Towards a Collective Layer in the Big Data Stack

Thilina Gunarathne
Department of Computer Science
Indiana University, Bloomington
tgunarat@indiana.edu

Judy Qiu
Department of Computer Science
Indiana University, Bloomington
xqiu@indiana.edu

Dennis Gannon
Microsoft Research,
Redmond,WA
dennis.gannon@microsoft.com

***Abstract*—We generalize MapReduce, Iterative MapReduce and data intensive MPI runtime as a layered Map-Collective architecture with Map-AllGather, Map-AllReduce, MapReduceMergeBroadcast and Map-ReduceScatter patterns as the initial focus. Map-collectives improve the performance and efficiency of the computations while at the same time facilitating ease of use for the users. These collective primitives can be applied to multiple runtimes and we propose building high performance robust implementations that cross cluster and cloud systems. Here we present results for two collectives shared between Hadoop (where we term our extension H-Collectives) on clusters and the Twister4Azure Iterative MapReduce for the Azure Cloud. Our prototype implementations of Map-AllGather and Map-AllReduce primitives achieved up to 33% performance improvement for K-means Clustering and up to 50% improvement for Multi-Dimensional Scaling, while also improving the user friendliness. In some cases, use of Map-collectives virtually eliminated almost all the overheads of the computations.**

Keywords: MapReduce, Twister, Collectives, Cloud, HPC, Performance, K-means, MDS

I. INTRODUCTION

During the last decade three largely industry-driven disruptive trends have altered the landscape of scalable parallel computing, which has long been dominated by HPC applications. These disruptions are the emergence of data intensive computing (aka big data), commodity cluster-based execution & storage frameworks such as MapReduce, and the utility computing model introduced by Cloud computing. Oftentimes MapReduce is used to process the "Big Data" in cloud or cluster environments. Although these disruptions have advanced remarkably, we argue that we can further benefit these technologies by generalizing MapReduce and integrating it with HPC technologies. This splits MapReduce into a Map and a Collective communication phase that generalizes the Reduce concept. We present a set of Map-Collective communication primitives that improve the efficiency and usability of large-scale parallel data intensive computations.

When performing distributed computations, data often 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, 2]. Collective communication primitives are very popular in the HPC community and used heavily in the MPI type of HPC applications. There has been much research [1] to optimize the performance of these collective communication operations, as they have a significant impact on the performance of HPC applications.

Our work highlights several Map-Collective communication primitives to support and optimize common computation and communication patterns in both MapReduce and iterative MapReduce computations. We present the applicability of Map-Collective operations to enhance (Iterative) MapReduce without sacrificing desirable MapReduce properties such as fault tolerance, scalability, familiar APIs and data model. The addition of Map-Collectives enriches the MapReduce model by providing many performance and ease of use advantages. These include providing efficient data communication operations optimized for particular execution environments & use cases, enabling programming models that fit naturally with application patterns and allowing users to avoid overhead by skipping unnecessary steps of the execution flow. Map-Collective operations substitute multiple successive steps of an iterative MapReduce computation with a single powerful collective communication operation.

We present these patterns as high level constructs that can be adopted by any MapReduce or iterative MapReduce runtime. We also offer proof-of-concept implementations of the primitives on Hadoop and Twister4Azure and envision a future where all the MapReduce and iterative MapReduce runtimes support a common set of Map-Collective primitives.

This paper focuses on mapping the All-to-All communication type of collective operations, namely 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 MapReduceMergeBroadcast as an important collective in all (iterative) MapReduce frameworks.

II. MAPREDUCE-MERGEBROADCAST (MR-MB)

We introduce MapReduce-MergeBroadcast[1] abstraction, called MR-MB from here onwards, as a generic abstraction to represent data-intensive iterative MapReduce applications. Programming models of most of the current iterative MapReduce frameworks can be specified as MR-MB.

A. API

The MR-MB programming model extends the *map* and *reduce* functions of traditional MapReduce to include the loop variant data 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)

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-meansClustering, PageRank,
Map-AllGather	Map→AllGather Communication→AllGather Combine	H-Collectives, Twister4Azure	MDS-BCCalc (matrix X matrix), PageRank (matrix X vector)
Map-AllReduce	Map→AllReduce (communication & computation)	H-Collectives, Twister4Azure	K-meansClustering, MDS-StressCalc

B. *Merge Task*

Merge[2] 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 API of the Merge task.

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

C. *Broadcast*

The broadcast operation transmits 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. For MR-MB, this 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 (e.g., …MR_n→ $Merge_n$→ $Broadcast_n$→ MR_{n+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[3]. It's possible to share broadcast data between multiple Map and/or Reduce tasks executing on the same node.

D. *Current iterative MapReduce Frameworks and MR-MB*

Twister4Azure[2] supports MR-MB natively. Twister[4] is a MapReduce-Combine model, where the Combine step is similar to the Merge step of MR-MB. Twister[4] 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 MR-MB. HaLoop[5] 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.

III. 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 other applications needed extra effort to implement using the MR-MB model owing to the execution patterns being slightly different than the MR-MB pattern. In order to solve such issues, we introduce Map-Collective primitives to the iterative MapReduce programming model, inspired by the MPI collective communications primitives[6].

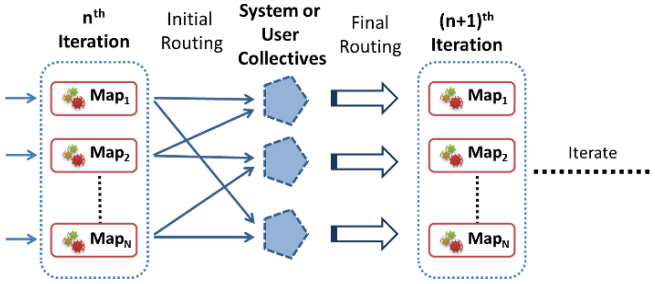

Figure 1. Map-Collective primitives

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 Map-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.

This paper proposes two Map-Collective primitives: Map-AllGather and Map-AllReduce. We can classify MR-MB as another collective communication primitive as well.

A. *Requirements*

Map-Collective architecture should fit with the MapReduce data model and the MapReduce computational model, which support multiple Map task waves, significant execution variations and inhomogeneous tasks. Also the primitives should retain scalability while keeping the programming model simple and easy to understand. These primitives should maintain the same type of framework-managed excellent fault tolerance supported by MapReduce.

B. *Advantages*

1) *Performance improvement*

Introduction of Map-Collective primitives provides 3 types of performance improvements to the iterative MapReduce applications. Map-Collectives reduce the overhead of the computations by skipping or overlapping certain steps

(e.g. shuffle, reduce, merge) of the iterative MapReduce computational flow. Map-Collective patterns also fit more naturally with the application patterns, avoiding the overheads of unnecessary trivial steps.

Map-Collectives enable the applications to send and receive the data across iterations much faster, since the execution of Map-Collectives can be started as soon as the first Map results are produced. This helps to mitigate the bad effects of task heterogeneity by not having to wait for global barriers of multiple processing steps. Task heterogeneity actually helps in this case by reducing the congestion of the data transfers and overlaps communication with computation.

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 example, a communication algorithm that's best for smaller data sizes may not be the best for larger ones. In such cases, the Map-Collective operations can opt to have multiple algorithm implementations to be used for different data sizes.

Map-Collectives also make it possible to perform some of the computations in the data transfer layer, like the hierarchical reduction in Map-AllReduce primitive.

2) Ease of use

Map-Collective operations present patterns and APIs that fit more naturally with real world applications. This simplifies the porting of new applications to the iterative MapReduce model. In addition, the developers do not have to implement, test and optimize certain steps of MR-MB, such as Reduce and Merge tasks, and can avoid MapReduce driver side hacks to broadcast the data.

3) Scheduling with iterative primitives

In addition to providing synchronization between the iterations, Map-Collective primitives also give us the ability to propagate the scheduling information for the next iteration to the worker nodes along with the collective communication data. This allows the frameworks to synchronize and schedule the tasks of a new iteration or application with minimal overhead.

For example, as mentioned in section VI, Twister4Azure successfully employs this strategy to schedule new iterations with minimal overhead, while H-Collectives use this strategy to perform speculative scheduling of tasks.

C. Programming model

Map-Collective 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 them. This also makes it easy for developers who are already familiar with MapReduce programming to use Map-Collectives. For example, a K-means Clustering MapReduce implementation with Map, Reduce and Merge tasks can be used with Map-AllReduce or vice versa without making any changes to the Map, Reduce or Merge function implementations.

D. Implementation considerations

Map-Collectives can be add-on improvements to MapReduce frameworks. The simplest implementation would be to implement the Map-Collectives communication and computation models as a user level library using the current MapReduce APIs. This will achieve ease of use for the users by providing a unified programming model that better matches the application patterns.

More optimized implementations can present 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 optimized group communication algorithms in the background.

Map-Collectives can support iteration level as well as task level fault tolerance. Iteration level fault tolerance re-executes the iteration in case of any failures. In this case, only the iteration results (smaller loop variable data) will be checkpointed. This is preferred when the iterations are relatively finer grained.

IV. MAP-ALLGATHER COLLECTIVE

AllGather is an all-to-all collective communication operation that gathers data from all the workers and distributes the gathered data right back to them[7]. AllGather pattern can be noticed 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 matrix-vector multiplication, where each Map task outputs part of the resultant vector. In this computation we would use the Reduce and Merge tasks to assemble the vector together and then broadcast the assembled vector to workers.

Data-intensive iterative applications that have the AllGather pattern include Multi-Dimensional Scaling (matrix-matrix multiplication) [8] and PageRank using inlinks matrix (matrix-vector multiplication).

A. Model

We developed a Map-AllGather iterative MapReduce primitive similar to the MPI AllGather[7] collective communication primitive to support applications in a more efficient manner.

1) Execution model

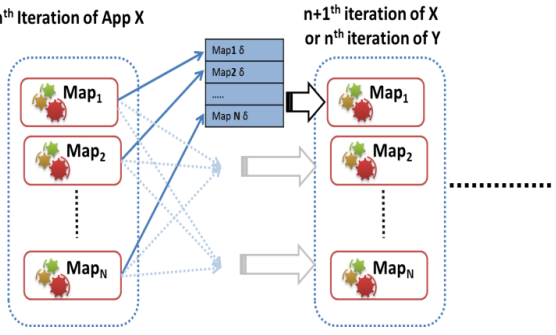

Figure 2. Map-AllGather Collective

Map-AllGather primitive broadcasts the Map Task outputs to all computational nodes (all-to-all communication) of the 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 flow of a Map-AllGather operation is Map phase followed by AllGather all-to-all communication followed by the AllGather combine. Map-AllGather substitutes the Map output processing (collect, spill, merge), Reduce task (shuffle, sort, reduce, write), Merge task (shuffle, barrier, execute), broadcast, and the barriers associated with these steps, with a single powerful and optimized AllGather operation.

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. 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 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 the default combiner. A 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 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 expects the Map outputs to be in <int, double[]> format. In a 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 therefore ignored.

Users can utilize their Map function implementations as is with the Map-AllGather primitive. They only need to specify the collective operation, after which the shuffle and reduce phases of MapReduce would get substituted by the Map-AllGather communication and computations.

3) Cost Model

Using an optimized implementation of AllGather, such as a bi-directional exchange-based implementation[7], we can estimate the cost of the AllGather component as follows using the Hockney model[3, 9], where α is the latency and β is the transmission time per data item (1/bandwidth)), m is the number of Map tasks and n_v is the size of AllGather data.

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

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

B. Fault tolerance

All-Gather partial data transfers from Map tasks to worker nodes can fail due to communication mishaps and other breakdowns. When task level fault tolerance is enabled, it's possible for the workers to retrieve any missing Map output data from the persistent storage (e.g. HDFS) to successfully perform the All-Gather computation.

The fault tolerance and the speculative execution of MapReduce enable possible duplicate execution of tasks. Map-AllGather can perform the duplicate data detection before the final assembly of the data at the recipient nodes to handle any duplicate executions.

C. Benefits

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

Oftentimes the Map task execution times are inhomogeneous[10] in typical MapReduce computations. Implementations of Map-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. The benefit will be even more significant when we have multiple waves of Map tasks.

In addition to improving the performance, this primitive also enhances 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.

V. 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[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.

A. Model

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

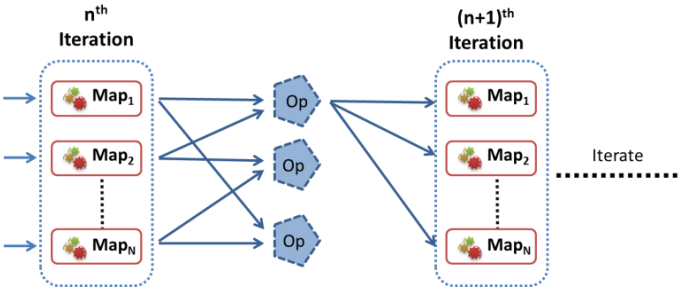

Figure 3. Map-AllReduce collective

1) Execution Model

The computation and communication pattern of a Map-AllReduce computation is a Map phase followed by the AllReduce communication and computation (reduction), as depicted in Figure 3. This model allows us to substitute the shuffle→sort→reduce→merge→broadcast steps of MR-MB with AllReduce communication in the communication layer.

The AllReduce phase can be implemented efficiently using algorithms such as bidirectional exchange (BDE) [7] or hierarchical tree-based reduction.

Map-AllReduce 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 communicating them to all the workers.

2) Data Model

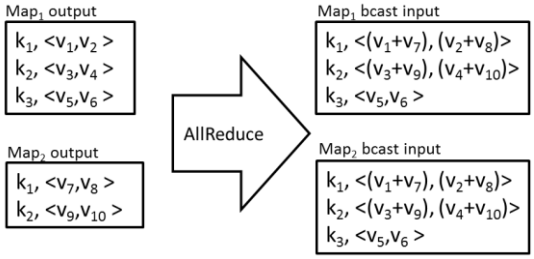

Figure 4. Example Map-AllReduce with Sum operation

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 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, 10 distinct Map output keys 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 the 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, Map-AllReduce has the ability to start combining the values as soon as the first Map task completes. It also allows the Map-AllReduce implementations to use reduction trees or bidirectional exchanges to optimize the 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 Map-AllReduce result or to check for the iteration termination condition. It would be executed in each worker node after all the Map-AllReduce data has been received.

```
list<Key, IOpRedValue> postOpRedProcess(
                list<Key, IOpRedValue> opRedResult);
```

3) Cost Model

An optimized implementation of Map-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 cost of AllReduce computation is small. Map-AllReduce substitutes the Map output processing, Reduce task, Merge task and broadcast overheads.

Other efficient algorithms to implement AllReduce communication include flat-tree/linear, pipeline, binomial tree, binary tree, and k-chain trees[3].

B. Fault Tolerance

If the AllReduce communication step fails for some reason, it's possible for the workers to read the 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. The most trivial fault tolerance model for Map-AllReduce would be a best-effort mechanism, where Map-AllReduce 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 done 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 localized areas of the reduction tree.

C. Benefits

Map-AllReduce reduces 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 optimize the computation by performing hierarchical reductions, reducing the number and the size of intermediate data communications. 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 n^{th} level in rack level, etc. The mapper level would be similar to the "combine" operation of vanilla MapReduce. 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.

VI. Implementations

In this section we present two implementations of Map-Collectives for Hadoop MapReduce and Twister4Azure iterative MapReduce. Map-AllGather is implemented using linear all-to-all communications, where each Map output is broadcasted to all the workers taking advantage of the inhomogeneous running times to avoid congestion. Map-AllReduce is implemented using hierarchical reduction trees.

We present sufficiently optimal proof-of-concept implementations of the primitives to show the performance efficiencies that can be gained through using even a modest implementation of these primitives. It's possible to further optimize these implementations 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[7]. The ability to improve the primitive implementations without

changing the user application makes it possible to optimize them as a future work.

It is not our objective to find the most optimal implementations for each of the environments, especially for Clouds since that might end up being a moving target due to the rapidly evolving and black box nature of Cloud environments. This presents an opportunity for Cloud providers to offer optimized implementations of these primitives as cloud infrastructure services that can be utilized by the frameworks.

A. H-Collectives: Map-Collectives for Apache Hadoop

H-Collectives is the Map-Collectives implementation for Apache Hadoop that can be used as a drop-in library with the Hadoop distributions. H-Collectives uses the Netty NIO library, node-level data aggregations and caching to efficiently implement the collective communications and computations. Existing Hadoop Mapper implementations can be used with these primitives with only very minimal changes. These primitives work seamlessly with Hadoop dynamic scheduling of tasks, support for multiple Map task waves, and other desirable features of Hadoop while supporting the typical Hadoop fault tolerance and speculative executions as well.

A single Hadoop node may run several Map workers and many more Map tasks belonging to a single computation. The H-Collectives implementation maintains a single node-level cache to store and serve the collective results to all the tasks executing in a worker node.

H-Collectives speculatively schedules the tasks for the next iteration, and the tasks are 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.

Task level fault tolerance checkpoints Map task output data to HDFS using a background daemon, avoiding overhead to the computation. In case this checkpointing fails for some reason, failed Map tasks or even the whole iteration can be re-executed.

1) H-Collectives Map-AllGather

This performs TCP-based best effort broadcasts for each Map task output. Task output data is transmitted as soon as a task is completed, taking advantage of the inhomogeneous Map task completion times. Final aggregation of these data products is done at the destination nodes only once per node. If an AllGather data product is not received through the TCP broadcasts, then it will be fetched from the HDFS.

2) H-Collectives Map-AllReduce

H-Collectives Map-AllReduce use n'ary tree-based hierarchical reductions, where Map task level and node level reductions would be followed by broadcasting of the locally aggregated values to the other worker nodes. The final reduce operation is performed in each of the worker nodes and is done after all the Map tasks are completed and the data is transferred.

B. Map-Collectives for Twister4Azure iterative MapReduce

Twister4Azure Map-Collectives are implemented using the Windows Communication Foundation (WCF)-based Azure TCP inter-role communication mechanism, while employing the Azure table storage as a persistent backup.

Twister4Azure collective implementations maintain a worker node-level cache to store and serve the collective result values to all the tasks executing in that node. Twister4Azure utilizes the collectives to perform synchronization at the end of each iteration. It also uses the collective operations to communicate the new iteration information to the workers to aid in the decentralized scheduling of the tasks for the next iteration.

1) Map-AllGather

Map-AllGather performs simple TCP-based broadcasts for each Map task output. Workers start transmitting the data as soon as a task is completed. The final aggregation of the data is performed in the destination nodes and is done only once per node.

2) Map-AllReduce

Map-AllReduce 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. 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.

VII. EVALUATION

In this section we evaluate and compare the performance of Map-Collectives with plain MapReduce using two real world applications, Multi-Dimensional Scaling and K-means clustering. The performance results are presented by breaking down the total execution time into the different phases of the MapReduce or Map-Collectives computations, providing a more finely detailed performance model. This provides a better view of various overheads in MapReduce and the optimizations provided by Map-Collectives to reduce some of those overheads.

In the following figures, ‘Scheduling’ is the per iteration (per MapReduce job) startup and task scheduling time. ‘Cleanup’ is the per iteration overhead from Reduce task execution completion to the iteration end. ‘Map overhead’ is the start and cleanup overhead for each Map task. ‘Map variation’ is the overhead due to variation of data load, compute and Map overhead times. ‘Comm+Red+Merge’ is the time for 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. The common components (data load, compute) are plotted at the bottom to highlight variable components.

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

A. Multi-Dimensional Scaling (MDS) using Map-AllGather

The objective of MDS is to map a dataset in high-dimensional space to a lower dimensional space, with respect to the pairwise proximity of the data points[8]. In this paper, we use parallel SMACOF[11, 12] MDS, which is 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 output 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 by the BCCalc step of the next iteration. MDS performs a relatively smaller amount of computations for a unit of input data. Hence MDS has larger data loading and memory overhead. Usage of the Map-AllGather primitive in MDS BCCalc computation eliminates the need for reduce, merge and broadcasting steps in that particular computation.

1) H-Collectives MDS Map-AllGather

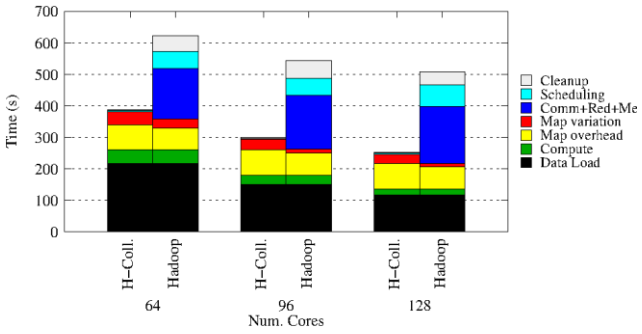

Figure 5. MDS Hadoop using only the BC Calculation MapReduce job per iteration to highlight the overhead. 20 iterations, 51,200 data points.

We implemented the MDS 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 5 shows the MDS strong scaling performance results, highlighting the overhead of different phases on the computation. We used only 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 into a 51200*3 matrix.

As seen in Figure 5, 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 few seconds between the scheduling to perform housekeeping tasks.

2) Twister4Azure MDS Map-AllGather

We implemented MDS for Twister4Azure using Map-AllGather primitive and MR-MB with optimized broadcasting. Twister4Azure optimized broadcast is an improvement over simple MR-MB as 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 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 by 13-42% over MapReduce with optimized broadcast in the current test cases.

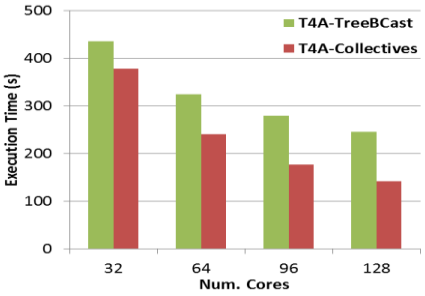

Figure 6. MDS application implemented using Twister4Azure. 20 iterations. 51,200 data points (~5GB).

3) Detailed analysis of overhead

This section presents a detailed analysis of overhead in the Hadoop MDS computation. Only the BCCalc MapReduce job is used. MDS computations 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.

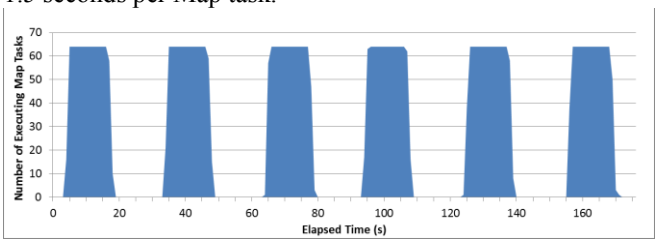

Figure 7. Hadoop MapReduce MDS-BCCalc histogram

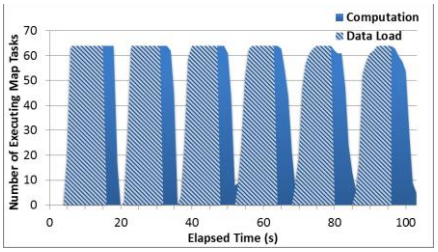

Figure 8. H-Collectives AllGather MDS-BCCalc histogram

Figure 7 presents the MDS using Hadoop MapReduce. Figure 8 presents MDS using H-Collectives AllGather implementation. These plot the total number of executing Map tasks at a given moment of the computation, which approximately represents the amount of useful work done in the cluster at that given moment. Each blue bar represents an iteration of the computation. The width of each blue bar indicates the time spent by Map tasks in that particular iteration. This includes input data loading, calculation and output data storage. The space between the blue bars represents the remaining overheads of the computation.

In Figure 8, the striped section on each blue bar represents the data loading time. As can be seen, the overhead between the iterations virtually disappears with the use of the Map-AllGather primitive.

4) Performance difference of Twister4Azure vs. Hadoop

Twister4Azure is already optimized for iterative MapReduce[2] and contains very low scheduling, data loading and data communication overheads compared to Hadoop. In terms of the overheads and comparison purposes, Twister4Azure can be considered as a near ideal Hadoop for iterative computations. Hence the overhead reduction we achieve by using Map-collectives is low in Twister4Azurecompared to H-Collectives. A major component of Hadoop MDS is due to the data loading, which Twister4Azure avoids by using data caching and cache-aware scheduling.

B. K-means Clustering using Map-AllReduce

K-means Clustering[13] 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 in the Reduce task, while centroid data is broadcasted at the beginning or end 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) are combined independently and the resultant key-value pairs (new centroids) are distributed to all the Map tasks of the next iteration. K-means Clustering has relatively smaller data loading and memory overhead vs. the number of computations compared to the MDS application discussed above.

1) H-Collectives K-means Clustering-AllReduce

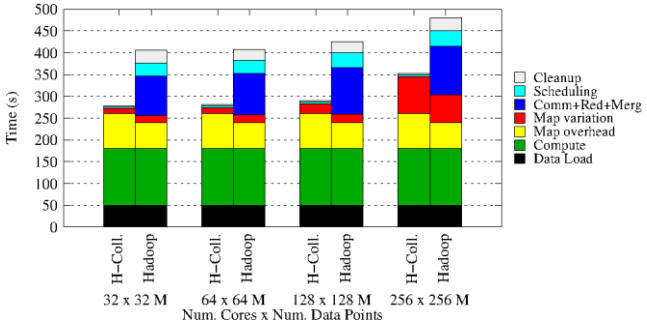

Figure 9. Hadoop K-means Clustering comparison with H-Collectives Map-AllReduce Weak scaling. 500 Centroids, 20 Dimensions, 10 iterations.

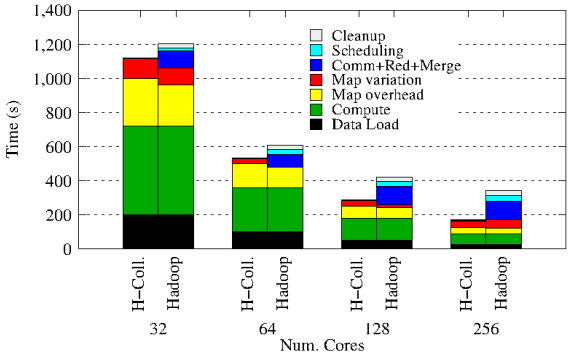

Figure 10. Hadoop MapReduce K-means Clustering & H-Collectives Map-AllReduce Strong scaling. 500 Centroids, 20 Dimensions,10 iterations.

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

Figure 9 illustrates the K-means Clustering weak scaling performance where we scaled the computation while keeping the workload per core constant. Figure 10 presents the K-means Clustering strong scaling performance, where we scaled the computation while keeping the data size constant. Strong scaling test cases with a smaller number of nodes use more Map task waves optimizing the intermediate data communication, resulting in relatively smaller overhead for the computation

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. A slight increase of Map task overhead and Map variation can be noticed in the case of Map-AllReduce based implementation, similar to the behavior observed and explained in MDS section 7.a.1.

2) Twister4Azure K-means Clustering-AllReduce

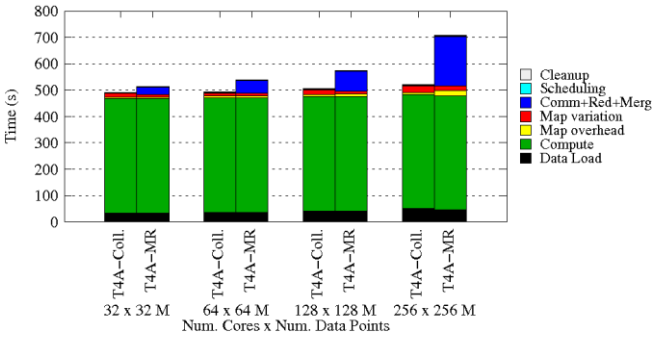

Figure 11. Twister4Azure K-means weak scaling with Map-AllReduce. 500 Centroids, 20 Dimensions. 10 iterations. 32 to 256 Million data points.

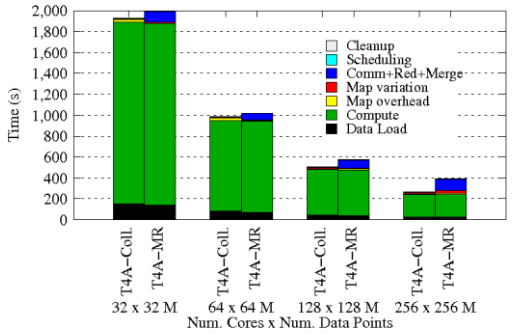

Figure 12. Twister4Azure K-means Clustering strong scaling. 500 Centroids, 20 Dimensions, 10 iterations. 128 million data points.

We implemented the K-means Clustering application for Twister4Azure using the Map-AllReduce primitive and MapReduce-MergeBroadcast. MR-MB implementation uses in-map combiners to perform local aggregation of the output 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 can be seen in these figures, the Map-AllReduce implementation gets rid of the communication, reduce and merge overheads of the MR-MB computation.

VIII. Background and Related Works

A. Collective Communication Primitives

Collective communication operations[6] facilitate optimized communication and coordination between groups of nodes of a distributed computation, and are used heavily in the MPI type of HPC applications. These powerful operations make it much easier and efficient to perform complex data communications and coordination 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 numerous algorithms and topologies suited to different environments and use cases. The best 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, 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 further categorize collective communication primitives based on the communication patterns as well, such as All-to-One (gather, reduce), One-to-All (broadcast, scatter), All-to-All (allgather, allreduce, reduce-scatter) and Synchronization (barrier).

The MapReduce model supports the All-to-One operations through the Reduce step. The broadcast operation of MR-MB model (section II) serves as an alternative to the One-to-All type operations. The MapReduce model contains a barrier between the Map and Reduce phases and the iterative MapReduce has a barrier between the 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 MR-MB model. For example, the 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.

B. MapReduce and Apache Hadoop

MapReduce, introduced by Google[14], consists of a programming model, storage architecture and an associated execution framework for distributed processing of very large datasets. MapReduce frameworks take care of data partitioning, task parallelization, task scheduling, fault tolerance, intermediate data communication, and many other aspects of these computations for the users. MapReduce provides an easy to use programming model, allowing users to utilize 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. The main goals of MapReduce frameworks include framework-managed fault tolerance, ability to run on commodity hardware, ability to process very large amounts of data, and horizontal scalability of compute resources.

Apache Hadoop[15], together with Hadoop distributed parallel file system (HDFS) [16], provides a widely used open source implementation of MapReduce. Hadoop supports data locality-based scheduling and reduces the data transfer overhead by overlapping intermediate data communication with computation. Hadoop performs duplicate executions of slower tasks and handles failures by rerunning the failed tasks using different workers. MapReduce frameworks like Hadoop trade off costs such as large startup overhead, task scheduling overhead and intermediate data persistence overhead for better scalability and reliability

C. Iterative MapReduce and Twister4Azure

Data-intensive iterative MapReduce computations are a subset of iterative computations, where individual iterations can be specified as MapReduce computations. Examples of applications that can be implemented using iterative MapReduce include PageRank, Multi-Dimensional Scaling[1, 17], K-means Clustering, Descendent query[5], LDA, and Collaborative Filtering with ALS-WR.

These data-intensive iterative computations can be performed using traditional MapReduce frameworks like Hadoop by manually scheduling the iterations from the job client driver, albeit in an un-optimized manner. However, there exist many possible optimizations and programming model improvements to enhance the performance and usability of the iterative MapReduce programs. Such optimization opportunities are highlighted by the development of many iterative MapReduce frameworks such as Twister[4], HaLoop[5], Twister4Azure[1], Daytona[18] and Spark[19]. Optimizations exploited by these frameworks include caching of loop-invariant data, cache-aware scheduling of tasks, iterative-aware programming models, direct memory streaming of intermediate data, iteration-aware fault tolerance, caching of intermediate data (HaLoop reducer input cache), dynamic modifications to cached data (e.g. genetic algorithm), and caching of output data (in HaLoop).

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, performing cache-aware scheduling, optimizing intermediate data transfers, optimizing data broadcasts and many other optimizations described in Gunarathne et al[1].

IX. Future works – Map-ReduceScatter

There are iterative MapReduce applications where only a small subset of loop invariant data product is needed to process the subset of input data in a Map task. In such cases, it's inefficient to make all the loop invariant data available to such computations. In some of these applications, the size of loop variant data is too large to fit into the memory and introduce communication and scalability bottlenecks as well. An

example of such a computation is PageRank. The Map-ReduceScatter primitive, modeled after MPI ReduceScatter, is aimed to support such use cases in an optimized manner.

Map-ReduceScatter gets rid of the inefficiency of simple broadcast ofing all the data to all the workers. Another alternative approach is to perform a join of loop invariant input data and loop variant data using an additional MapReduce step. However, this requires all the data to be transported over the network from Map tasks to Reduce tasks, making the computation highly inefficient.

Map-ReduceScatter primitive is still a work in progress and we are planning on including more information about it in our future publications.

X. Conclusions

We introduced Map-Collectives, collective communication operations for MapReduce inspired by MPI collectives, as a set of high level primitives that encapsulate some of the common iterative MapReduce application patterns. Map-Collectives improve the communication and computation performance of the applications by enabling highly optimized group communication across the workers, getting rid of unnecessary/redundant steps, and by enabling the frameworks to use a poly-algorithm approach based on the use case. Map-Collectives also improve the usability of the MapReduce frameworks by providing abstractions that closely resemble the natural application patterns. They also decrease the implementation burden on the developers by providing optimized substitutions for certain steps of the MapReduce model. We envision a future where many MapReduce and iterative MapReduce frameworks support a common set of portable Map-Collectives and consider this work as a step in that direction.

In this paper, we defined Map-AllGather and Map-AllReduce Map-Collectives and implemented Multi-Dimensional Scaling and K-means Clustering applications using these operations. We also presented the H-Collectives library for Hadoop, which is a drop-in Map-Collectives library that can be used with existing MapReduce applications with only minimal modification. We also presented a Map-Collectives implementation for Twister4Azure iterative MapReduce framework as well. MDS and K-means applications were used to evaluate the performance of Map-Collectives on Hadoop and on Twister4Azure, depicting up to 33% and 50% speedups over the non-collectives implementations by getting rid of the communication and coordination overheads.

Acknowledgment

We would like to thank colleagues and members of the Digital Science Center at Indiana University for helpful discussions and contributions to Twister4Azure and the present work. We gratefully acknowledge support from Microsoft for Azure Cloud Academic Resources Allocation, which was critical for our experiments. Thilina Gunarathne was supported by a fellowship sponsored by Persistent Systems.

References

[1] T. Gunarathne, B. Zhang, T.L. Wu, and J. Qiu, "Portable Parallel Programming on Cloud and HPC: Scientific Applications of Twister4Azure," Proc. Fourth IEEE International Conference on Utility and Cloud Computing (UCC), pp 97-104, 5-8 Dec. 2011, doi: 10.1109/UCC.2011.23.

[2] T. Gunarathne, B. Zhang, T.-L. Wu, and J. Qiu, "Scalable parallel computing on clouds using Twister4Azure iterative MapReduce," Future Generation Computer Systems, vol. 29, pp. 1035-1048, Jun 2013.

[3] J. Pjesivac-Grbovic, T. Angskun, G. Bosilca, G. E. Fagg, E. Gabriel, and J. J. Dongarra, "Performance analysis of MPI collective operations," Proc. 19th IEEE International Parallel and Distributed Processing Symposium, 2005, doi:10.1109/IPDPS.2005.335

[4] J.Ekanayake, H.Li, B.Zhang, T.Gunarathne, S.Bae, J.Qiu, et al., "Twister: A Runtime for iterative MapReduce," Proc. First International Workshop on MapReduce and its Applications at ACM HPDC, 2010,pp 810-818, doi: 10.1145/1851476.1851593

[5] Y. Bu, B. Howe, M. Balazinska, and M. D. Ernst, "HaLoop: Efficient Iterative Data Processing on Large Clusters," Proc. VLDB Endow., vol. 3, pp. 285-296, Sep 2010.

[6] MPI Forum, "MPI: A Message-Passing Interface Standard, Version 3.0"[Online], Sep 2012, Available: http://www.mpi-forum.org/docs/mpi-3.0/mpi30-report.pdf

[7] E. Chan, M. Heimlich, A. Purkayastha, and R. van de Geijn, "Collective communication: theory, practice, and experience," Concurrency and Computation: Practice and Experience, vol. 19, pp. 1749-1783, 2007.

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

[9] R. W. Hockney, "The communication challenge for MPP: Intel Paragon and Meiko CS-2," Parallel Computing, vol. 20, pp. 389-398, 1994.

[10] J. Ekanayake, T. Gunarathne, and J. Qiu, "Cloud Technologies for Bioinformatics Applications," Parallel and Distributed Systems, IEEE Transactions on, vol. 22, pp. 998-1011, 2011.

[11] S.H. Bae, J. Y. Choi, J. Qiu, and G. C. Fox, "Dimension reduction and visualization of large high-dimensional data via interpolation," Proc. 19th ACM International Symposium on High Performance Distributed Computing, 2010, pp 203-214, doi:10.1145/1851476.1851501

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

[13] S. Lloyd, "Least squares quantization in PCM," Information Theory, IEEE Transactions on, vol. 28, pp. 129-137, 1982.

[14] J. Dean and S. Ghemawat, "MapReduce: simplified data processing on large clusters," Commun. ACM, vol. 51, pp. 107-113, 2008.

[15] ASF, "Apache Hadoop" [Online], Available: http://hadoop.apache.org/core/. (retrieved: Mar 2014)

[16] ASF, "Hadoop Distributed File System - HDFS" [Online], Available: http://hadoop.apache.org/hdfs/ (retrieved: Mar 2014)

[17] Z. Bingjing, R. Yang, W. Tak-Lon, J. Qiu, A. Hughes, and G. Fox, "Applying Twister to Scientific Applications," Proc. Second IEEE International Conference on Cloud Computing Technology and Science (CloudCom), pp 25-32, Dec 2010, doi: 10.1109/CloudCom.2010.37

[18] Microsoft, "Microsoft Daytona" [Online], Available : http://research.microsoft.com/en-us/projects/daytona/. (ret: Mar 2014)

[19] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster Computing with Working Sets," 2nd USENIX conference on Hot Topics in Cloud Computing (HotCloud '10), Boston, 2010.