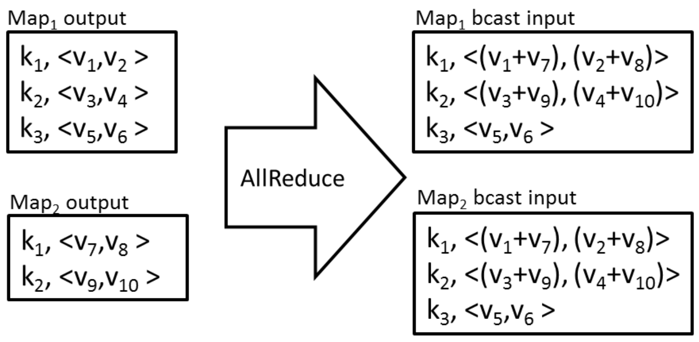

Figure 8. Example Map-AllReduce with Sum operation

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, adding an additional element (dimension) to the vector 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.

It is 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 be executed in each worker node after all the AllReduce data has been received by them.

list<Key, IOpRedValue> PostOpRedProcess(list<Key, IOpRedValue> opRedResult);

6.1.3 *Cost Model*

An optimized implementation of AllReduce, such as a bi-directional exchange-based implementation[7], will reduce the cost of the AllReduce component to:

$$T_{AllReduce} = \log(m)\,(\alpha + n_v\beta + f(n_v))$$

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. $\frac{m}{p}$ gives the average number of Map tasks per computation that executes in a given worker node. In the case of AllReduce, the average size of each map output would be approximately equal to the loop variant data size of the next iteration ($\frac{n_m}{m} \approx n_v$). The variation of Map task completion times will also help to avoid network congestion in these implementations.

$$T_{AR} = \log(p)\,(\alpha + n_v\beta + f(n_v)) + \frac{m}{p} f(n_v)$$

Map-AllReduce substitutes the Map output processing (collect, spill, merge), Reduce task (shuffle, merge, execute, write), Merge task (shuffle, execute) and broadcast overhead with a less costly AllReduce operation. The MapReduce job startup overhead can also be reduced by utilizing the information contained in the AllReduce transfers to aid in scheduling the tasks of the next iteration.

6.2 Fault Tolerance

When task level fault tolerance (typical map/reduce fault tolerance) is used and the AllReduce communication step fails for whatever reason, it's possible for the workers to read map output data from the persistent storage to perform the AllReduce computation.

The fault tolerance model and the speculative execution model of MapReduce make it possible to have duplicate execution of tasks. These 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 once the first Map task output is present. For example, if a mapper is re-executed or duplicate-executed, then it's possible for the AllReduce to combine duplicate values emitted from such mappers. 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 (e.g. HDFS) in case duplicate results are detected. Duplicate detection can be performed by maintaining a set of map IDs with each combined data product. It's possible for the frameworks to implement richer fault tolerance mechanisms, such as identifying the duplicated values in local areas of the reduction tree.

6.3 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 amount of intermediate data and optimizing the computation. The hierarchical reduction can be performed in as many levels as needed for 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.

6.4 IMPLEMENTATIONS

In this section we present two implementations of the Map-AllReduce primitive. These implementations are proofs of concept to show the advantages achieved by using the MapReduce primitive. It's possible to further optimize these implementations using more advanced 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 [2]. One of the main advantages of these primitives is the ability to improve the implementations without any need to change the user application implementations, leaving us the possibility of optimizing these implementations in the future.

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[8].

6.4.1 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. A single Azure worker node may run several Map workers and many more map tasks belonging to the computation. Hence the Twister4Azure Map-AllReduce implementation maintains a worker node-level cache of the AllReduce result values that would be available to any map task executing on that node. The iteration check happens in the destination nodes and can be specified as a custom function or as a maximum number of iterations.

6.4.2 H-COLLECTIVES MAP-ALLREDUCE

H-Collectives Map-AllReduce primitive is implemented on top of Apache Hadoop using node-level local aggregation and the Netty NIO library to broadcast the locally aggregated values to the other worker nodes of the computation. A single worker node may run several Map workers and many more map tasks belonging to the computation. Hence the Hadoop Map-AllReduce implementation maintains a node-level cache of the AllReduce result values. 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.

7 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.

7.1 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[10]. Dimensional scaling is used mainly in the visualization of high-dimensional data by mapping them onto two- or three-dimensional space. MDS has been used to visualize data in diverse domains, including but not limited to bioinformatics, geology, information sciences, and marketing. We use MDS to visualize dissimilarity distances for hundreds of thousands of DNA and protein sequences and identify relationships.

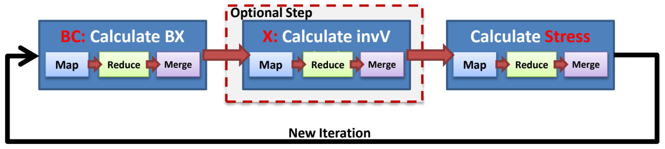

Figure 9. Twister4Azure Multi-Dimensional Scaling

In this paper, we use Scaling by MAjorizing a COmplicated Function (SMACOF)[11], an iterative majorization algorithm. The input for MDS is an N*N matrix of pairwise proximity values, where N is the number of data points in the high-dimensional space. The resultant lower dimensional mapping in D dimensions, called the X values, is an N*D matrix. In this paper, we implement the parallel SMACOF algorithm described by Bae et al[12]. This results in iterating a chain of three MapReduce jobs, as depicted in Figure 6. For the purposes of this paper, we perform an unweighted mapping that results in two MapReduce job steps per iteration, BCCalc and StressCalc. MDS is challenging for Twister4Azure due to its relatively finer-grained task sizes and multiple MapReduce applications per iteration.

Each BCCalc Map task generates a portion of the total X matrix. The reduce step of MDS BCCalc computation is a simple aggregation operation, where the reduction simply assembles the outputs of the Map tasks together in order. This X value matrix then needs to be broadcasted in order to be used in the StressCalc step of the current iterations as well as in the BCCalc step of the next iteration. The compute and communication flow of MDS BCCalc computation matches very well with the Map-AllGather primitive. Usage of the Map-AllGather primitive in MDS BCCalc computation eliminates the need for reduce, merge and the broadcasting steps for that particular computation.

7.1.1 MDS BCCALCULATION STEP COST

For the sake of simplicity, in this section we assume each MDS iteration contains only the BCCalculation step and analyze the cost of MDS computation.

Map compute cost can be approximated for large n to $d*n^2$, where n is the number of data points and d is the dimensionality of the lower dimensional space. Input data points in MDS are n dimensional ($n*n$ matrix). The total input data size for all the map tasks would be n^2 and the loop invariant data size would be $n*d$.

In MDS, the number of computations per *l* bytes of the input data is in the range of *k***l***d*, where *k* is a constant and *d* is typically 3. Hence MDS has larger data loading and memory overhead compared to the number of computations.

7.1.2 Twister4Azure MDS-AllGather

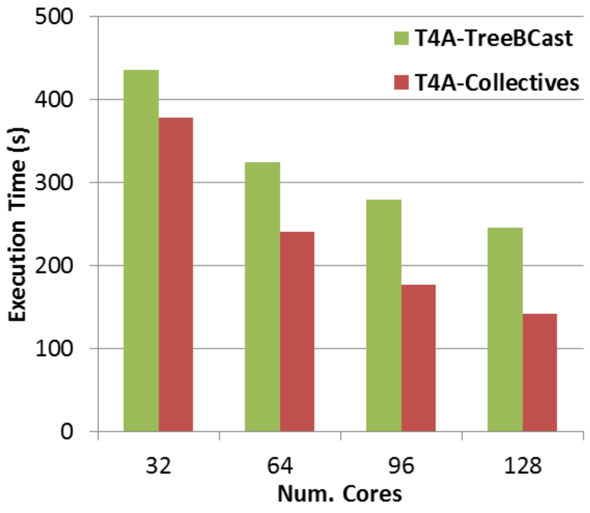

Figure 10. MDS application implemented using Twister4Azure. 20 iterations, 51200 data points (~5GB)

We implemented the Multi-Dimensional Scaling application for Twister4Azure using Map-AllGather primitive and MapReduce-MergeBroadcast with optimized broadcasting. Twister4Azure optimized broadcast version is an improvement over vanilla MapReduce since it uses an optimized tree-based algorithm to perform TCP broadcasts of in-memory data. Figure 10 shows the MDS strong scaling performance results comparing the Twister4Azure Map-AllGather-based implementation with the MapReduce-MergeBroadcast implementation. This test case scales a 51200*51200 matrix into a 51200*3 matrix. The test is performed on Windows Azure cloud using Azure extra-large instances [Table 1]. 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.

7.1.3 H-Collectives MDS-AllGather

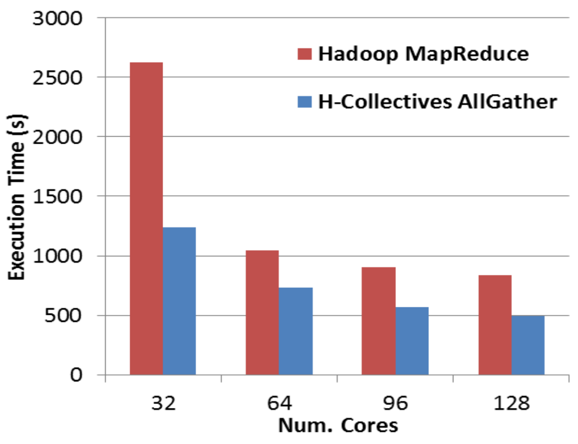

Figure 11. MDS application (BC calculation & Stress calculation in each iteration) implemented using Hadoop. 20 iterations, 51200 data points (~5GB)

We implemented the Multi-Dimensional Scaling application for Hadoop using vanilla MapReduce and H-Collectives Map-AllGather primitive. Vanilla MapReduce implementation uses the Hadoop DistributedCache to broadcast loop variant data to the Map tasks. Figure 11 shows the MDS strong scaling performance results comparing Map-AllGather based implementation with the MapReduce implementation. This test case scales a 51200*51200 matrix into a 51200*3 matrix. The test is performed on the IU Alamo cluster. The number of map tasks per computation is equal to the number of total cores of the computation. Here Map-AllGather-based implementation exceeds MapReduce performance by 30%, and 50% in the current test cases.

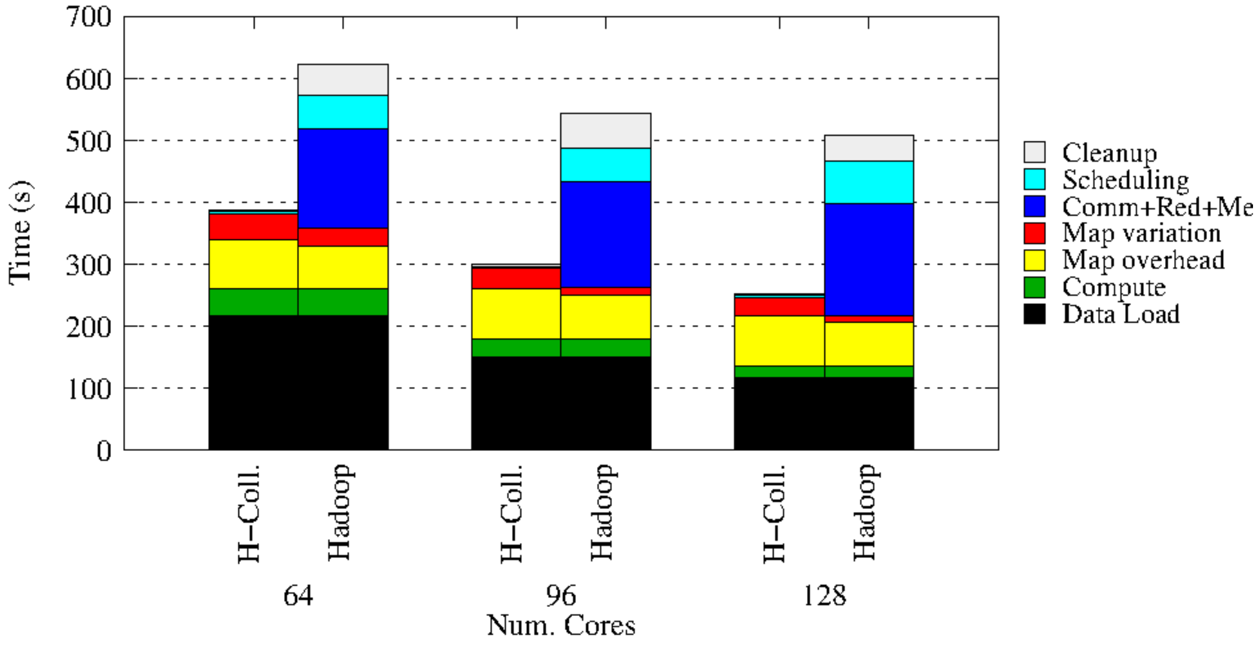

Figure 12. MDS Hadoop using only the BC Calculation MapReduce job per iteration to highlight the overhead. 20 iterations, 51200 data points

Figure 12 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. The 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 pure compute 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.

As we can see, the H-Collectives implementation gets rid of the communication, reduce, merge, task scheduling and job cleanup overhead of the vanilla MapReduce computation. However, we notice a slight increase of Map task overhead and Map variation in the case of 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 a few seconds between the scheduling.

This test case scales a 51200*51200 matrix into a 51200*3 matrix. The test is performed on the IU Alamo cluster.

7.1.3.1 Detailed analysis of overhead

In this section we perform detailed analysis of overhead of the Hadoop MDS BCalc calculation using a histogram of executing Map Tasks. In this test, we use only the BCCalc MapReduce job and remove the StressCalc step to show the overhead. MDS computations depicted in the graphs of 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.

These graphs plot the total number of executing Map tasks at a given moment of the computation. The number of executing Map tasks approximately represent the amount of useful work done in the cluster at that given moment. The resultant graphs comprised of blue bars represent an iteration of the computation. The width of each blue bar illustrates 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 overhead of the computation.

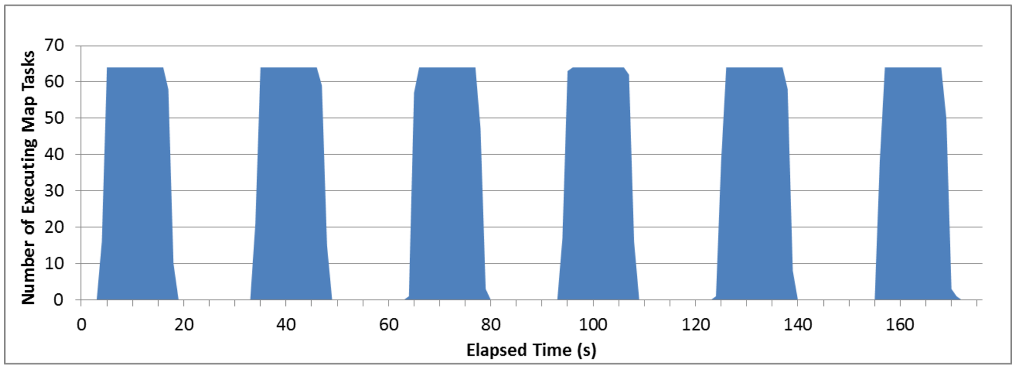

Figure 13. Hadoop MapReduce MDS-BCCalc histogram

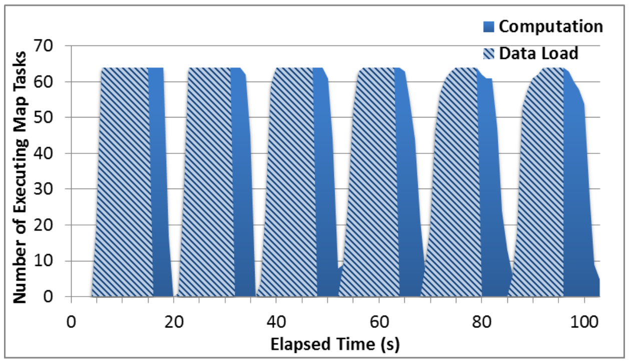

Figure 14. H-Collectives AllGather MDS-BCCalc histogram

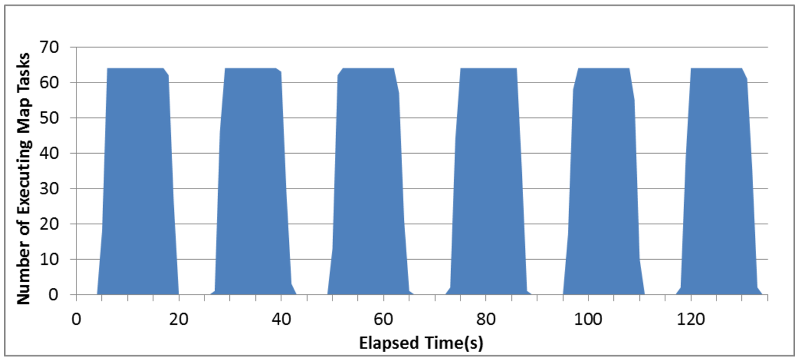

Figure 15. H-Collectives AllGather MDS-BCCalc histogram without speculative scheduling

Figure 14 presents MDS using H-Collectives AllGather implementation. Hadoop driver program performs speculative (overlap) scheduling of iterations by scheduling the tasks for the next iteration while the previous iteration is still executing. The scheduled tasks wait for the AllGather data to start the actual execution. Blue bars represent the map task time of each iteration, while the striped section on each blue bar represents the data loading time (time it takes to read input data from HDFS). Overhead of this computation includes AllGather communication and task scheduling. A MapReduce job for the next iteration is scheduled while the previous iteration is executing. Overhead between the iterations virtually disappears with the use of AllGather primitive.

Figure 15 presents MDS using H-Collectives AllGather implementation without the speculative (overlap) scheduling. In this graph, the MapReduce job for the next iteration is scheduled after the previous iteration is finished. Compared to Figure 14, this figure shows the gains that can be achieved by enabling optimized task scheduling with help from the information from collective communication operations. Hadoop MapReduce implementation can't overlap the iterations, as we require adding the loop variant data (only available after the previous iteration is finished) to the Hadoop DistributedCache when scheduling the job.

7.1.4 Twister4Azure vs Hadoop

Twister4Azure is already optimized for iterative MapReduce[13] and contains very low scheduling, data loading and data communication overhead compared to Hadoop. Hence the overhead reduction we achieve by using collective communication is relatively less in Twister4Azure compared to Hadoop.

Another major component of Hadoop MDS Map task cost is due to the data loading, seen in Figure 14. Twister4Azure avoids this by using data caching and cache aware scheduling.

7.2 K-MEANS CLUSTERING USING MAP-ALLREDUCE

The K-means Clustering[14] algorithm has been widely used in many scientific and industrial application areas due to its simplicity and applicability to large datasets. We are currently working on a scientific project that requires clustering of several terabytes of data using K-means Clustering and millions of centroids.

K-means clustering is often implemented using an iterative refinement technique in which the algorithm iterates until the difference between cluster centers in subsequent iterations, i.e. the error, falls below a predetermined threshold. 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. Centroid data is broadcasted at the beginning of the iteration. Intermediate data communication is relatively costly in K-means Clustering, as each Map Task outputs data equivalent to the size of the centroids for an 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) need to be combined independently, and the resultant key-value pairs (new centroids) are distributed to all the Map tasks of the next iteration.

7.2.1 K-MEANS CLUSTERING COST

K-means centroid assignment step (Map tasks) cost can be approximated for large n to $n*c*d$, where n is the number of data points, d is the dimensionality of the data and c is the number of centroids. The total input data size for all the map tasks would be $n*d$ and the loop invariant data size would be $c*d$.

K-means Clustering approximate computation and communication cost when using the AllReduce primitive is seen in the following diagram. The cost of the computation component of AllReduce is $k*c*d$, where k is the number of data sets reduced at that particular step.

In K-means Clustering, the number of computations per l bytes of the input data is in the range of $k*l*c$, where k is a constant and c is the number of centroids. Hence for non-trivial number of centroids, K-means Clustering has relatively smaller data loading and memory overhead vs. the number of computations compared to the MDS application discussed above.

The compute cost difference between K-means Clustering MapReduce-MergeBroadcast and Map-AllReduce implementations is equal or slightly in favor of the MapReduce due to the hierarchical reduction performed in the AllReduce implementation. However, typically the compute cost of the reduction is almost negligible. All the other overhead including, the startup, disk and communication overhead, favors the AllReduce-based implementation.

7.2.2 TWISTER4AZURE K-MEANS CLUSTERING-ALLREDUCE

We implemented the K-means Clustering application for Twister4Azure using the Map-AllReduce primitive, vanilla MapReduce-MergeBroadcast and optimized broadcasting. Twister4Azure tree broadcast version is also an improvement over vanilla MapReduce since it uses an optimized tree-based algorithm to perform TCP broadcasts of in-memory data. The vanilla MapReduce implementation and optimized broadcast implementation uses in-map combiners to perform local aggregation of the values to minimize the size of map-to-reduce data transfers. Figure 10's left diagram shows the K-means Clustering weak scaling performance results, where we scale the computations while keeping the workload per core constant. Figure 16's right side presents the K-means Clustering strong scaling performance, where we scaled the number of cores while keeping the data size constant. Results compared different implementations of Twister4Azure.

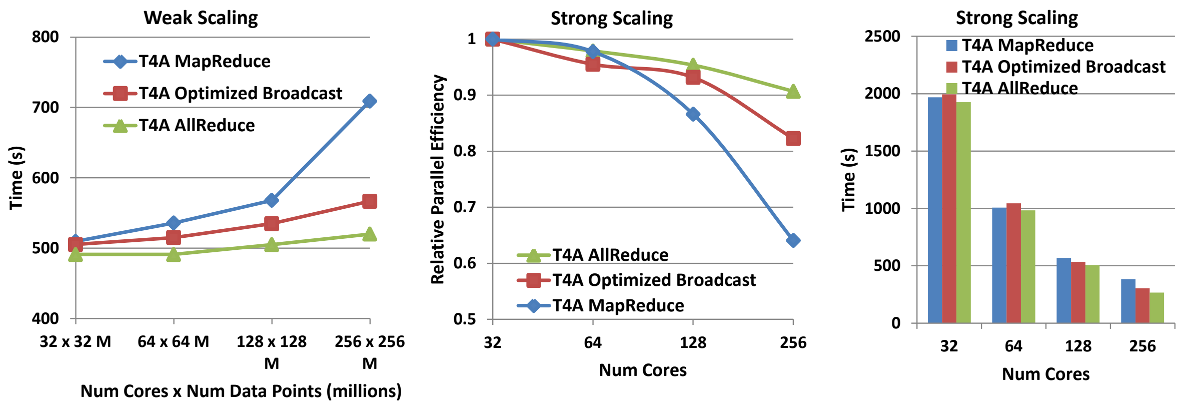

Figure 16. Twister4Azure K-means Clustering comparison with Map-AllReduce. 500 Centroids (clusters), 20 Dimensions, 10 iterations. Left: Weak scaling 32 to 256 Million data points. Right: 128 Million data points. Parallel efficiency relative to the 32

7.2.3 H-Collectives K-means Clustering-AllReduce

We implemented the K-means Clustering application for Hadoop using the Map-AllReduce primitive and 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. Figure 17 illustrates the Hadoop K-means Clustering weak scaling performance results, in which we scale the computations while keeping the workload per core constant. Figure 18 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 uses more map task waves optimizing the intermediate communication, resulting in relatively smaller overhead for the computation (HDFS replication factor of 6 increasing data locality).

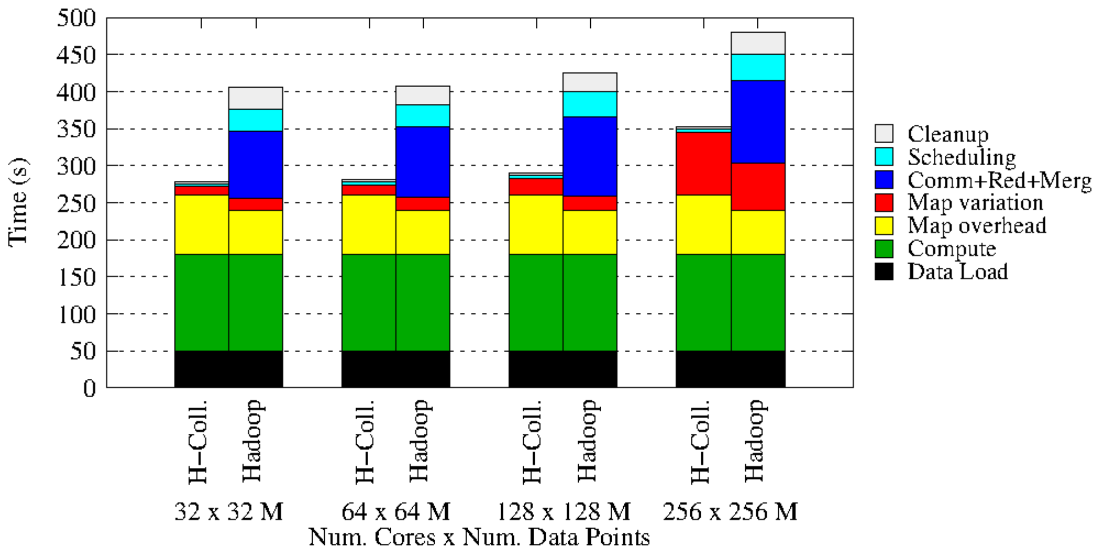

Figure 17. Hadoop K-means Clustering comparison with H-Collectives Map-AllReduce Weak scaling. 500 Centroids (clusters), 20 Dimensions, 10 iterations

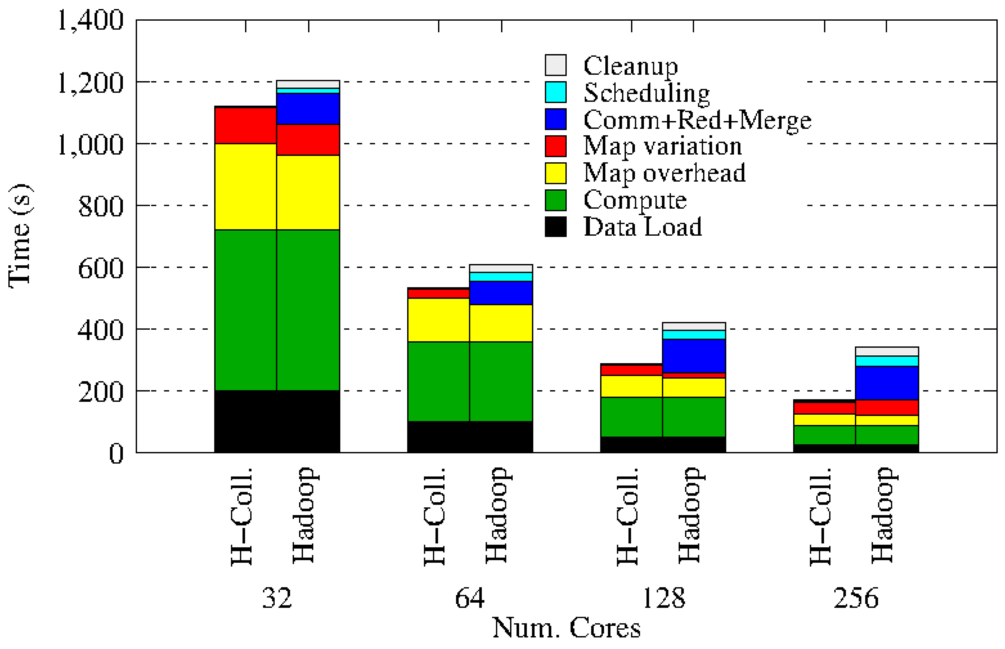

Figure 18. Hadoop K-means Clustering comparison with H-Collectives Map-AllReduce Strong scaling. 500 Centroids (clusters), 20 Dimensions, 10 iterations

The 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, execution, merge and broadcast. "Compute" and "data load" times are calculated using the average pure compute 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.

As we can see, the H-Collectives implementation gets rid of the communication, reduce, merge, task scheduling and job cleanup overhead of the vanilla MapReduce computation. Again, there is a slight increase of Map task overhead and Map variation in the case of H-Collectives Map-AllReduce-based implementation, possibly due to the rapid scheduling of Map tasks across successive iterations in H-Collectives. In vanilla MapReduce the Map tasks have a few seconds between scheduling.

7.2.4 Twister4Azure vs. Hadoop

KMeans performs more computation per data load than MDS, and the compute time dominates the run time. The pure compute time of C#.net based application in Azure is much slower than the java-based application executing in a Linux environment. Twister4Azure is still able to avoid lots of overhead and improves the performance of the computations, but the significantly lower compute time results in lower running times for the Hadoop applications.

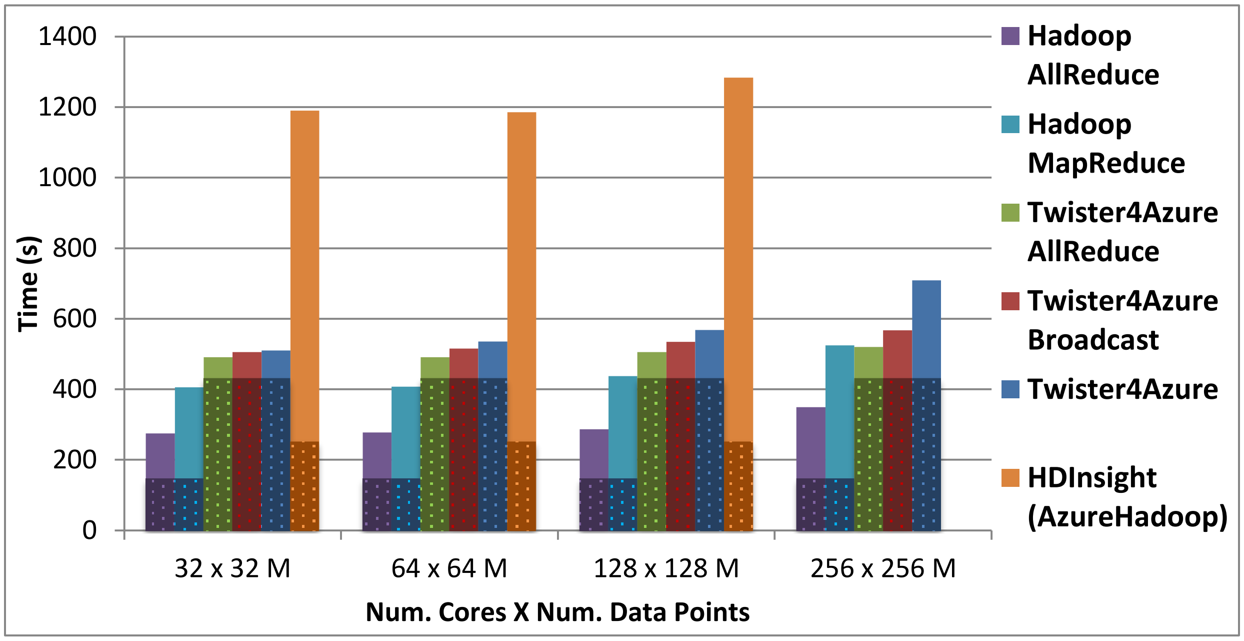

Figure 19. HDInsight KMeans Clustering compared with Twister4Azure and Hadoop

HDInsight offers hosted Hadoop as a service on the Windows Azure cloud. Figure 19 presents the K-means Clustering performance on the HDInsight service using Windows Azure large instances. We executed the same Hadoop MapReduce-based K-means Clustering implementation used in section 7.2.3 on HDInsight. HDInsight currently limits the number of cores to 170, which doesn't allow us to perform the 256 core test on it.

Input data for the HDInsight computation was stored in Azure Blob Storage and accessed through ASV (Azure Storage Vault), which provides a HDFS file system interface for the Azure blob storage. Input data for the Twister4Azure computation was also stored in Azure blob storage and cached in memory using the Twister4Azure caching feature.

The darker areas of the bars represent the approximated compute-only time for the computation based on the average map task compute-only time. The rest of the area in each bar represents the overhead, which would be the time taken for task scheduling, data loading, shuffle, sort, reduce, merge and broadcast. The overhead is particularly high for HDInsight due to the data download from the Azure Blob storage on each iteration. The variation of the time to download data from the Azure Blob storage adds significant variation to the map task execution times, affecting the whole iteration execution time.

Twister4Azure computation is significantly faster than HDInsight due to the data caching and other improvements, such as hybrid TCP-based data shuffling, cache aware scheduling, etc. , even though the compute only time (darker areas) is much higher in Twister4Azure (C# vs Java) than in HDInsight.

8 BACKGROUND & RELATED WORK

8.1.1 TWISTER4AZURE

Twister4Azure[13] is a distributed decentralized iterative MapReduce runtime for Windows Azure Cloud that was developed utilizing Azure cloud infrastructure services. Twister4Azure extends the familiar, easy-to-use MapReduce programming model with iterative extensions and novel communication primitives, enabling a wide array of large-scale iterative and non-iterative data analysis, as well as scientific applications to utilize Azure platform easily and efficiently in a fault-tolerant manner. Twister4Azure utilizes the eventually-consistent, high-latency Azure cloud services effectively to deliver performance comparable to (non-iterative) and outperforming (for iterative computing) traditional MapReduce runtimes. Twister4Azure has minimal management and maintenance overhead and provides users with the capability to dynamically scale the amount of compute resources up or down.

8.1.2 TWISTER

The Twister[15] iterative MapReduce framework is an expansion of the traditional MapReduce programming model, which supports traditional as well as iterative MapReduce data-intensive computations. Twister supports MapReduce in the manner of "configure once, run many times." Twister configures and loads static data into Map or Reduce tasks during the configuration stage and then reuses the loaded data through the iterations. In each iteration, the data is first mapped in the compute nodes, reduced, then combined back to the driver node (control node). Using direct TCP as well as messaging middleware, Twister supports direct intermediate data communication across the workers without persisting the intermediate data products to the disks. With these features, Twister supports iterative MapReduce computations efficiently when compared to other traditional MapReduce runtimes such as Hadoop[16]. Fault detection and recovery are supported between the iterations. In this paper, we use the java implementation of Twister and identify it as Java HPC Twister.

Java HPC Twister uses a master driver node for management and control of the computations. The Map and Reduce tasks are implemented as worker threads managed by daemon processes on each worker node. Daemons communicate with the driver node and each other through messages. For command, communication and data transfers, Twister uses a Publish/Subscribe messaging middleware system. ActiveMQ[17] is used for the current experiments. Twister performs optimized broadcasting operations by utilizing chain method[5] and uses minimum spanning tree method[18] for efficiently sending Map data from the driver node to the daemon nodes. Twister supports data distribution and management through a set of scripts as well as through the HDFS[19].

8.1.3 MICROSOFT DAYTONA

Microsoft Daytona[5] is a recently announced iterative MapReduce runtime developed by Microsoft Research for Microsoft Azure Cloud Platform that builds on some of the ideas of the earlier Twister system. Daytona utilizes Azure Blob Storage for storing intermediate data, final output data enabling data backup and easier failure recovery. Daytona supports caching of static data between iterations. It combines the output data of the reducers to form the output of each iteration. Once the application has finished, the output can be retrieved from Azure Blob storage or continually processed by using other applications. In addition to the above features similar to Twister4Azure, Daytona also provides automatic environment deployment and data splitting for MapReduce computations. It also claims to support a variety of data broadcast patterns between the iterations. However, as opposed to Twister4Azure, Daytona uses a single master node-based controller to drive and manage the computation. This centralized controller substitutes for the 'Merge' step of Twister4Azure, but makes Daytona prone to single points of failure.

Currently Excel DataScope is presented as an application of Daytona. Users can upload data in their Excel spreadsheet to the DataScope service or select a dataset already in the cloud, then choose an analysis model from our Excel DataScope research ribbon to run against the selected data. The results can be returned to the Excel client or remain in the cloud for further processing and visualization. Daytona is available as a "Community Technology Preview" for academic and non-commercial use.

8.1.4 HALOOP

HaLoop[1] extends Apache Hadoop to support iterative applications and supports caching of loop-invariant data as well as loop-aware scheduling. Similar to Java HPC Twister and Twister4Azure, HaLoop also provides a new programming model which includes several APIs that can be used for expressing iteration-related operations in the application code.

However, HaLoop doesn't have an explicit Combine operation to get the output to the master node. It also uses a separate MapReduce job to do the calculation (called Fixpoint evaluation) for terminal condition evaluation. HaLoop provides a high-level query language, which is not available in either Java HPC Twister or Twister4Azure.

HaLoop performs loop-aware task scheduling to accelerate iterative MapReduce executions. It enables data reuse across iterations by physically co-locating tasks that process the same data in different iterations. In HaLoop, the first iteration is scheduled similar to traditional Hadoop. After that the master node remembers the association between data and node while the scheduler tries to retain previous data-node associations in the following iterations. If the associations can no longer hold due to the load, the master node will associate the data with another node. HaLoop also provides several mechanisms of on-disk data caching, such as reducer input cache and mapper input cache. In addition to these two, there is another cache called reducer output cache, which is specially designed to support Fixpoint Evaluations. HaLoop can also cache intermediate data (reducer input/output cache) generated by the first iteration, iMapReduce. These caches are in addition to Hadoop Distributed Cache which holds data associated with other nodes.

9 Acknowledgements

We would like to thank colleagues and members of the Digital Science Center at Indiana University for helpful discussions and contributions to the Twister4Azure project and the present work. We gratefully acknowledge support from Microsoft for Azure Cloud Academic Resources Allocation, which were critical for our experiments.

10 References

[1] 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.
[2] Z. Bingjing, R. Yang, W. Tak-Lon *et al.*, "Applying Twister to Scientific Applications." pp. 25-32.
[3] 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.
[4] 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.
[5] *Microsoft Daytona*, Retrieved Feb 1, 2012, from : http://research.microsoft.com/en-us/projects/daytona/.
[6] 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.
[7] 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.
[8] J. Pjesivac-Grbovic, T. Angskun, G. Bosilca *et al.*, "Performance analysis of MPI collective operations." p. 8 pp.
[9] 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.
[10] J. B. Kruskal, and M. Wish, *Multidimensional Scaling*: Sage Publications Inc., 1978.
[11] J. de Leeuw, “Convergence of the majorization method for multidimensional scaling,” *Journal of Classification,* vol. 5, pp. 163-180, 1988.
[12] 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.
[13] 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.
[14] S. Lloyd, “Least squares quantization in PCM,” *Information Theory, IEEE Transactions on,* vol. 28, no. 2, pp. 129-137, 1982.
[15] 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.
[16] Bingjing Zhang, Yang Ruan, Tak-Lon Wu *et al.*, “Applying Twister to Scientific Applications,” in CloudCom 2010, IUPUI Conference Center Indianapolis, 2010.
[17] Apache, "ActiveMQ," http://activemq.apache.org/, 2009].
[18] J. Lin, and C. Dyer, “Data-Intensive Text Processing with MapReduce,” *Synthesis Lectures on Human Language Technologies,* vol. 3, no. 1, pp. 1-177, 2010/01/01, 2010.
[19] "Hadoop Distributed File System HDFS," December, 2009; http://hadoop.apache.org/hdfs/.