# Efficient, Fault-tolerant and Elastic Allreduce Framework For Iterative Algorithms

Bingjing Zhang, Judy Qiu
Indiana University
Bloomington, IN, USA

Shravan Narayanamurthy, Markus Weimer
Microsoft Research
Redmond, WA, USA

***Abstract*— Many learning algorithms fitting into Statistical Query Model can be written in a certain summation form [1]. In this form, the calculation can be easily distributed: iterations of local data computation and subsequent global data aggregation and redistribution. However, this parallelism pattern is not well presented in the current Apache Big Data Stack. On the other hand, MPI has long adopted these global data aggregation and redistribution operations. Since this technique deeply anchors in the HPC systems, it cannot be transplanted to clouds easily. In this paper, we express the algorithms as iterations of allreduce operations and develope an allreduce framework on top of the REEF framework to support an efficient, fault-tolerant and elastic way to run the algorithms. Instead of using traditional broadcast + reduce with tree topology on allreduce operators, for efficiency, we apply Hypercube-like topology, making the time complexity of allreduce operations O(nlogp) for unsplittable data objects and O(2n) for chunked data objects, where p is the number of processes and n is the size of data objects in each process. For fault tolerance, unlike popular disk-based checkpointing methods which only ensures computation fault tolerance, we design a fault tolerant allreduce operator and recover the computation from failures using an algorithmic method by re-synchronizing global data through allreduce operations. For elasticity, a property not owned by many contemporary tools in which only the number of tasks is fixed, we make computation tasks join or quit allreduce topology dynamically so that the algorithm can be executed in ramp up mode and failure ignorance mode, thereby the efficiency on online processing is improved and the expenses on cloud resources can be saved.**

***Keywords*—allreduce; iterative algorithm; big data processing**

## I. Introduction

Past research shows that iterative algorithms following Statistical Query Model in which the algorithms calculate sufficient statistics or gradients fitting in the model, and are expressible as a summation over data points, such as locally weighted linear regression, nave bayes, Gaussian discriminative analysis, k-means clustering, logistic regression, neural network, principal components analysis, independent component analysis, independent component analysis, expectation maximization, and support vector machine, can be parallelized as iterations of local data computation with corresponding global data aggregation and redistribution [1].

Nowadays many such kinds of iterative applications are implemented in the MapReduce model for big data processing [2]. However, these MapReduce implementations suffer from repeated input data loading from the distributed file system and slow diskbased intermediate data communication (shuffling) through iterations. It has been ten years since MapReduce

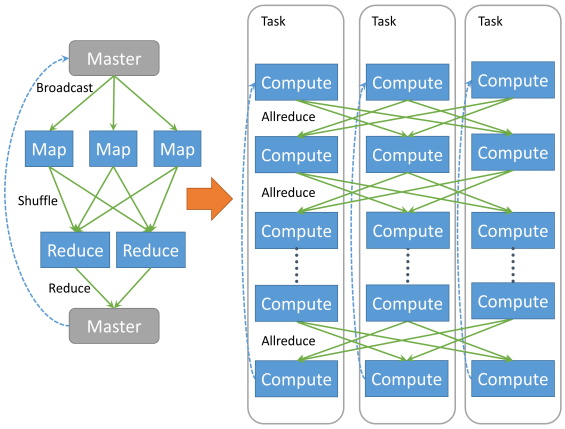


Fig. 1: Comparison between a typical (iterative) MapReduce application flow and an iterative allreduce application flow.

came onto the scene of computing [3], and in this time computers have gained more and more memory capacity, as such many new MapReduce-like tools [4][5] are designed to utilize memory for data caching and communication instead of using disk-based operations to improve the performance of iterative algorithms.

However, the global data aggregation and redistribution, this kind of communication pattern in the iterative algorithms does not appear in initial MapReduce model. Later in Hadoop [6] MapReduce, data aggregation is added through a single Reduce task and then redistribution occurs using distributed cache. Other MapReduce-like tools prefer to add additional APIs/components to complete this process: first gather the data to a driver/master task and compute the final results, then broadcast them to all the tasks for the next round of computation [4][5]. However, using broadcast + reduce not only shows the limitation of the original MapReduce model, but also results in bad performance upon implementation. While there are engineering enhancements [7][8][9][10], it does not change the fact that data movement has to be done in two rounds and the master/driver task is always a bottleneck in the communication. Additionally, these tools using broadcast + reduce only ensure the fault tolerance of the computation, but not the process of the communication itself. Once the communication fails, the master has to roll back the computation to the last disk checkpoint. By contrast, MPI [11] exposes all the global data aggregation and redistribution operations as collective communication operations. These can be divided into two

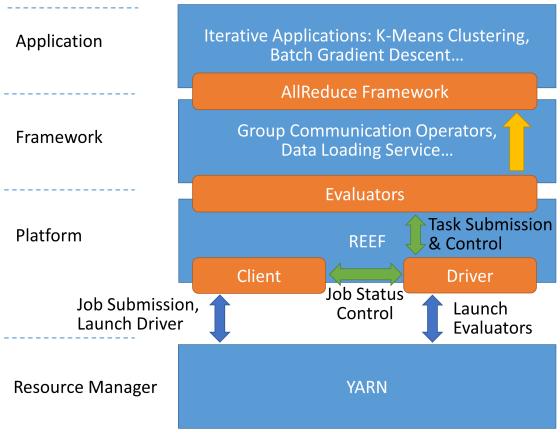


Fig. 2: REEF Architecture

categories [12]. One is data redistribution operations which include broadcast, scatter, gather, and allgather. The other is data consolidation operations, including reduce, reduce-scatter and allreduce. The MPI collective communication operations are implemented with good performance. Despite this they cannot be used in Apache Big Data Stack directly because MPI is designed almost exclusively for HPC systems whose infrastructure is very different from cloud environments.

In this paper, instead of using broadcast + reduce, we directly use an allreduce operation to express the data aggregation and redistribution in iterative applications (see Fig. 1). We propose a new framework which describes the iterative applications as iterations of parallel local data computations and allreduce operations that are able to run these applications in an efficient, fault tolerant and elastic way. We implement the allreduce framework on REEF [13] (see Fig. 2). REEF provides a reusable control plane for scheduling and coordinating task-level work on cluster resource managers. With REEF, we can simplify task failure detection, as well as task adding or removing.

There are several contributions in our research work. In this solution, we apply a hypercube-like topology on allreduce operators and provide two different allreduce algorithms for efficiency. In cloud systems, which use relatively slow machines and networks, the failure rate is high. Instead of using expensive disk-based checkpointing [14], we make allreduce operators be fault tolerant and use an algorithmic way to recover the computation from failures. Applications on cloud systems may also need to handle elasticity events to expand or shrink the computation dynamically to save expenses on resource usage. This aspect is not covered by current tools which only support a fixed number of tasks. We design a mechanism to allow tasks to join in or leave from the allreduce topology dynamically.

For the remainder of this paper, Section 2 talks about the programming model in the allreduce framework. Section 3 demonstrates how the applications are written in the framework. Section 4 introduces the hypercube topology and how it is applied on allreduce algorithms. Section 5 describes how failure and elasticity events are handled in the allreduce framework. Section 6 shows experiment results. We give discussions about the related work in Section 7 and conclusions in Section 8.

TABLE I: Allreduce Operation Interfaces

| |
|---|
| AllReduceResult $< T >$ allreduce(T aElement); |
| AllReduceResult$< T >$ allreduce(T aElement); |

## II. ALLREDUCE PROGRAMMING MODEL

In allreduce programming model, the computations in each task are connected by allreduce operations. By definition, allreduce is an all-to-all communication. Every task is a participator and no task can control the whole process as the master. In allreduce operations, each task takes the local data as the input and outputs the result returned by the reduce function which runs on all the input data. The reduce function is required to be commutative and associative.

In REEF, allreduce operators are firstly configured and then used in the tasks. Configuration happens on the driver side. Each allreduce operator is defined with a name and related reduce function object. Later, at the task side, every operator is fetched from the task context according to its name. The allreduce operation returns an AllReduceResult object or an AllReduceResultList object. If allreduce succeeds, it contains the output. If not, it shows it has an empty value. Because allreduce is an all-to-all operation, if a portion of the tasks fail, some other tasks may also fail to accomplish the communication. As a result, among those tasks staying alive, some may have the result while others do not. Through this manner, we expose the execution failure of allreduce operation to the application (see Table I).

This definition of allreduce in the allreduce framework is different from the original allreduce definition in MPI. Failures are not exposed to the application; instead the framework automatically recovers all the tasks back to a consistent system state through checkpointing and rollback [15]. Therefore the allreduce operator in MPI assumes it can always get the correct output. However in the cloud environment, it is unlikely we can make the execution pause and synchronize all the tasks to check if each remaining task has the final result. As such, we have to loosen original strict allreduce definitions and expose failures to the applications, granting them flexibility on how they continue with the computation. We also provide an API to check and update the allreduce topology changes if a failure is detected. In the next section, we will show how to combine all these interfaces to write fault-tolerant and elastic iterative applications.

Finally, we describe the application control flow on each task as iterations of local computations and allreduce operations (see Fig. 3). In each iteration, the first allreduce operation is Control Message Allreduce, then one or more allreduce operations follow sequentially. The control flow redirects the control to the related local computation and the allreduce operation based on the results of the control message allreduce. If any of the allreduce operations fails, the control flow goes back to the beginning, updates the topology, redoes the control message allreduce operation and then redirects the control to one of the local computations again. For failure recovery, each

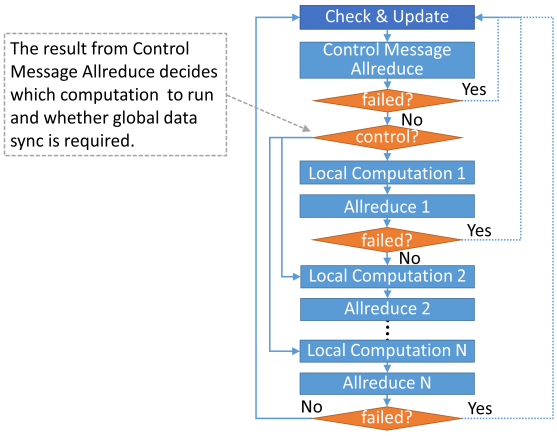


Fig. 3: Control Flow on Each Task in Allreduce Framework

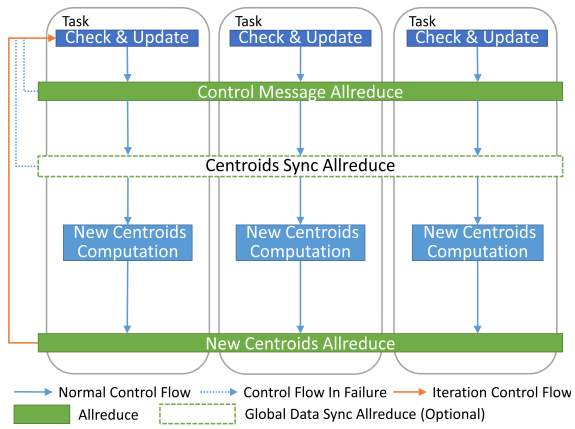


Fig. 4: Parallelism Pattern and Control Flow on Each Task in K-Means Clustering

computation may need to re-synchronize the global data, which is the allreduce output from the previous allreduce operation, among the tasks through another allreduce. In addition to the time used on task restart and data reloading, the main cost of failure recovery depends on the global data re-synchronization, which relies on the performance of allreduce operations. As long as this is efficient, we can greatly reduce the time cost on failure recovery.

## III. APPLICATIONS

As what we point out in previous sections, many iterative algorithms fitting into Statistical Query Model can be expressed as iterations of local computation plus subsequent allreduce operations. Here we use K-means Clustering [16][17] and Batch Gradient Descent [10] to demonstrate how the proposed fault-tolerant and elastic allreduce operator works.

### A. K-Means Clustering

In K-means Clustering, every iteration generates a new version of centroids (cluster centers) to use as the input in the next iteration. Each centroid is calculated through averaging the coordination values of points which belong to the same cluster center from the last iteration. In distributed K-means clustering, a task only owns a portion of data points so that any task only has a local summation of point coordination values after local computation. As a result, to generate the global new centroids for the next iteration, we use an allreduce operation with addition as the reduce function to get the global summation result.

When a failure happens, some tasks may lose the most updated version of the centroids and cannot continue the computation for the next iteration. In this situation, the broadcast operation is required to distribute the most updated centroids from one task to the rest. Here we use another allreduce operator to simulate the broadcast operation. The task which is identified to broadcast the most updated data puts the centroids into the allreduce operator as the input, while tasks which need the newest centroids for the current iteration input an empty value. In the reduce function, we keep the real data and ignore empty values. In Section 4, we will show how this operation simulation takes effect without influencing the performance.

Finally, the control flow of the K-Means clustering is as follows: at the beginning of each iteration, we invoke a allreduce operation called Control Message Allreduce. Using the status of all the tasks as the input, this allreduce operation determines control information such as the current iteration number, and whether global centroid data is required to be synchronized. Once the control information is decided, if data synchronization is required, the tasks start to perform allreduce on centroids synchronization first followed by new centroids calculation. If not, they apply allreduce on new centroids calculation directly. At the end of the iteration, or if a failure happens during the iteration, the task always goes back to the beginning of the control flow, checks and updates the allreduce topology, and applies Control Message Allreduce again (see Fig. 4).

### B. Batch Gradient Descent

Batch gradient descent (BGD) is another application example. It tries to minimize an objective function to

achieve maximum-likelihood estimation through moving the training model to the direction learned from the gradient calculated in each iteration. In a normal algorithm flow, the BGD application needs two allreduce operations per iteration. One is loss and gradient calculation and another is line search. The results are used to update a model vector which is held on all the tasks across iterations (see Fig. 5). We have seen that in K-Means Clustering, an additional allreduce operator is used to synchronize the global centroids data. Similarly, here we need two additional allreduce operators if failure occurs. A allreduce operator is used to synchronize the global parameter vector (the training model) before applying loss and gradient allreduce. Additionally an allreduce operator for the model vector and gradient descent vector is used before doing line search allreduce. The control flow is very similar to that in K-Means Clustering only with more steps. At the beginning of the iteration, control message allreduce is executed to decide the control status of the current iteration. It returns the current iteration number, determining which allreduce operation to execute and whether global data synchronization is required. Once an iteration ends (successfully or due to a failure), the control flow returns to the beginning of the iteration to

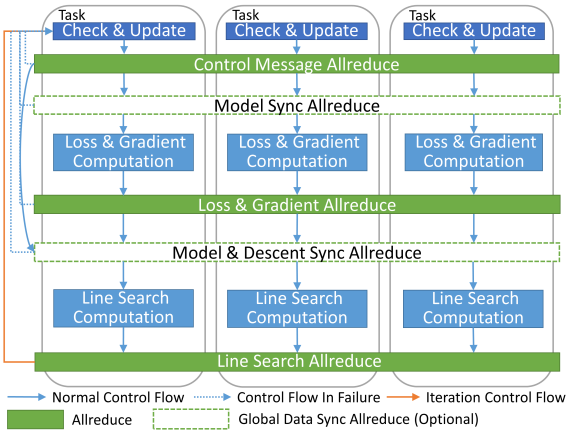


Fig. 5: Parallelism Pattern and Control Flow on Each Task in BGD

check and update the topology and launch Control Message Allreduce.

Now we have talked about the mechanism of failure recovery in two applications. The general method is to re-launch the failed tasks, update the topology, redo Control Message Allreduce and then perform global data synchronization. Noting that in REEF a task failure is exposed to the application driver, and it can decide if it just removes the failed task or needs to add a new task, it shows that our allreduce framework is not only able to recover from failures, where the original computation can be resumed, but also able to handle elasticity demands, where we can shrink or expand the computation scale.

We find that the BGD application, as a learning algorithm, can maintain the accuracy in the change of the computation scale, namely the elasticity in computation. It can be executed in ramp-up mode and failure-ignorance mode as shown in Section 6. In ramp-up mode, it can start with a small number of tasks with a portion of sample data if all the computation resources are not available yet. Then when more tasks are allocated, they will automatically join the computation at later iterations. In failure-ignorance mode, if some tasks fail and there are no computation resources available to replace them, the application can continue with the rest of the tasks. When these tasks are allocated later, they can still join back with the computation. BGD shows that in both modes, the application can obtain correct results.

## IV. ALLREDUCE TOPOLOGY AND ALGORITHMS

We use hypercube topology for the topology in allreduce operators. The original topology introduced in MPI requires the number of nodes to be a power of 2. Here we allow the topology to work on any number of p processes so as to easily add or remove nodes in O($log_2 p$)steps. There are two data exchange algorithms for allreduce operation in hypercube topology. One is for small unsplittable data and another is for large chunkable data. Both of them can achieve good theoretical performance.

### A. Hypercube Topology

A typical hypercube topology is a 3D cube with 8 nodes (see Fig. 6). Along the x-, y-, z-axes, each node has 3 different neighbors. To obtain the reduced value from all the nodes, all

nodes firstly exchange data with their neighbors on the x-axis, then on the y-axis, and finally on the z-axis.

A general hypercube topology is a simulation of a hypercube of dimension d which contains $2^d$ nodes. It is constructed from two hypercubes of dimension $d-1$ by connecting nodes with a corresponding index, then adding a leading binary bit to the index of each node [12]. For all nodes in one of the two hypercubes this leading bit is set to 0 and to 1 for the nodes in the other hypercube. In the example of 8 nodes: we firstly build a hypercube topology of 1 dimension with two nodes 0 and 1. Next we build a hypercube of two dimensions with 2 hypercubes of 1 dimension. Then we connect Node 0 with Node 2 and connect Node 1 with Node 3. Now we have a square. Finally we build a hypercube with 3 dimensions by combining two 2D hypercubes. Then we connect Node 0 with Node 4, Node 2 with Node 6, Node 1 with Node 5, and Node 3 with Node 7. In the allreduce algorithm, a node always exchanges data with its connected neighbor nodes.

### B. Allreduce Topology in REEF

We adapt this hypercube topology to an allreduce operator in REEF. However, in practice, the total number of nodes may not be a power of two. We can add additional rules to control the construction of the allreduce topology. We give each node an ID when adding it to the topology, which starts from 0 and increases in order. Dimension d is labeled from 0 to $d_{max}-1$. where $d_{max}$ is the max dimension of the hypercube and is calculated based on the current max node ID in the topology:

$$d_{max} = \begin{cases} 1, & \text{if } maxNodeID = 0 \\ \lceil log_2(maxNodeID+1) \rceil, & \text{if } maxNodeID > 0 \end{cases}$$

For example, in a hypercube of 8 nodes, nodes are numbered from 0 to 7 and the 3 dimensions are numbered as Dimension 0, 1 and 2. If some nodes are removed from the

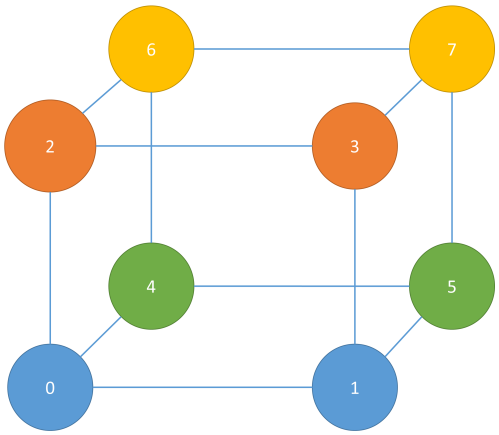


Fig. 6: Hypercube Topology of 8 Nodes

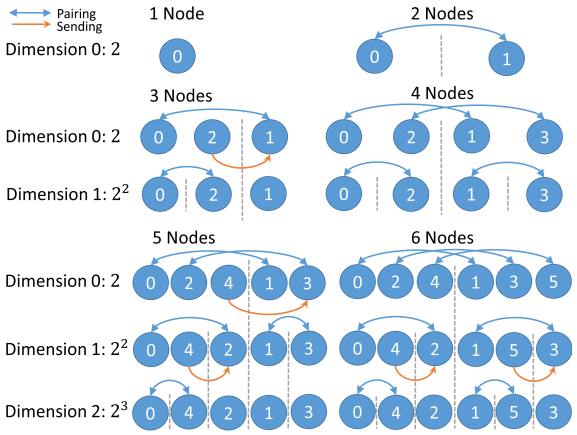


Fig. 7: Add Node 0 $\sim$ Node 5 to Allreduce Topology

topology, their IDs are kept in a freed ID list and reused for later node addition. Nodes are connected in three ways based on their communication: pairing, sending, or receiving. Pairing means the two connected nodes both send and receive data from each other. Sending means that one node sends data to the other node. A receiving node only receives data from the other node.

### C. Add a Node to Allreduce Topology

When a node with a given ID is added (the node ID is assigned either by adding 1 on the current max node ID, or being retrieved from a list with freed node IDs), we search its neighbor on this dimension. The algorithm is as follows:

1) Calculate the neighbor ID of this node ID for pairing.

$$neighborID = \begin{cases} nodeID - 2^d, & if \\ maxNodeID \bmod 2^{d+1} \geq 2^d \\ nodeID + 2^d, & if \\ maxNodeID \bmod 2^{d+1} < 2^d \end{cases}$$

2) If the nodeID for pairing is not available, find an alternative node to send data. Mark sending on the node, and receiving on the neighbor node.
$altNeighborID = neighborID \pm 2^{d+1}$
3) If no alternative node exists, mark the neighborID as -1.
4) If no alternative node is available, mark the neighborID as -2.
5) If the neighbor ID of the neighbor node (which is calculated based on Rule 1 and Rule 2) was originally assigned to -2, find all the nodes available in Rule 2, receive data from all these nodes.

Here we give examples by adding nodes from 0 5 to a topology. At the beginning, there is only one node, Node 0. Based on Rule 2, we find its neighbor for pairing is Node 1. But this ID is larger than the current max node ID. Based on Rule 2 and Rule 3, its neighbor node is set to -1. Then we add Node 1. Now the max dimension is 1. Based on Rule 1, we find its neighbor for pairing is Node 0. Then we connect Node 0 and Node 1.

Next we have Node 2. Now the max dimension is 2, so we need to find Node 2s neighbors on Dimension 0 and Dimension 1. On Dimension 0, based on Rule 1, we find its neighbor for pairing is Node 3. However, this node is not available in the topology. So we use Rule 2 and get its neighbor Node 1 for sending. On Dimension 1, based on Rule 1, we find that its neighbor node is Node 0. Next comes Node 3. On Dimension 0, we find its neighbor is Node 2. So Node 2 removes its original neighbor for sending, and connects to Node 3 for pairing. On Dimension 1, we connect Node 3 with Node 1 based on Rule 1. Now these 4 nodes form a complete hypercube of 2 dimensions. Similar strategies are also used when adding Node 4 and Node 5 (see Fig. 7).

### D. Remove a Node from Allreduce Topology

To remove a node, we extract it from the topology and return the ID to the freed node ID list. Then we check the neighbors of the node on all dimensions and do related modifications on each dimension. We obey the following rules:

1) If the neighbor node was "sending" or "pairing" to this node, find an alternative node to send the data towards.
$altNeighborID = neighborID \pm 2^{d+1}$
2) If no alternative node is available, mark the neighborID as -2.
3) If the neighbor node was receiving data from this node, remove the receiving directly.

Taking the topology of 6 nodes as an example, we want to remove Node 3 from the topology. Node 3 has two neighbors. They are Node 2 on Dimension 0, Node 1 and Node 5 on Dimension 1. Node 2 was pairing with Node 3 on Dimension 0. Based on Rule 1, we choose Node 5 as the new neighbor for Node 2 to send data towards. Node 1 was pairing with Node 3. Because there is no alternative node, based on Rule 2, we mark the neighbor ID as -2. Node 5 was sending data to Node 3. Based on Rule 2, its new neighbor ID is also -2. If we want to add back Node 3 later, we can use Rule 1 and 5 for node addition (see Fig. 8).

Both adding and removing a node takes about $O(log_2 p)$ steps. In the worst case when we need to apply Rule 5 in

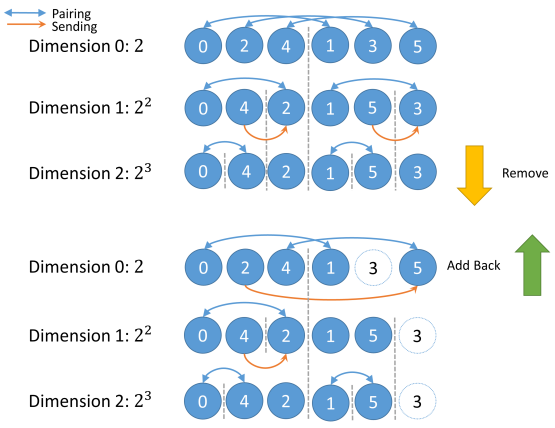


Fig. 8: Remove Node 3 and Add Node 3 Back

node addition or Rule 2 in node deletion, both take about $O(p)$ steps. For example, if all but one node with even ID numbers are removed, then the only node left has $p/2$ links. In this situation, both removing this node and adding it back later take about $p/2$ steps. But this is a very low possibility

### E. Allreduce Algorithms

We use two different algorithms to do allreduce on this topology. For unsplittable data, which is usually a small data object containing a few numbers, strings, and Boolean values, we exchange this kind of data object directly on the topology. If the topology is balanced, each node has around $O(log_2 p)$ neighbors and the allreduce time is about $O(nlog_2 p)$, where $n$ is the size of the data object in bytes and p is the number of processes.

For the chunkable data object, which is usually a large array, we use reduce-scatter and allgather to implement allreduce. Assuming the data object can be split into chunks with IDs from 0 to numChunks-1 (numChunks means the number of chunks), in reduce-scatter on every dimension d, each node sends the chunks to the neighbor with the following criteria: $chunkIDmod2^{d+1} = neighborIDmod2^{d+1}$ In this way, each task (which is a node in the topology) only needs to reduce a small number of chunks. Then in allgather, on each dimension $d$, nodes send the data originally owned after doing reduce-scatter and data received on the dimensions whose IDs are less than d. On a balanced topology, both processes take around $O(n)$ time complexity where n is the total size of all the chunks in bytes.

The performance could suffer when the topology becomes unbalanced. Potentially, if all but one node with odd ID numbers fail, then all the nodes with even ID numbers need to send data to the only node left, and the allreduce time complexity is about $O(np)$. One possible solution is to rebalance the whole topology when there is a large number of node failures.

These two allreduce algorithms can simulate broadcasting in global data synchronization. In this simulation, the task which owns the global data puts the data as the allreduce input while other tasks put an empty value. Inside of an allreduce function, we keep the real data and ignore empty values. With this method, we can have broadcasting done in $O(log_2 p)$ ime for unsplittable data and $O(2n)$ time for chunkable data.

## V. TOPOLOGY CONTROL

In REEF, task failure events and new task incoming events are reported to the application driver. We use this information to control topology update and notify client side topology update on the tasks dynamically. In this way, we can re-launch failed tasks for failure recovery, or directly shrink or expand the computation scale for the elasticity. This section will illustrate how tasks and the driver work together to track and update the allreduce topology.

### A. Iteration Control

Because there is no master to coordinate tasks in the allreduce operation, once a task enters an allreduce operation of an allreduce operator, it communicates with the neighbor nodes and then leaves the operation without waiting for the completion of the execution on other tasks. As a result, some tasks may finish the execution earlier and enter the next round of the allreduce operation while others remain in the current one.

We then employ iteration numbers to track the progress on each task. When the current iteration ends, the iteration number is updated and this indicates that the execution can enter the next. Later we will show how to use this information to guide topology updates. To synchronize the allreduce operations in one iteration, we dont update the iteration number of each allreduce operator immediately at the end of every allreduce operation, instead we wait and update it at the beginning of each iteration as a part of topology checking and update.

### B. Remove Failed Tasks

When some tasks fail, the event is reported to the task driver and processed inside of it. In this situation, the allreduce operation cannot be executed on the current topology, therefore the driver sends a message with type Source Dead to all the tasks in the current topology. From the viewpoint of a task, once it receives a Source Dead message, it will send an ACK message to the driver to report its current iteration number.

The driver waits for the arrival of all the task reports. It examines the iteration number on each task and finds the maximum number $maxIteration$. The driver generates the topology with the new iteration number$maxIteration + 1$ through deleting the nodes related to the failed tasks, then sends the new topology to all the tasks (each task gets a list of its own neighbors with related communication types). When a task receives the new topology, it stores the topology with the iteration number as the key. Once it is checked and updated, tasks are redirected to the new iteration with the new topology.

### C. Add New Tasks

Handling adding a new task is similar to handling removing a task. In contrast, the original topology can still work when a new task is coming. For the purpose of decreasing the number of tasks communicating with the driver and getting better scalability, the driver only communicates with the tasks which need to communicate with the new task.

When an event about an incoming new task is learned by the driver, it sends a message with type Source Add to the tasks. When all the related tasks get the message, they reply to the driver with an ACK message which contains their current iteration. Similarly, the driver waits for the arrival of all the ACK messages. It finds the current max iteration number and generates the topology for the new iteration $maxIteration+1$.

Once the tasks receive the new topology from the driver, they save it locally with its iteration number as the key. As opposed to the task failure method, tasks wont update to the new topology immediately at the coming iteration. They still go through the iterations one by one until the iteration which contains the new topology arrives. Because only about $log_2 p$ number of nodes are notified of the topology update, many other tasks are still working with the original pace.

### D. Control under Multiple Events and Multiple Operators

In the real execution environment, there may be many task failure events or new task adding events, even a mixture of both. This presents a challenge: how to ensure the integrity of the topology when it changes dynamically. We take the following 5 steps to do topology updates (see Fig. 9):

1) When a task event arrives, we copy the current active topology to a sandbox topology.
2) Process task adding/removing events on the sandbox topology.
3) We notify each task with a Source Dead/Add message and wait for the ACK message from each task. Notice that each task only gets one message; if a task already gets one Source Dead/Add message, it wont receive another one.
4) Once all replies are received, the driver sets the sandbox topology as the active topology and updates the related iteration number.
5) The driver sends the new topology to all the tasks including new tasks if they exist. The message also includes other information such as if there is a failure in the old topology or not. If the topology update is because of the failure, the tasks are redirected to the new iteration with the new topology in topology checking and updating. If the topology is updated because of new tasks, all the tasks just increase the iteration number as normal.

In this mechanism, the driver is responsible for failure detection and takes the initiative in topology update. The driver always waits for the arrival of replies from all the tasks related to the events. Then the topology update is consistent with the events processed.

We synchronize the topology update on all the allreduce operators in one iteration. Afterwards the operator topologies are checked and updated together at the end. This helps to avoid deadlocks and computation incorrectness caused by topology inconsistency.

There is a control parameter on the driver side which tells the minimum number of tasks required in the allreduce task group. If the minimum number of tasks is equal to the total number of tasks in the execution, when a failure happens, the driver has to re-launch the failed task otherwise the execution will pause. In this way, we make the computation fault-tolerant because we resume the original failed computation. If the minimum number of tasks is smaller than the total number of tasks, the driver will remove the task directly. In this way, we shrink the original computation scale and make the execution elastic.

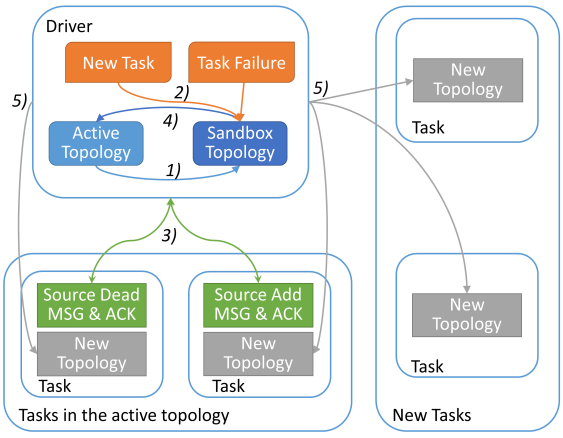


Fig. 9: Topology update control

## VI. EXPERIMENTS

We deployed REEF on the Big Red II [18] supercomputer at Indiana University. Following this we ran experiments on REEF Allreduce framework in which we benchmarked allreduce and tested two real applications, K-Means Clustering and Batch Gradient Descent. We allocated 128 compute nodes and ran up to 512 tasks.

### A. Test Environments

We used the nodes in the cpu queue on Big Red II for the experiments. Each of these nodes has 32 processors and 64GB memory, and are connected with Cray Gemini interconnect. Every experiment utilized 128 nodes, which is the maximum number that can be allocated on Big Red II, and we chose Cluster Compatibility Mode in job submission to let the compute nodes behave like nodes in a normal Linux cluster.

The following software was installed on Big Red II: JDK 1.7, Hadoop-2.5.0 and REEF 0.8. Hadoop is not naturally adopted by supercomputers such as Big Red II because each node does not have large local disk space to run HDFS. As a result we had to use only the local /tmp directory, which is mapped to 32GB of the total memory on a compute node.

In all the experiments, scaling was evaluated based on the number of tasks; in other words, the number of containers. We decided not to exploit the locality of the task assignment in the allreduce topology in order to explore the scalability by using a small number of compute nodes, thereby simulating a distributed environment with a large number of nodes. The allreduce benchmark started with 2 tasks and then kept doubling the number of tasks to 4, 8, 16, all the way up to 1024. In K-Means clustering, we ran strong scaling tests initially with 32 tasks and then continued doubling as described above. For Batch Gradient Descent, to easily select input data, we ran weak scaling tests on 10, 50, 100, 200 and 500 tasks.

### B. Allreduce Benchmarking

We benchmarked allreduce performance with three different methods: allreduce without chunking input data, allreduce with chunked input data, and broadcast + reduce using tree topology.

We first tested allreduce on 100MB data. The input data on each task is a double array with 13,107,200 and the reduce function is addition. We present the results using logarithmic scale based on 2. In the first test, we allocate only one task in each node (see Fig. 10). As what we analyzed for this paper, allreduce with chunked input data performs best of all. When the number of tasks increases, the performance does not change much, maintaining around 2.7 seconds. The execution

time of allreduce without chunking input data grows following a logarithmic scale. At the same time, the third method, which broadcasts 100 MB data and reduces all the data back with tree topology, takes around twice the time of allreduce without chunking input data.

Then we tested again with setting the maximum number of tasks per node up to 8 and the total number of tasks up to 1024 (see Fig. 11). We note that although the scaling trend doesnt change on 3 methods, the execution time of allreduce without chunking input data and allreduce with chunked input data increases.Taking the results on 128 tasks as an example, allreduce

without chunking input data originally uses 9.2 seconds but now it uses 15.2 seconds. Allreduce with chunked input data originally uses 2.7 seconds but now it uses 4 seconds. But broadcast + reduce with tree topology doesnt change much.The execution time only increases from 15.8 seconds to 17.9 seconds. This test shows the limitation of the test environment. In the first allreduce two methods, each task has $log_2 p$ connections in the communication while in the last broadcast + reduce method each task only has 2 connections at maximum. As the number of tasks per node increases, network contention occurs on the first two allreduce methods. In spite this, we still can see allreduce methods have better performance compared with using broadcast + reduce.

### C. K-Means Clustering

The experiments on K-means clustering with two different data sets. One is to cluster 200 million 3D points to 4 thousand clusters (50K points : 1 cluster in average). Another is to cluster 2 million 3D points to 400 thousand clusters (5 points : 1 cluster in average). The former is a normal use scenario while the latter is a special case which uses clustering as classification.

We increased the number of tasks from 32 to 512 as shown in Fig. 12. The speedup in two test cases is considered linear over 32 tasks. With this baseline, the speedup on 512 tasks in clustering 200 million points to 4 thousand centroids is about 491, which is near linear speedup. In the other test, the speedup on 512 tasks is 432, slightly lower than the linear speedup due to the allreduce overhead of the big centroids data.

As explained in previous sections, the implementation of this K-Means Clustering application is both fault tolerant and elastic. The driver can add new tasks with new point data to the allreduce topology for subsequent iterations. Note the accuracy of elastic K-Means Clustering application is not covered in this paper. As a result, we are not going to evaluate elastic K-Means Clustering here.

We evaluated the cost of failure recovery in K-Means Clustering application (see Fig. 13). When some tasks failed, the REEF driver automatically re-launched the failed tasks. We measured the average time of normal iterations and the average iteration time of recovered iterations. Our findings show the time consumption in failure recovery is dominated by task input data loading and the global data re-synchronization. Since both input data per task and global data are not very large, a recovered iteration only takes slightly more time than a normal iteration.

### D. Batch Gradient Descent

In this experiment, we used batch gradient descent to learn splice site recognition data in order to recognize a human acceptor splice site [9]. We ran a Hadoop Map-only job to generate 500 partitions of feature data while taking sample data of 50 million sequences as the input. Each partition created is about 1.5GB after compression. Later the partitions are processed by one task in the BGD application. The global model data is a double array with 11,725,480 dimensions, about 100MB in total.

We tested the weak scaling of the BGD application by running the application with 10 partitions, then with 50, 100, 200, and 500. We measure the performance of each iteration using 3 different allreduce methods. The results are shown in Fig. 14. Allreduce with chunked input data still performs the best, although there is overhead of splitting the data into chunks in each allreduce. The execution time per iteration with 200 partitions shows similar performance as 500 partitions. For the other methods, namely allreduce without chunking input

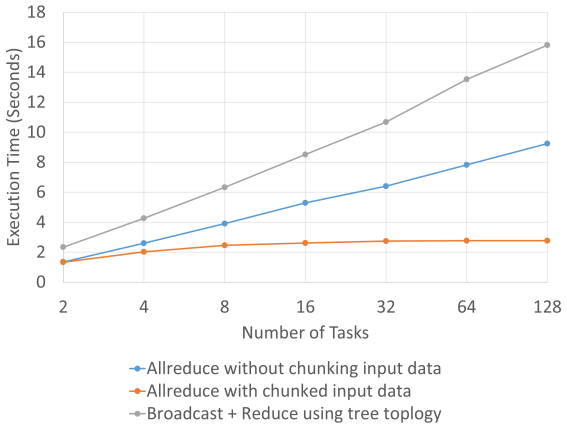


Fig. 10: Allreduce with 100MB data with Each Node One Task Only

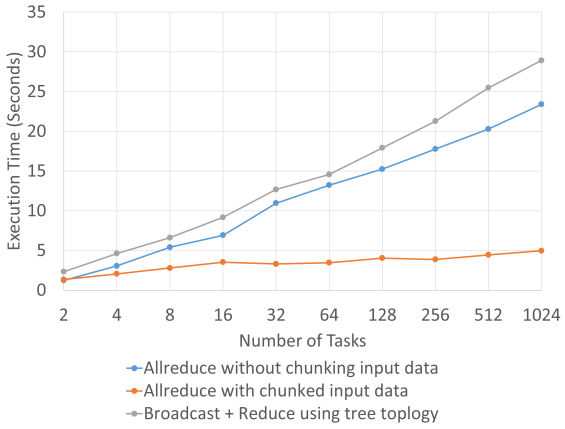


Fig. 11: Allreduce with 100MB data with Each Node up to 8 Tasks

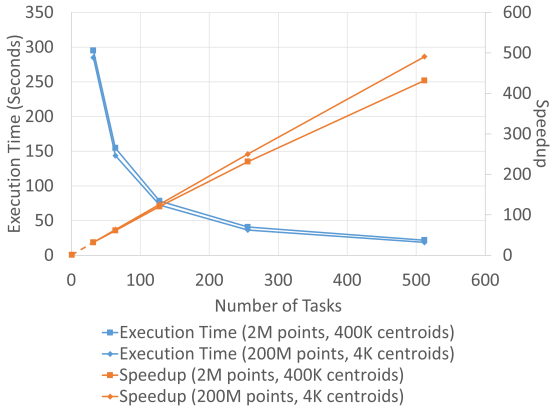


Fig. 12: K-Means Clustering Execution Time per Iteration and Speedup

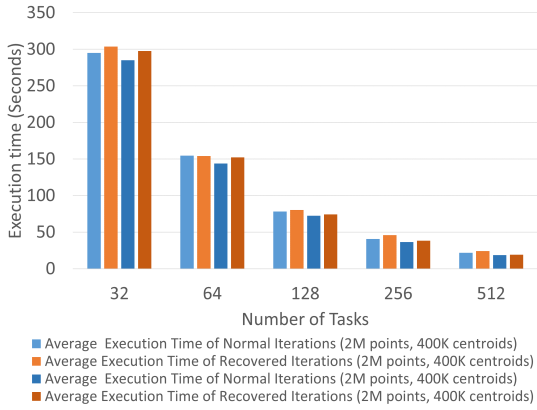


Fig. 13: Average Execution Time Comparison of Normal Iterations and Recovered Iterations on 2M Points, 400K Centroids and 200M Points, 4K Centroids

data and broadcast + reduce using tree topology, both followed the logarithmic scale but are much slower. All these results match with our observations in the allreduce benchmark test.

BGD is an algorithm with elasticity. The training process can be based on any number of partitions of the input data. Here we present the performance of the BGD application in ramp up mode and the accuracy of the model learned in each iteration. We start with 10 partitions and add 14 in each iteration. Finally we wind up with 500 partitions on Iteration 36 and run with that number until Iteration 100. By recording the model trained every 5 iterations and evaluating them on 500 data partitions, we get the result shown in Fig. 15. Tasks are launched sequentially every 4 seconds. In ramp up mode, the first few iterations take longer than the following iterations because the new tasks joined in those iterations have to load data into memory first. This process may take about 200 seconds. In spite of this, ramp up mode still takes hundreds of seconds less than the normal execution mode. For accuracy,

the loss values of the models trained in the first several iterations stay on high values (which means the accuracy is low) and the trend decreases slowly. After all the data is added,

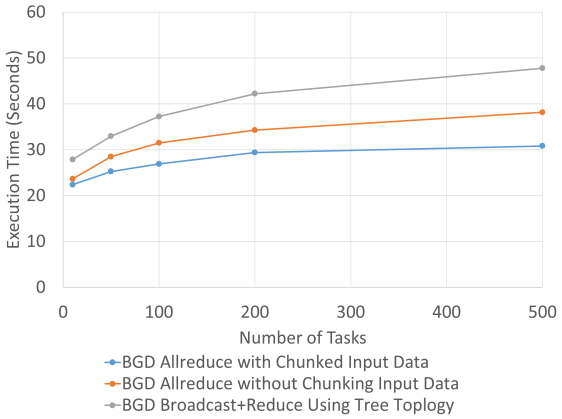


Fig. 14: BGD Weak Scaling

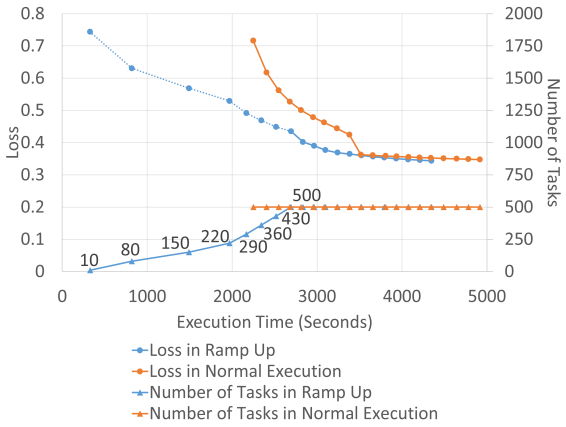


Fig. 15: Ramp up Test v.s. Normal Execution on 500 Tasks

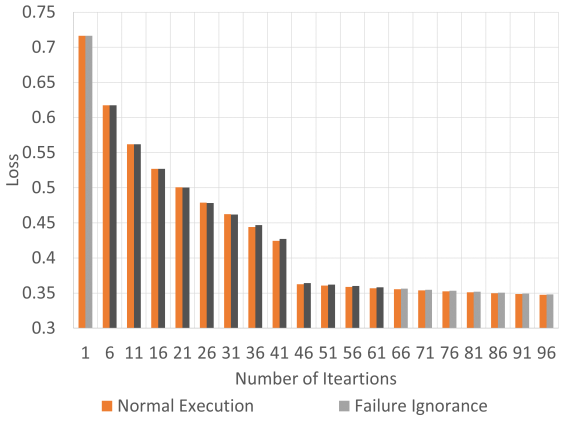


Fig. 16: Normal Execution vs. Failure Ignorance on 500 Tasks

the loss value drops and draws close to the loss value computed in the normal execution.

Fig. 16 shows the results on a failure ignorance test. In this experiment, 400 tasks are killed in Iteration 5 and recovered in Iteration 61. Later we evaluated all the models on 500 input data partitions. The result shows that the model trained in the iterations with 100 of 500 tasks has higher loss values compared with the models trained on the same iteration in the normal execution. When all the tasks are added back, the difference between the loss values on the same iteration in two different execution modes gets progressively smaller, and finally becomes negligible.

## VII. RELATED WORK

Research in collective communication operations for big data processing is gaining attention. Through our earlier research on iterative applications, we have come to the realization that the performance of these collective communication operations is a distinctive feature of data intensive computation.

Some initial work in this area has been done in Twister [7][8] and Spark [9]. Both tools try to improve broadcast operations in iterative MapReduce chains. Further research [10][19] aims to add allgather and allreduce into Hadoop. Later work on Harp [20] attempts to build a complete collective communication layer which is available to be used in MapReduce big data processing tools. Many existing frameworks only try to provide an in-memory communication solution. The performance and related communication topology is not well studied [19], and these work are done either on top of Hadoop directly or Hadoop-like MapReduce tools [7][8][9][10][19]. So all these solutions only support a fixed number of tasks without elasticity. Furthermore, for fault tolerance, they mainly use disk-based check pointing between iterations to ensure computation fault tolerance without considering communication fault tolerance.

## VIII. CONCLUSION

This paper presents an efficient, fault-tolerant, and elastic allreduce framework. It enables users to express iterative applications as iterations of allreduce operations. Our research shows that with a fault-tolerant and elastic allreduce operator, removing failed tasks and adding new ones can be executed transparantly between iterations of the computation.

We improved hypercube topology and made an allreduce topology which can work on any number of tasks. Two different allreduce algorithms are deployed on this topology: allreduce without chunking input data and allreduce with chunked input data through reduce scatter + allgather. Both methods have better performance and are at least twice as fast compared with allreduce through broadcast + reduce using tree topology. Furthermore, allreduce with chunked input data has better performance than allreduce without chunking the input data on allreduce of the large data. The experimental results show that our allreduce framework is highly scalable with almost constant execution time.

We included a mechanism for updating the allreduce topology when removing failed tasks and adding new ones. The cost of updating topology is low, which only involves message exchange between tasks and drivers, especially when adding new tasks, where only $log_2 p$ tasks are required on the driver. Our results also show that the cost of computation recovery from the topology changes is very low. Other than waiting for the start of the new tasks, one only needs to re-synchronize the global data shared between tasks. This process can run without using any disk-based checkpointing, just an additional allreduce operation.

## REFERENCES


[1] C.-T. Chu,et al."Map-Reduce for Machine Learning on Multicore". NIPS, 2006.

[2] Apache Mahout. https://mahout.apache.org/

[3] J. Dean and S. Ghemawat. "MapReduce: Simplified Data Processing on Large Clusters". OSDI, 2004.

[4] J. Ekanayake, H. Li, B. Zhang, T. Gunarathne, S.-H, Bae, J. Qiu, G. Fox. "Twister: A Runtime for iterative MapReduce." Workshop on MapReduce and its Applications, HPDC, 2010.

[5] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. "Spark: Cluster Computing with Working Sets". HotCloud, 2010.

[6] Apache Hadoop. http://hadoop.apache.org

[7] J. Qiu, B. Zhang, "Mammoth Data in the Cloud: Clustering Social Images." In Clouds, Grids and Big Data, IOS Press, 2013.

[8] B. Zhang, J. Qiu. "High Performance Clustering of Social Images in a Map-Collective Programming Model". Poster in proceedings of ACM Symposium On Cloud Computing, 2013.

[9] M. Chowdhury et al. "Managing Data Transfers in Computer Clusters with Orchestra". ACM SIGCOM, 2011.

[10] A. Agarwal et al. "A Reliable Eective Terascale Linear Learning System". Journal of Machine Learning Research. 2014

[11] MPI Forum. "MPI: A Message Passing Interface". In Proceedings of Supercomputing, 1993.

[12] E. Chan, M. Heimlich, A. Purkayastha, and R. Geijn. "Collective communication: theory, practice, and experience". Concurrency and Computation: Practice and Experience (19), 2007.

[13] REEF. http://www.reef-project.org/

[14] F. Cappello et al. "Toward Exascale Resilience: 2014 Update". Supercomputing Frontiers and Innovations. 2014

[15] J. Hursey et al. "The Design and Implementation of Checkpoint/Restart Process Fault Tolerance for Open MPI". Workshop on Dependable Parallel, Distributed and Network-Centric Systems, IPDPS, 2007.

[16] J. MacQueen, "Some Methods for Classification and Analysis of MultiVariate Observations." Berkeley Symp. On Mathematical Statistics and Probability, 1967.

[17] S. Lloyd. "Least Squares Quantization in PCM". IEEE Transactions on Information Theory 28 (2): 129137, 1982.

[18] Big Red II. https://kb.iu.edu/data/bcqt.html

[19] T. Gunarathne, J. Qiu, D. Gannon, "Towards a Collective Layer in the Big Data Stack". The proceedings of the 14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing Conference, 2014.

[20] Harp. http://salsaproj.indiana.edu/harp/index.html