Parallelization and Distribution for Massively Online Analysis

Vladimir Petko

February 18, 2013

1 Introduction

The Large Hadron Collider and electricity networks, news feeds and high-frequency trading - we need to extract useful information out of those data streams. Sheer volume makes storage infeasible and requires special data mining algorithms. The speed of the data often exceeds the computing power of a single machine and requires parallel processing. Massively Online Analysis (MOA) framework allows evaluation and comparison of such data mining algorithms from the evolving data streams and provides a common framework for their implementation[17]. Storm is a free and open source distributed real-time computation system[12]. The goal of the project is to provide MOA/Storm integration to allow staging of the scalable experiments, speed up evaluation of multiple algorithms over the same data and provide scalable implementations of the data stream mining algorithms suitable for the distributed environment. The report is structured as following: *Previous work* covers existing paralellization frameworks used for the machine learning and the algorithms to be paralellized, *Problem Statement* describes the scope of the project, *Trident-MOA Integration* describes the failed attempt to use high-level Storm computation API and the reasons for the failure, *Storm-MOA Integration* describes the successful implementation, *Experimental Results* section presents speed up graphs and performance benchmarks, *Conclusions* section concludes the report.

2 Previous Work

Large scale parallel machine learning systems must be able to adapt quickly to changes in the data and models in order to facilitate rapid prototyping, experimental analysis, and model tuning. To achieve these goals an effective high-level parallel abstraction must hide the challenges of parallel algorithm design, including race conditions, deadlock, state-partitioning, and communication[23]. This section describes several frameworks used for the machine learning and stream processing tasks and their approach to those problems.

Apache Mahout Apache Mahout[2] provides a Hadoop-compatible interface to a collection of the machine learning algorithms. The iterative ML algorithms are modelled as a sequence of Map-Reduce tasks. Implementation of iterative ML algorithms encounters difficulty sharing common data in the Map-Reduce environment - bundling the relevant state information with each training example adds substantially to the size of the training set. The vanilla implementation does not provide any official method of the iterative shared state updates[21]. In Hadoop, each map task runs in its own Java Virtual Machine (JVM), leaving no access to a shared memory space across multiple map tasks running on the same node. The shared data in the Hadoop implementation is distributed using external sources (distributed file system or database) which reduces level of the parallelism and introduces synchronization problems. Iterative Map-Reduce implementations such as Twister use broadcasts of the model parameters to the

mappers[13]. Map-Reduce does not provide facilities for the modelling of the cyclic dependencies found in the iterative algorithms. Hadoop Map-Reduce and Machine Learning algorithms are being fused by the desire to utilize existing data processing pipelines.

GraphLab GraphLab uses a shared-memory implementation to provide efficient framework for the construction of the parallel machine learning algorithms[22]. The data is represented as an undirected graph and algorithm is expressed as a set of update functions - a function which has read-write access to the current vertex and its neighbours. The update function may schedule execution of itself on the current vertex or schedule execution of the neighbours when update criteria is met. Sync functions are used to update global parameters of the algorithm. This framework makes use of the fact that many machine learning algorithms have local dependencies which can be expressed as a graph, e.g. multiplayer perceptron, decision tree. A distributed version implements graph distribution which minimizes the number of edges between parts of the graph assigned to different machines, thus minimizing cross-machine traffic[23]. Unlike other parallel frameworks GraphLab performs asynchronous computation - each update function uses most recent data available. Depending on the scheduler's settings GraphLab can guarantee various levels of sequential consistency/parallelism. Updates can be prioritised to ensure fastest algorithm convergence.

Pregel Google Pregel is a large scale graph execution framework which uses message passing abstraction instead of the shared memory. It implements the Bulk-Synchronous-Parallel (BSP) model which partitions execution into supersteps[24]. The graph is executed in parallel within one super step, and super steps are executed sequentially, e.g. this model is used in JStar[9]. Messages are typically passed between vertexes connected by edges, though a vertex can also acquire the identifier of the unconnected vertex and send a message directly to it. Unlike Graphlab, in Pregel it is possible to dynamically change the graph configuration. A number of open-source frameworks implement the BSP model:

- Apache Hama is a pure BSP (Bulk Synchronous Parallel) computing framework on top of the HDFS (Hadoop Distributed File System)[8] for massive scientific computations such as matrix, graph and network algorithms[7]. It seamlessly integrates with the Hadoop framework and provides a way to solve iterative problems, e.g. implement machine learning algorithms within the existing Hadoop data processing pipeline. The superstep synchronization task is assisted by Zookeeper[4] and Hadoop RPC is used for the communication between workers within the superstep.
- Apache Giraph[15] is a graph computation framework similar to GraphLab. While Apache Hama focuses on generic BSP API - a set of parallel tasks which can exchange messages and synchronize at the barrier, Apache Giraph provides explicit graph manipulation API - the algorithm is expressed

as a set of connected vertexes. During the superstep they perform local mutations and send messages, the graph mutation is performed just before the next superstep, and sent messages are received during next superstep.

- GPS: A Graph Processing System implements a system similar to Pregel the with following differences: graph is dynamically repartitioned between machines to reduce communication overhead, global state updates are performed using a special Master class separating vertex based computation and global state update[28]. GPS uses HDFS to store the graph configuration and superstep checkpoint files.

Storm Framework Storm[12] shares the asynchronous nature of computation with GraphLab and message passing semantics with Pregel. It provides an asynchronous event processing system composed of the following elements:

- *nimbus* service - a central coordinator which performs the task distribution and hosts master code and configuration respository. It updates the Zookeeper cluster with the most current configuration.
- *supervisor* services which perform code and configuration synchronization from Zookeeper cluster and launch worker processes
- *worker* processes that run topologies (jobs) submitted by the users

Figure 1 shows the Storm cluster architecture.

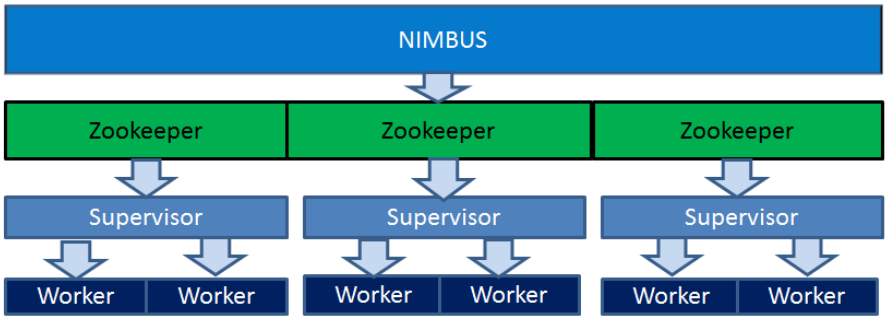

Figure 1: Storm Architecture

The programming model of the Storm framework describes *topologies* - parallel algorithms as

- Spouts - event sources
- Bolts - event consumers and producers
- Streams - tags assigned to the event emitted by spout or bolt

Figure 2 shows topology processing events.The spout emits events tagged with the *stream* label, bolt subscribes to this stream and processes events. The bolt may emit event after processing with the same stream tag, or assign a new one (e.g. *other*). The spout automatically receives acknowledgement message when topology finishes processing of the messages. The acknowledgement mechanism may be circumvented either by emitting new event without specifying which event it is derived from, or by disabling acknowledgement service.

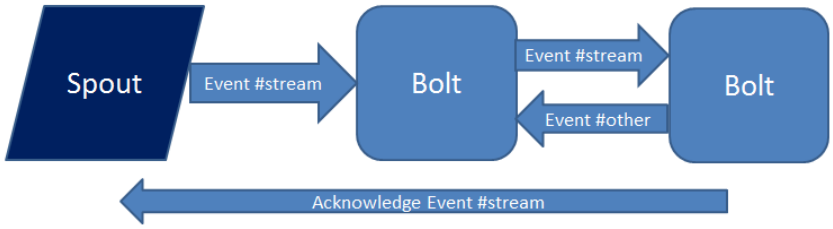

Figure 2: Sample Topology

The topology is distributed between worker processes. Each component - a bolt or a spout - is assigned an arbitrary number of tasks to be distributed between worker processes, by default it is 1. The tasks are executed within the thread pool exclusive for each component - executors - by default the size of this pool is 1. The relationship between components (spouts and bolts), tasks and executors is shown in the Figure 3.

The Trident extension of the Storm framework is a high-level abstraction for the real-time computing which allows to perform stateful stream processing and low latency distributed querying[25]. It defines concepts similar to Pig[16] and Cascading[5] - joins, aggregations, grouping, functions, and filters. In addition Trident defines states backed up by the persistent database storage. The stream-processing work flow is called topology and Storm/Trident takes care of the optimal distribution of its processing elements between hosts of the cluster. Processing Elements can be either spouts - stream sources, or bolts - groups of the stream processing functions. Trident groups stream processing functions to minimize message passing between machines.

Online meta-classifiers: OzaBag and OzaBoost Iterative online meta-classifiers OzaBag and OzaBoost[27] can be easily parallelized:

- The OzaBag implementation in MOA[17] performs training by iterating over members of the ensemble and checking the random Poisson-distributed variable to see whether current example should be used for the training of the ensemble member. Prediction is performed by presenting an example to each member of the ensemble and summing their votes. Both can be viewed as Map-Reduce jobs - an example is mapped to each

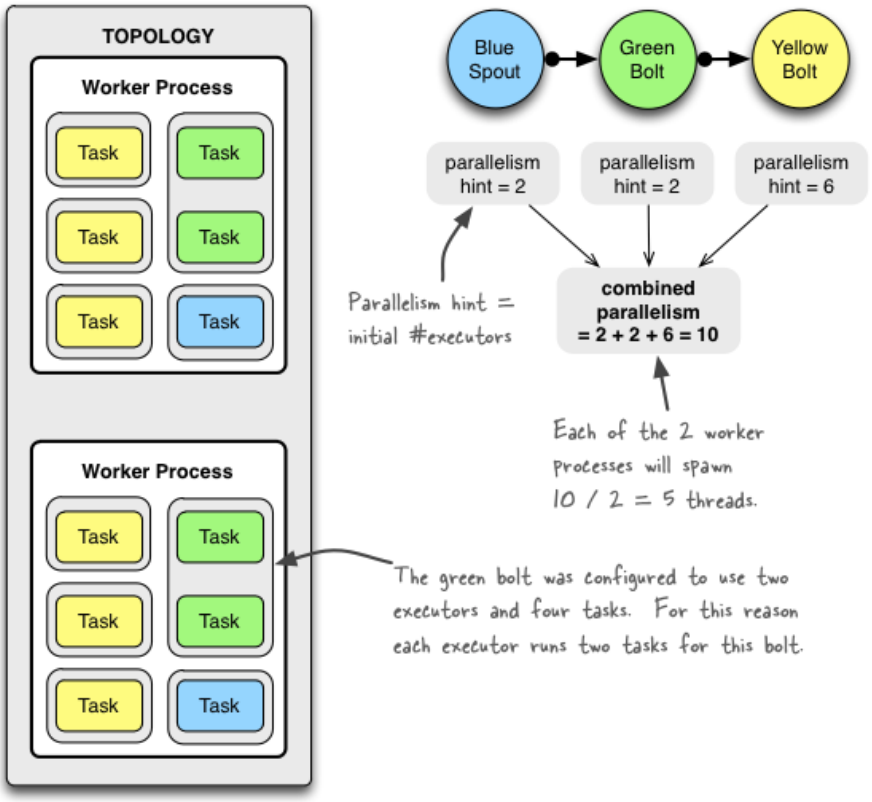

Figure 3: Storm Parallelism[26]. A *Blue* spout is instantiated twice and it is assigned two threads - each task has its own thread. *Green* bolt is instantiated four times - it has two tasks executed in two executor threads - parallelism hint two. *Yellow* bolt is instantiated six times and assigned six executor threads - parallelism hint six.

member of the ensemble. In the prediction case the reduction is performed on the resulting votes.

- The OzaBoost implementation in MOA[17] performs training by iterating over members of the ensemble and checking the random Poisson-distributed variable to see whether current example should be used for the training of the ensemble member. The model parameter λ is iteratively updated and used to calculate ensemble member's weights. Prediction is performed by presenting an example to each member of the ensemble and summing their weighted votes. Due to the presence of the iteratively updated parameter λ it is impossible to perform training in a parallel manner using this algorithm. The parallel computation framework should allow to pipeline it, increasing training throughput at the expense of the latency due to the network transfer and framework overhead. The prediction can be viewed as a Map-Reduce job - an example is mapped to each member of the ensemble. In the prediction case the reduction is performed on the resulting weighted votes.

3 Problem Statement

Stream mining algorithms are iterative in nature[17] and require access to most of the model which can be too large to effectively fit into a single machines memory, which poses a problem for their distribution as it is imperative to minimize inter-machine communication overhead. Ensemble meta-algorithms models are smaller and can be easily replicated between machines, as long as individual members of the ensemble can fit into single machine's memory. This allows to reuse existing sequential stream mining algorithm's implementations as base learners in the ensemble and pass required meta-information in the messages (e.g. OzaBoost requires λ parameter to be passed and OzaBag does not require any inter-classifier communication for training). This project focuses on the implementation of the meta-classifiers OzaBag and OzaBoost using Storm framework and measurement of the performance speed up compared to single threaded MOA implementation of those meta-classifiers.

4 Trident-MOA Integration

Sequential Topology The topology architecture is shown in the Figure 4. The MOA client uses the AMPQ message broker - RabbitMQ to submit training instances to the Storm topology. The Storm topology consists of the AMPQ message spout which retrieves training instances in small batches and a shared state backed up by memcached which uses an aggregation operator to feed them into an existing MOA algorithm. The trained classifier and statistics can be retrieved via remote procedure call query. The evaluation part of the topology consists of the AMPQ spout which retrieves evaluation instances, shared state query which uses trained classifer to make prediction, and a filter which submits

predictions along with the corresponding instances back to the MOA client. This implementation essentially results in a master-slave architecture with single-threaded training of the classifier and multiple hosts performing evaluation. This topology was used as a proof-of-concept of the Storm-Trident integration.

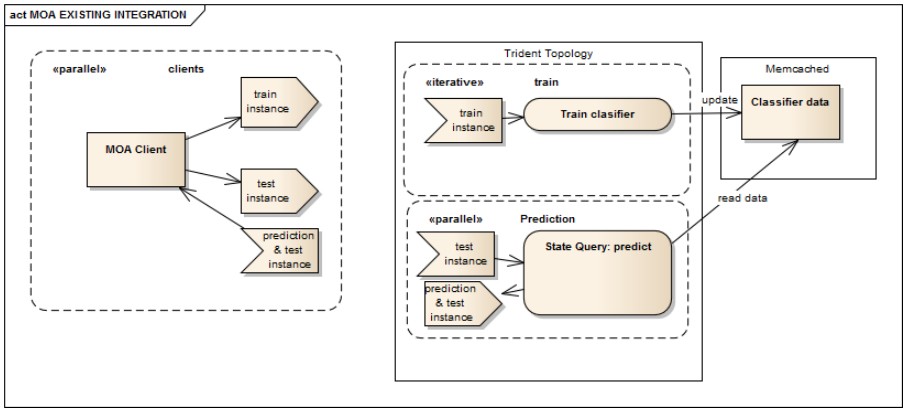

Figure 4: Sequential Topology

OzaBag topology The OzaBag[27] is a simplest distributed MOA algorithm to implement. The storm topology uses architecture described above with a single difference - each classifier in the ensemble uses its own shared state backed up by the separate set of the memcached instances. This allows to accommodate larger ensembles than is possible with the single threaded MOA implementation. The topology architecture is shown in the Figure 5.

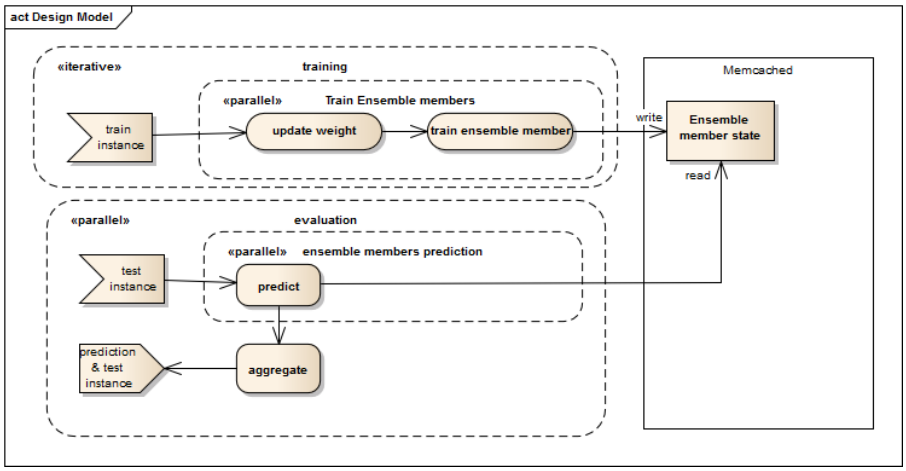

Figure 5: OzaBag Topology

Cluster Configuration Cluster used in the project is built using ML laboratory 8 Intel(R) Core(TM)2 Duo CPU E6850 @ 3.00GHz with 4 GB RAM computers running Linux 2.6.32 and Java 1.6.0.27b7. Storm framework version 0.8.2 is deployed on those machines. The shared state functionality is supported by memcached 1.4.2. RabbitMQ 2.8.4 Advanced Message Queue Protocol message broker cluster is used to communicate with the topologies. The cluster architecture is shown in the Figure 6.

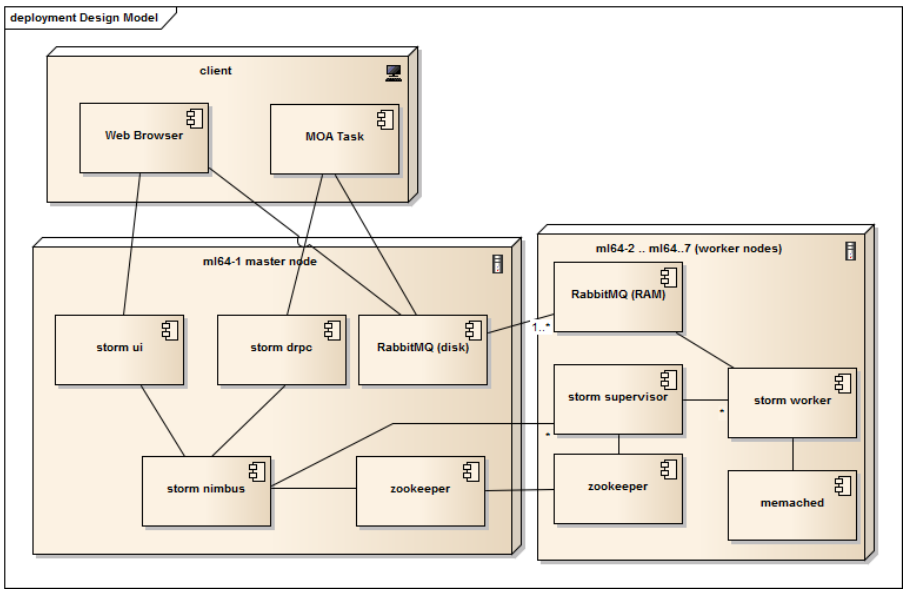

Figure 6: Cluster Setup

Performance benchmarks The speed up benchmarks showed that this implementation is much worse than single-threaded MOA client as shown in Figure 7. Below is the list of the problems which prevented further evaluation:

Stream splitting and merging The instance should be delivered to all classifiers of the ensemble in parallel. The split method is presented in Figure 8.

The call to the null partitioning function *shuffle()* is required to force Trident to avoid putting functions into the same bolt as stream merge operation. Topology generated by Trident using *Stream.each()* contains the number of bolts equal to the number of splits. Thus an ensemble with 1000 members will require 1000 threads on the cluster to execute if each classifier is hosted in a separate function. One of the possible workarounds is to add a grouping field to the event and perform partitioning using this field. This approach was used in the Trident OzaBag implementation. The OzaBoost training implementation

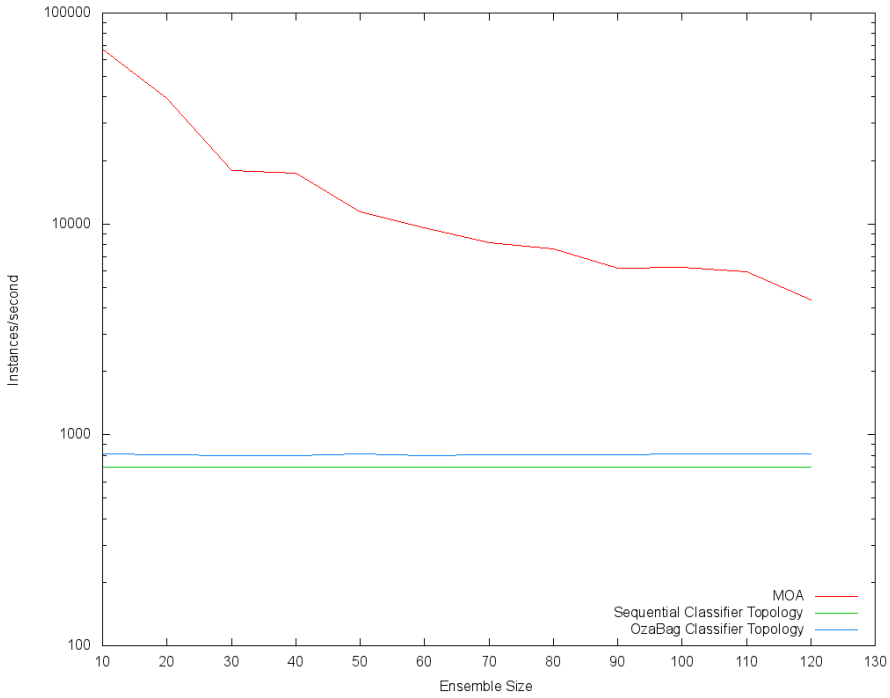

Figure 7: Trident Prediction Performance

```
// Trident filter function
Debug filter = new Debug();
// Trident stream
Stream stream = topology.newStream("stream", spout);
// Split procedure - event emitted by the spout
// will be sent to numSplit functions in parallel.
Stream[] streams = new Stream[numSplits];
for (int i = 0; i < numSplits ; i ++)
        streams[i] = stream
                .each(new Fields("field"), filter)
                .shuffle();
...
// Merge streams back after processing
Stream merged = topology.merge(streams);
```

Figure 8: Stream Splitting and Merging

would have suffered same problem, as it would be a choice of either putting all classifiers in the single bolt, essentially making a sequential implementation, or specifying partitioning operation after each function, multiplying number of bolts.

Combiner implementation Combiner is a function which aggregates computation results of the functions hosted in the single machine to reduce communication overhead. Merge operation in Figure 8 makes combiner implementation impossible, as it requires input from all the functions. The workaround for this problem was not found.

State Implementation Transactional state implemented in Storm is snapshotted for querying for each batch to make sure that each batch is processed using the correct version of the state. Thus if the topology allows processing of several pending batches, there are multiple instances of the classifiers present in the memory. This poses a problem as it limits either the performance or the size of the classifier which can be used. Non-transactional state consistency is not guaranteed between batches - Trident fault-tolerance mechanism may replay old batches out of order which will pose a problem for the algorithms which use time window.

Memached[11] Trident state implementation was initially used, but it was not suited for the storage of the large objects[1] and did not provide persistence support.

Debugging and Tracing problems The current Storm API does not provide Trident tracing functionality - it is impossible to see generated function-bolt assignment and generated streams. A utility to perform dumps of the Trident topologies was developed to visualize generated function-bolt distribution.

5 Storm-MOA Integration

Due to the inability to solve the combiner implementation problem, overhead imposed by stream splitting, necessity to have leaner state implementation and more transparent debugging and tracing it was decided to continue the project without the use of Trident.

Stream splitting Storm API allows to specify number of tasks for each bolt and subscribe the bolt to the stream. Thus to deliver an instance to each classifier it is sufficient to subscribe the classifier bolt to the instance stream and specify a grouping which will deliver instance to each of the bolt's tasks. This allows to avoid the problem with too many bolts present in the topology as classifiers are distributed per task, not per bolt. Though this requires implementation of the bootstrapping code within the tasks hosting the classifier.

Combiner implementation Storm API allows retrieval of the topology context information. It contains assignments of the bolt tasks to the worker processes. Combiner bolt subscribes to the output of the classifier bolts hosted locally using the custom grouping and uses topology context information to find out how many instances it should receive before passing result to the aggregator.

State Implementation The goal of the project is to provide scalable implementation of the meta-classification algorithms. Initially chosen memcached-based state implementation was inadequate due to the excessive memory requirements and lack of persistence. A lot of projects[2][7][15] use HDFS as the backend storage. Its architecture shown in the Figure 9 provides a scalable, hardware-failure tolerant and portable storage platform[8]. While HDFS

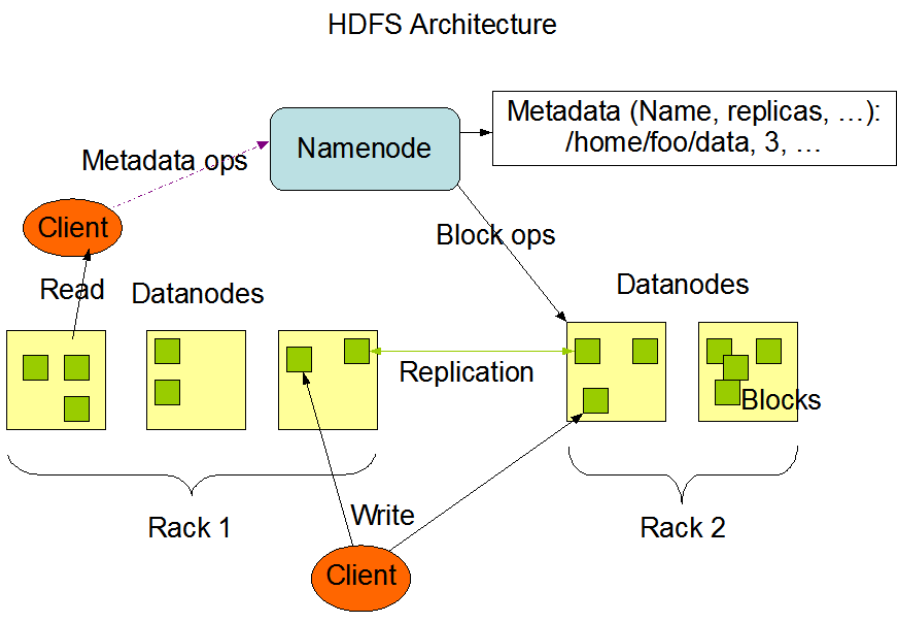

Figure 9: HDFS Architecture[8]

is a proven big-data storage, it does not scale linearly due to the NameNode bottleneck[18]. A NoSQL database such as Apache Cassandra provides linear scalability[14], though it has its limitations - Apache Thrift[3] is used as the communication protocol, and thus each issued database update is limited by the set amount of memory. A Cassandra database used for the classifier storage will require a sophisticated client which will manage transactions of the large objects storage and retrieval. There is successful implementation of such client which has shown performance equal to a comparable HDFS cluster[18]. While Cassandra-based state is preferable due to the better scalability this project

used HDFS due to the simpler implementation.

Instance Generation To streamline benchmarking in the experiments the events were generated within the Storm cluster, as opposed to the reading from the message queue to avoid any impact of the middleware or the client limitations.The project provides Kryo[10] serializers to limit communication overhead, though to simulate instances with significant amount of data, default Java serialization was used in the benchmarks.

Cluster Setup The cluster used in the experiments consisted of 8 Intel(R) Core(TM)2 Duo CPU E6850 @ 3.00GHz with 4 GB RAM. They were running Linux 2.6.32, Java 1.6.0.27b, Storm framework version 0.8.2, Hadoop 1.1.1. The project source code is located at `http://github.com/vpa1977/stormmoa`. The cluster architecture is shown in the Figure 10.

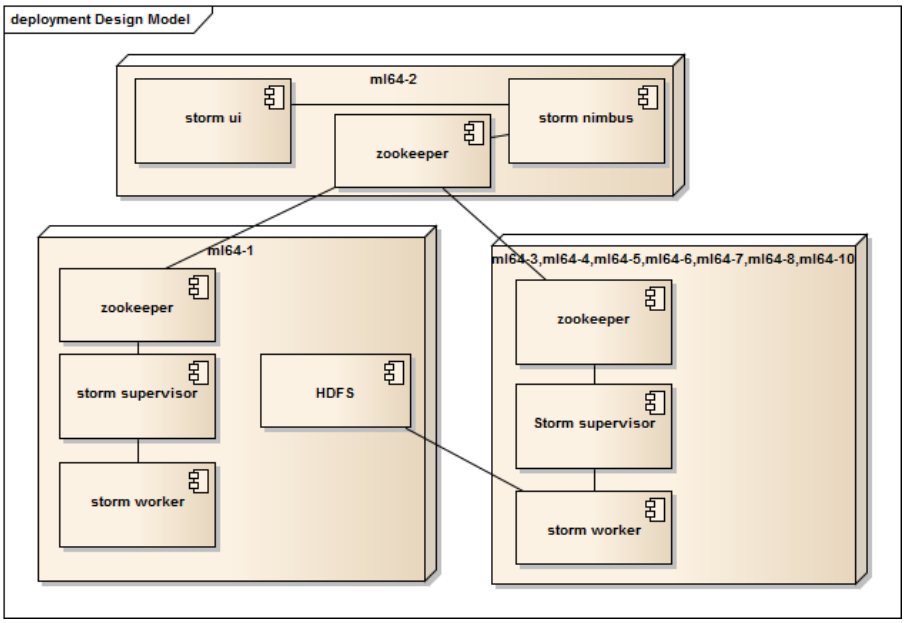

Figure 10: Cluster Setup

Stateless OzaBag implementation The architecture of the topology is shown in the Figure 11. This topology keeps a single instance of each classifier within the bolt's task instance. The spout generates an event and sends to the *broadcast* bolt. The *broadcast* bolt sends it to each worker. Then another *broadcast* bolt hosted inside the worker broadcasts it to the each *classifier* bolt hosted in this worker. Events from the *training* stream are consumed at this point. The *prediction* stream events are updated with the predictions

and sent to the *combiner* bolt. The *combiner* bolt maintains a map of the $event - id => prediction$ and when all predictions from the current worker are accumulated, it sends the result to the *aggregator* bolt and removes the *event-id* from the map. The events are cleared from the *combiner* bolt storage if they are older than the certain threshold. The *aggregator* bolt shares implementation with combiner and uses the same logic. During benchmarking tests the output of the *aggregator* bolt was sent to the *statistics* bolt which counted number of instances processed and output average number of instances per second to the file.

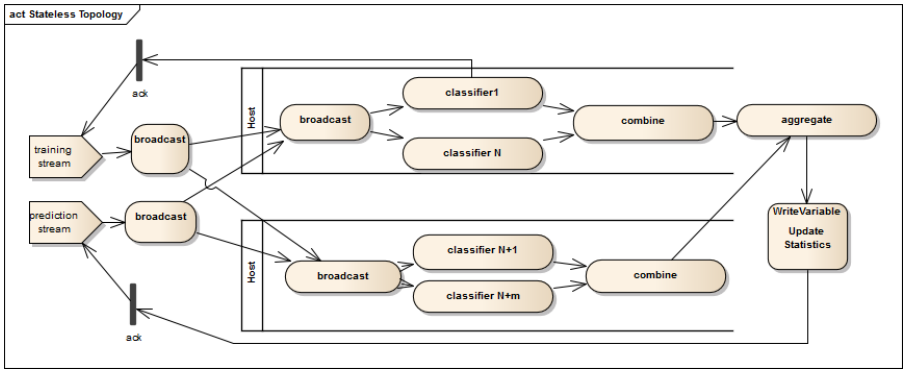

Figure 11: Stateless Topology

This topology does not provide a consistent ensemble state - in case of the failure the ensemble member is retrained from scratch, the training and prediction operations are mutually exclusive and it is not possible to horizontally scale the prediction part of the topology. This topology has minimal overhead, and provides best performance of the bagging implementations. Storm uses Thrift[3] for inter-machine communication. The need to serialize messages exchanged between machines severely increases overhead as shown by Figure 12 which compares prediction throughput of the OzaBag algorithm with 10 Hoeffding trees trained with 100,000 instances. To reduce the message passing overhead, the topology passed batches of 100 instances within one message. The spout had a limit of 10 unacknowledged messages to avoid queue congestion.

Stateful OzaBag implementation The stateful OzaBag is composed of two topologies: *training* and *prediction*. The *training* topology consumes events from the training stream and updates the shared state. The *prediction* topology consumes events from the prediction stream and periodically updates ensemble from the shared state.Prediction topology can be instantiated multiple times to provide linear scalability for prediction. In fact due to the communication overhead for the ensemble of the fixed size it will give better performance - 1.25 times better with 2 topologies running on 4 workers each compared to 1 distributed between 8 workers. The performance does not scale linearly for the

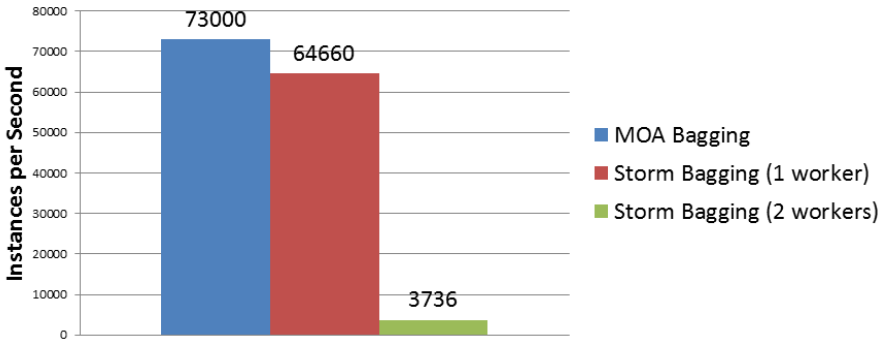

Figure 12: OzaBag prediction performance

same topology due to the increased communication overhead. The overhead

Workers	Instances/Second	Speed up
1	457	1
2	782	1.71116
3	930	2.035011
4	1100	2.407002
5	1300	2.844639
6	1500	3.282276
7	1610	3.522976
8	1760	3.851204

Table 1: Stateless Ozabag Prediction (2000 members)

imposed by task switching when many classifiers are hosted within same worker can be reduced by processing a partition of classifiers within the same task as opposed to hosting 1 classifier per task.

Training topology The architecture of the training topology is shown in the Figure 13. The spout generates an event and attaches a version field to it. It is broadcast to each worker by the *broadcast* bolt. Then another *broadcast* bolt hosted inside worker sends it to each *train classifier partition* bolt hosted in this worker.The bolt trains its partition using the provided instances and if a message contained save state flag, sends an asynchronous message to the *persist* bolt hosted within same worker. The last part is important as at this point the partition of classifiers is copied and serialization of potentially megabytes of data should be avoided. The *persist* bolt saves the data and sends confirmation message to the singleton *update version* bolt. The latter counts responses for the last provided version, and when all *persist* bolt responded - it saves a new version identifier in the persistent storage.

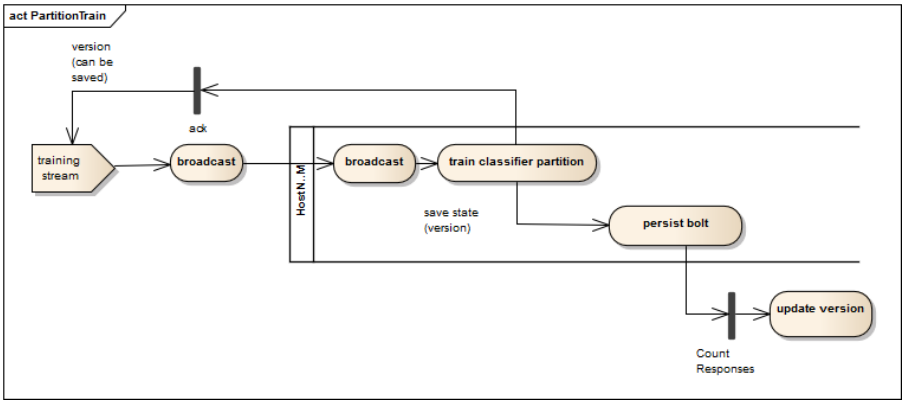

Figure 13: Stateful OzaBag Training

Prediction Topology The architecture of the prediction topology is shown in the Figure 14. The prediction topology has the following differences compared to Stateless OzaBag:

- The spout reads the current ensemble version from persistent storage and sends message with it to the *cache update* bolts. It does not produce evaluation stream messages until all *cache update* bolts confirmed that they have loaded their corresponding partitions. It then updates the *prediction version* of the ensemble in the persistent storage.
- The spout attaches version information to each sent instance and prediction bolts read get corresponding versions from the shared cache.
- The spout periodically checks ensemble version and sends update message to *cache update* bolts.
- The *cache update* bolts make sure that they keep at most two versions of the classifier partition - the one currently used and the one which is being loaded.
- An external clean-up process periodically reads *prediction version* and deletes all previous ensemble versions from persistent storage.

Stateful OzaBoost implementation Stateful OzaBoost implementation is also composed from *prediction* and *training* topologies. The *prediction* topology architecture is essentially the same and differs only in evaluation bolt implementation. The *training* topology is shown in the Figure 15. The message sent by the *training* spout is sent to the first classifier partition, processed, sent to the next partition hosted within the same worker, if the current worker is exhausted it is sent to the next worker until all partitions are trained on the current example. Unlike bagging each message contains only one example.

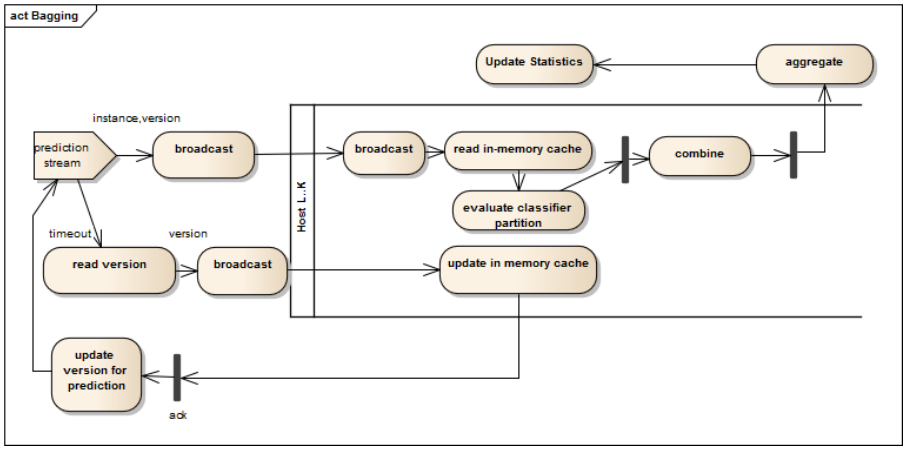

Figure 14: Stateful OzaBag Prediction

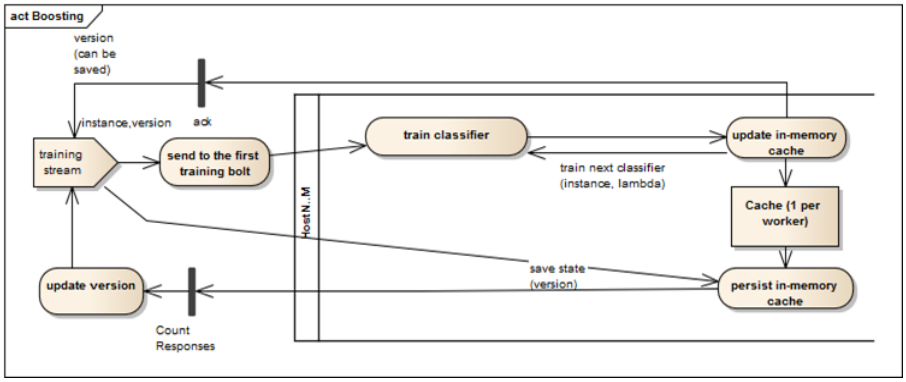

Figure 15: OzaBoost Training Topology

6 Experimental Results

All performance benchmarks were taken using the hardware configuration described in the Cluster Setup section using MOA implementation of Hoeffding tree classifiers as a base learner with the following options: *trees.HoeffdingTree -m 10000000 -e 10000*. It limited memory occupied by single tree to 10 million bytes and the tree checked its conformance to this criteria every 10,000 instances trained. The OzaBoost implementation used pure boost option set to false, meaning that each classifier had chance to train on instances regardless of the λ value. The instances per second measurements were taken as a 10 minutes averages. Each worker was assigned 1 executor per each bolt with the single exception of the classifier bolt which was assigned 2 executors per worker.

Ensemble Size The test was performed with stateless OzaBag topology. The dependency between performance and ensemble size is shown in Figure 16. Communication overhead makes Storm framework-based stream classification infeasible with the used hardware setup and topology until the ensemble size reaches more than 200 elements. For the small ensembles the performance gain can be obtained by using more efficient serialization - for instance swiching to Kryo[10] serialization for the weka.core.DenseInstance class increased prediction throughput of 10 classifier stateless topology to 54600 instances per second resulting in a 14x increase in speed and approaching single core result with 2 threads (64000). Tasks with more computation, e.g. the same topology with 1000 members showed 1.1x speedup compared to Java serialization. The speedup obtained from Kryo serialization compared to Java serialization is shown in Figure 17

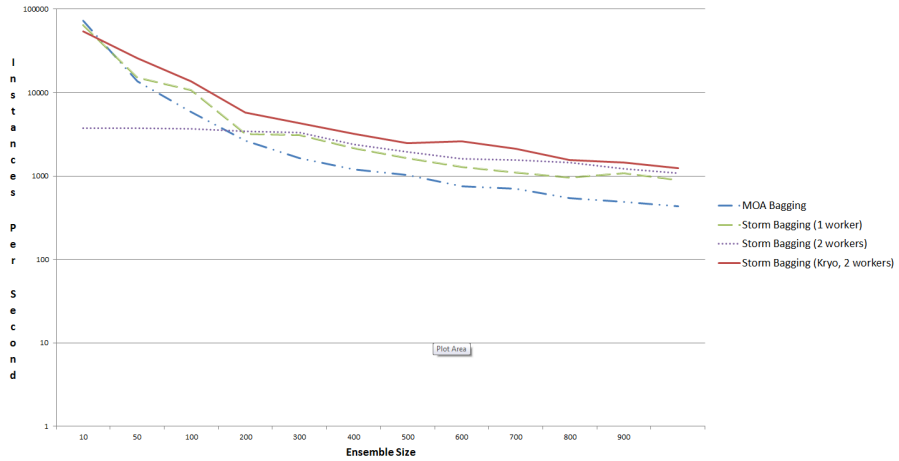

Figure 16: Prediction Ensemble Size

Training Performance The training performance is shown in Figures 18, 20, 19. The OzaBoost topology training throughput is higher due to the fact that it

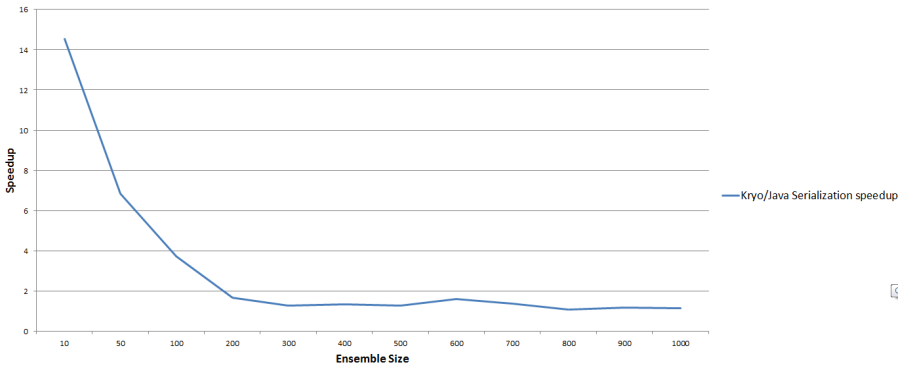

Figure 17: Kryo vs. Java serialization speedup

does not produce spikes of the messages within workers - a message emitted by the spout travels between workers when OzaBag topology causes *broadcast* bolt to emit copies of original message. Though it should be noticed that latency of the OzaBoost training topology is proportional to the ensemble size, as opposed to constant latency of the OzaBag training topology. The speed up is reduced as more machines are added to the ensemble by the HDFS bottleneck (persistent topologies show a reduction in speed up as opposed to stateless bagging). The OzaBoost training can only be performed when partitioning classifiers - one classifier per task proved to greatly increase communication overhead resulting in 2 instances per second training performance with 6 workers and 2000 classifiers. An optimization would be to calculate whether an instance should be sent to the classifier in the grouping function, thus reducing the communication overhead.

Prediction performance The prediction performance is shown in Figures 21, 23, 22. The main limitation of the prediction performance is the communication overhead - the speed up flattens out with an increase in the number of machines for stateless bagging and the same behaviour would probably be observed for persistent topologies if it was possible to run them on more machines. The main contributor is the *weka.core.Instance* class which is serialized for transmission. Using Kryo[10] serialization and stripping header information from the instance it is possible to increase performance, but the speed up ratio will not be greatly increased as shown in Figure 17, e.g. stateless OzaBag ensemble with 1000 members can process 3400 tuples with 6 workers, and 1010 tuples on one worker, giving a 3.4x speed up as opposed to a 3.3x speed up with the Java serialization.

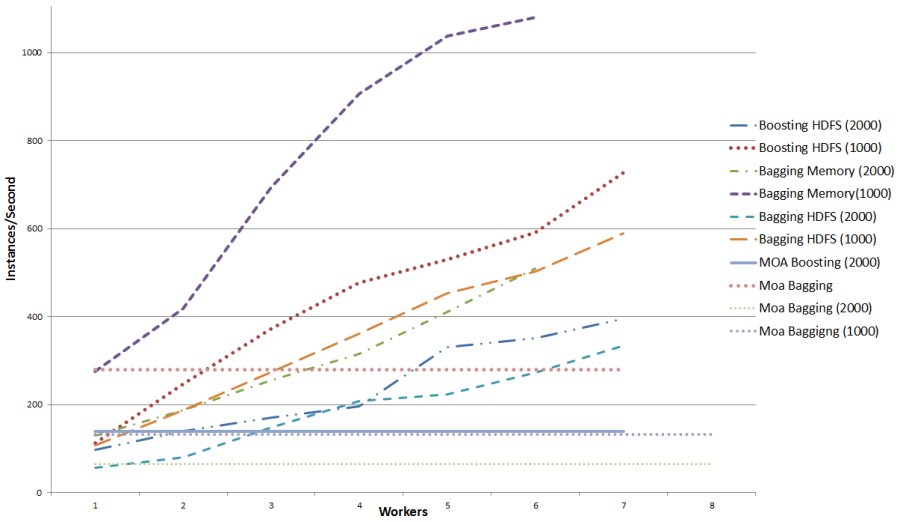

Figure 18: Training Performance

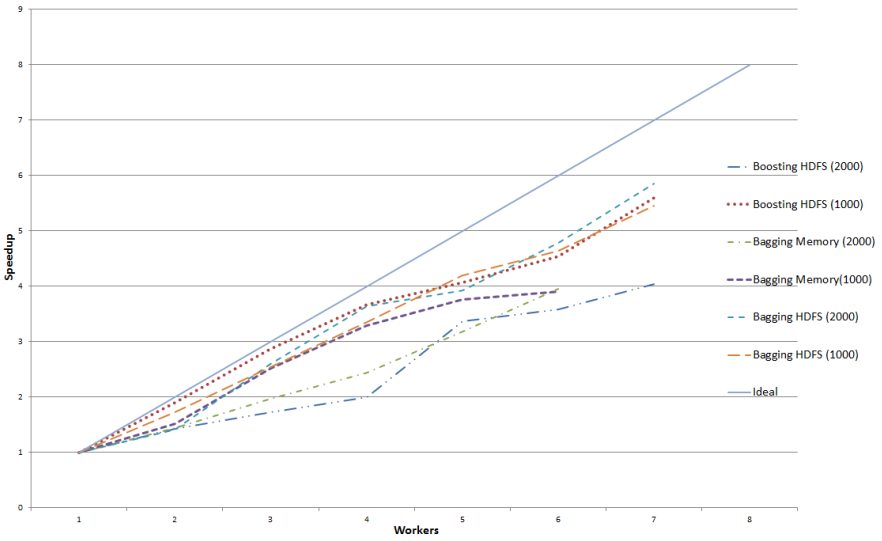

Figure 19: Training Speed up

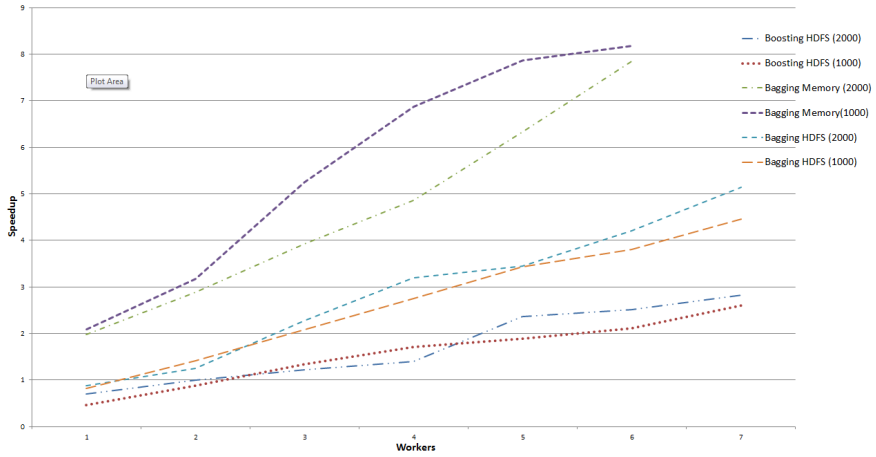

Figure 20: Training Speed up Compared To MOA

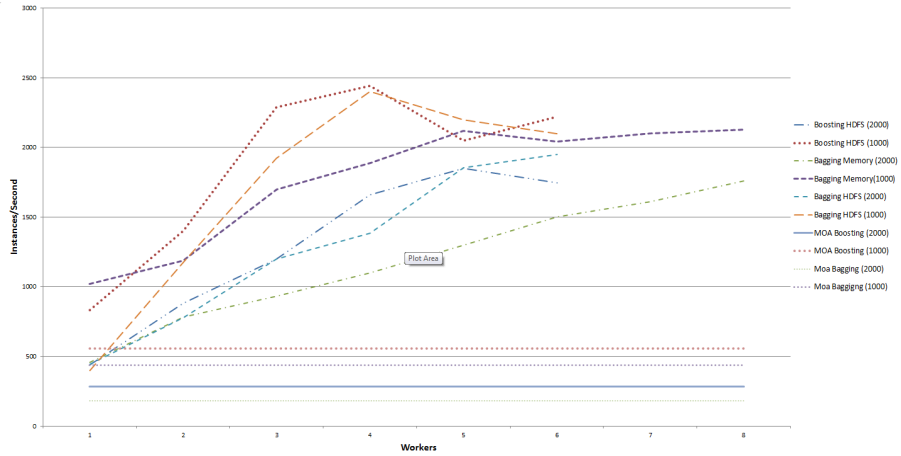

Figure 21: Prediction Performance

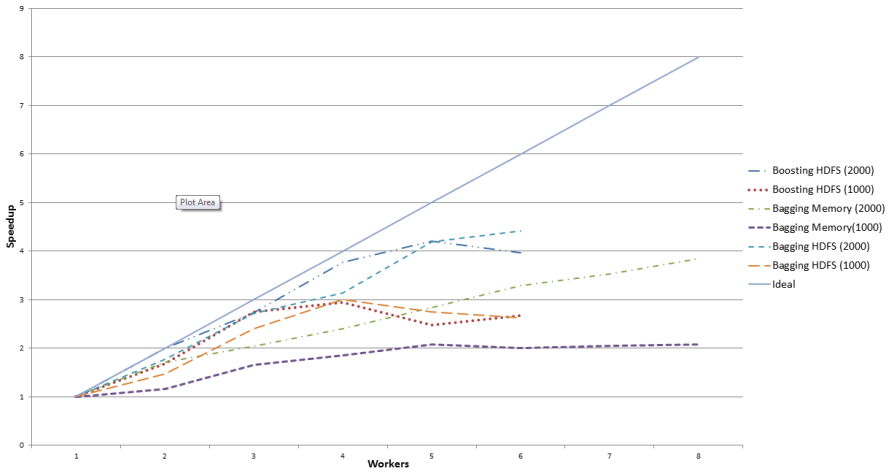

Figure 22: Prediction Speed up

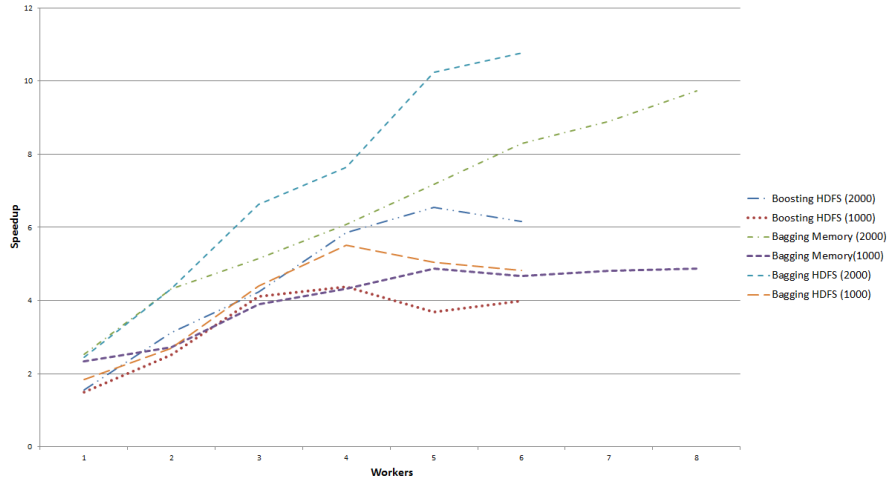

Figure 23: Prediction Speed up Compared To MOA

7 Future Work

Optimizations Update grouping functions to avoid communication overhead - only send instances to the classifier partitions if any of the classifiers are going to use it. Implement worker-local instance cache to avoid inclusion of the immutable test or train instances into the messages.

Computation Redundancy At the moment topologies contain only one instance of the ensemble members, which makes them sensitive to struggler machines. Duplication of classifiers and updated *combiner* bolts will allow to handle struggler problem.

Cassandra State Implement Cassandra State which is able to hold large objects. Current Cassandra State implementation is limited by the maximum column entry size configured in Cassandra configuration.

GraphLab Shared memory model of GraphLab is well suited for the implementation of machine learning algorithms[23]. Internally it is using Message Passing Interface(MPI)[19] to facilitate inter-process communication[6]. GraphLab / OpenMPI[20] cluster may prove to be a more robust solution for the stream classification problems due to the shared memory abstraction implemented in the MPI middleware.

Clustering Add support for the online clustering algorithms, as OzaBoost training topology proved that benefit from distribution exceeds performance loss from the network overhead.

8 Conclusions

Storm framework can be used to solve stream classification problems in a distributed manner. This project provides topology implementing meta-classifier which uses existing MOA classifiers as ensemble members.Configurations with high computation to communication ratios (large ensembles, large models) perform significantly better in distributed setting than single-threaded implementations and vice versa. To facilitate it the ensembles should be reasonably partitioned as opposed to hosting one classifier per task. Efficient serialization such as Kryo[10] will reduce communication overhead, though it will still be present and prevent linear scalability of multi-machine clusters. Efficient shared state is the key factor limiting successful parallel stream mining implementations due to the size of the model data which has to be shared - a 2000 members ensemble limited at 10Mb per member is 20Gb regularly written and read from the shared storage. Tasks should be grouped to minimize inter-machine communication, performance will benefit from dynamic rebalancing based on the inter-task communication.

Bibliography

[1] 15.6.5 memcached faq. http://docs.oracle.com/cd/E17952_01/refman-5.0-en/ha-memcached-faq.html#qandaitem-16-6-5-1-3.

[2] Apache mahout: Scalable machine learning and data mining. http://mahout.apache.org/.

[3] Apache thrift. http://thrift.apache.org/.

[4] Apache zookeeper. http://zookeeper.apache.org/.

[5] Cascading — application platform for enterprise big data. http://www.cascading.org/.

[6] Graphlab - the software. http://graphlab.org/home/the-software/.

[7] Hama - a bulk synchronous parallel computing framework on top of hadoop. http://hama.apache.org/.

[8] Hdfs architecture. http://hadoop.apache.org/docs/stable/hdfs_design.html.

[9] Jstar research group. http://www.cs.waikato.ac.nz/research/jstar/.

[10] Kryo -fast, efficient java serialization and cloning. http://code.google.com/p/kryo/.

[11] memached - a distributed memory object caching system. http://memcached.org/.

[12] Storm, distributed and fault-tolerant realtime computation. http://storm-project.net/.

[13] Twister: Iterative mapreduce. http://www.iterativemapreduce.org/userguide.html#An_Iterative_MapReduce_Application_using_Twister.

[14] Welcome to apache cassandra. http://cassandra.apache.org/.

[15] Welcome to apache giraph. http://giraph.apache.org/.

[16] Welcome to apache pig! https://pig.apache.org/.

[17] Albert Bifet, Geoff Holmes, Bernhard Pfahringer, Philipp Kranen, Hardy Kremer, Timm Jansen, and Thomas Seidl. Moa: Massive online analysis, a framework for stream classification and clustering. *Journal of Machine Learning Research - Proceedings Track*, pages 44–50, 2010.

[18] DataStax Corporation. http://www.datastax.com/wp-content/uploads/2012/09/WP-DataStax-HDFSvsCFS.pdf.

[19] The MPI Forum. Mpi: A message passing interface, 1993.

[20] Edgar Gabriel, Graham E. Fagg, George Bosilca, Thara Angskun, Jack J. Dongarra, Jeffrey M. Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, Ralph H. Castain, David J. Daniel, Richard L. Graham, and Timothy S. Woodall. Open MPI: Goals, concept, and design of a next generation MPI implementation. In *Proceedings, 11th European PVM/MPI Users' Group Meeting*, pages 97–104, Budapest, Hungary, September 2004.

[21] Dan Gillick, Arlo Faria, and John Denero. Mapreduce: Distributed computing for machine learning, 2006.

[22] Yucheng Low, Joseph Gonzalez, Aapo Kyrola, Danny Bickson, Carlos Guestrin, and Joseph M. Hellerstein. Graphlab: A new framework for parallel machine learning. *CoRR*, abs/1006.4990, 2010.

[23] Yucheng Low, Joseph Gonzalez, Aapo Kyrola, Danny Bickson, Carlos Guestrin, and Joseph M. Hellerstein. Distributed graphlab: A framework for machine learning in the cloud. *CoRR*, abs/1204.6078, 2012.

[24] Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, SIGMOD '10, pages 135–146, New York, NY, USA, 2010. ACM.

[25] Nathan Marz. Trident tutorial - nathanmarz/storm wiki. https://github.com/nathanmarz/storm/wiki/Trident-tutorial.

[26] Michael Noll. Storm example of a running topology. http://www.michael-noll.com/blog/uploads/Storm_example_of_a_running_topology.png.

[27] N. Oza and S. Russell. Online bagging and boosting. In *Artificial Intelligence and Statistics 2001*, pages 105–112. Morgan Kaufmann, 2001.

[28] Semih Salihoglu and Jennifer Widom. Gps: A graph processing system. Technical report.