On the Performance of Distributed Data Clustering Algorithms in File and Streaming Processing Systems

Kathleen Ericson and Shrideep Pallickara

Computer Science Department
Colorado State University
Fort Collins, CO USA
{ericson,shrideep}@cs.colostate.edu

Abstract— There is often a need to cluster voluminous amounts of data. Such clustering has application in fields such as pattern recognition, data mining, bioinformatics, and recommendation systems. Here we evaluate the performance of 4 clustering algorithms viz. K-means, Fuzzy k-means, Dirichlet, and Latent Dirichlet Allocation within two different cloud runtimes: Hadoop and Granules. Our benchmarks use identical clustering code with both Hadoop and Granules. The difference between these implementations stem from how the Hadoop and Granules runtimes (1) support and manage the lifecycle of individual computations, and (2) how they orchestrate exchange of data between different stages of the computational pipeline during successive iterations of the clustering algorithm. We also include an analysis of our results for each of these clustering algorithms in a distributed setting.

Keywords- Machine Learning; Distributed Stream Processing; Hadoop; Mahout; Clustering; Granules

I. INTRODUCTION

There is often a need to cluster voluminous amounts of data. Data clustering algorithms are an *unsupervised* machine learning technique that facilitates the creation of *clusters*, which allow us to group similar items (also called observations) together so that these clusters are similar in some quantifiable sense. Clustering has broad applications in areas such as data mining, recommendation systems, pattern recognition, identification of abnormal cell clusters for cancer detections, and bioinformatics among others. Clustering algorithms have certain unique characteristics. First, the algorithms can involve multiple *rounds* (also called *iterations*) of execution, where the output of the previous round is the input to the subsequent round. Second, the number of iterations of the algorithm is determined by the *convergence* characteristics of the algorithm. This convergence is generally based on distance measures (in n-dimensional space) and also the movement of cluster centers in successive iterations of the algorithm. To account for cases where convergence may not occur for a very large number of iterations, it is also possible to specify an upper bound on the number of iterations. Finally, the algorithm may operate on n-dimensional data and will cluster along these dimensions. Beyond the first iteration the progress of the clustering computation depends on (1) the state that it has built up in previous iteration (2) the initial set of data points that it holds, and (3) the adjustments to the cluster centers that it receives from the previous iteration.

As data volumes increase, it quickly becomes untenable to perform this clustering over a single machine. One challenge in implementing distributed clustering algorithms is that it is possible that an algorithm will get stuck in local optima, never finding the optimal solution. Attempting to converge on an optimal solution can be even more difficult when data is distributed, where no single node is fully aware of all data points.

To facilitate distributed processing one approach is to use the MapReduce [1] framework. Here, a large input dataset is sliced into smaller datasets, each of which is then operated upon by a different computation – these separate computations are the *Mappers*. The results of the processing are then fed into a *Reducer* (or a set of reducers) which will account for boundary conditions and adjust the cluster centers accordingly; it is possible that a cluster center residing in one *Map* may have points within its purview that is part of the input data to another Map function. The outputs from the reducer are then fed into the appropriate mappers to begin the next round of processing.

Mahout [2] is a library that implements several clustering and classification algorithms for machine learning which have been modified to fit the Map-Reduce [1] model. The Mahout implementations have been deployed within Apache Hadoop [3] – a MapReduce based cloud runtime. While Mahout has been designed to work specifically with Hadoop, there is nothing to preclude using the Mahout library within another processing system which supports the MapReduce paradigm.

We have compared the efficiency of orchestrating these distributed executions of the clustering algorithms within our distributed stream processing system, Granules [4, 5]. Our benchmarks compare the *same* Mahout code running inside Granules and Hadoop. The only difference between these implementations stem from how the runtimes (1) support and manage the lifecycle of individual computations, and (2) orchestrate exchange of data between different stages of the computational pipeline during successive iterations of the clustering algorithm.

We chose Hadoop and Granules for this comparison as they are representative of file processing and stream processing systems respectively. As both support the MapReduce framework, we can use the Mahout codebase without modifications in both runtimes. With the clustering algorithms identical and unmodified, the only differences in computation speed should be a result of the lifecycle support for individual computations and the underlying communications framework.

In this work we have implemented 4 clustering algorithms viz. K-means, Fuzzy k-means, Dirichlet and Latent Dirichlet Allocation within Granules. These algorithms are representative of two broad classes of clustering algorithms: (1) *discriminative*, where we are making decisions if a point belongs in a predefined set of clusters (k-means, fuzzy k-means) and (2) *generative*, were a model is tweaked to fit the data and we can even generate the data the model has been fit to using the models parameters (Dirichlet and Latent Dirichlet Allocation). We believe that these algorithms are also a good example of performance improvements that can be accrued by moving away from execute-once and stateless semantics in traditional MapReduce implementations such as Hadoop.

The remainder of this paper is ordered as follows: In section II, we discuss Hadoop and Granules. In section III, we discuss clustering in a distributed setting, describing the code modifications needed when switching from the Hadoop paradigm to Granules. We then review our experimental setup in section IV, and in section V we review the dataset. Section VI-IX discusses the clustering algorithms we have implemented, while section X reviews related work. We then report our conclusions and describe future work in section XI.

II. BACKGROUND

A. Hadoop

Hadoop is a Java-based cloud computing runtime which supports the Map-Reduce paradigm. Hadoop has execute-once semantics, meaning that with iterative tasks all state information needs to be written to file and then read back in for every step of the computation.

Hadoop is open-source, and widely used for Map-Reduce computations. Mahout has been built to run on top of Hadoop and the Hadoop Distributed File System (HDFS) [6]. HDFS is an implementation of the Google File System where a large file is broken into fixed size chunks each of which is then replicated. When processing the data, the runtime pushes computations to the machines where these blocks were staged to maximize data locality for faster executions.

When running Mahout through Hadoop on top of HDFS, it is able to take advantage of data locality, and read from files that are stored on the local machine. This can cut down on the amount of time it takes to load data from file drastically.

B. Granules

Granules [4, 5] is a lightweight distributed stream processing system (DSPS) and is designed to orchestrate a large number of computations on a set of available machines. The runtime is designed specifically to support processing of data streams. Granules supports two of the most dominant models for cloud computing: MapReduce and dataflow graphs [7]. In Granules individual computations have a finite state machine associated with them. Computations change state depending on the availability of data on any of their input datasets or as a result of external triggers. When the processing is complete, computations become dormant awaiting data on any of their input datasets.

In Granules, computations specify a scheduling strategy, which in turn govern their lifetimes. Computations specify their scheduling strategy along three dimensions: counts, data driven and periodicity. The counts axis specifies limits on the number of times a computation task needs to be executed. The data driven axis specifies that a computation task needs to be scheduled for execution whenever data is available on any one of its constituent datasets, which could be streams or files. The periodicity axis specifies that computations be scheduled for execution at predefined intervals. One can also specify a custom scheduling strategy that is a combination along these three dimensions; for example, limit a computation to be executed 500 times either when data is available or at regular intervals. A computation can change its scheduling strategy during execution, and Granules enforces the newly established scheduling strategy during the next round of execution. Computations in Granules build state over successive rounds of execution. Though the typical CPU burst time for computations during a given execution is short (seconds to a few minutes), these computations can be long-running with computations toggling between activations and dormancy. Domains that Granules has been deployed include handwriting recognition, Brain Computer Interfaces, and epidemiological simulations.

In our experiments [8] with Brain Computer Interfaces we were able to process electroencephalograms (EEG) signals generated by electrodes placed on a user's scalp in real time. Running concurrently over 10 machines, Granules was able to classify EEG signals generated at the rate of once every 250 milliseconds from 150 concurrent users in less than 70 milliseconds. Specifically, we were able to classify signals at the rate of 1 GB of EEG data every 83 seconds and about 1 TB every 23 hours.

III. CLUSTERING IN A DISTRIBUTED SETTING

Clustering is a machine learning algorithm in which the program is responsible for discovering commonalities across voluminous datasets, and finding appropriate groups to put, or *cluster*, all incoming data. Clustering is a very useful tool in unsupervised data mining and can help uncover relationships between data points which are not otherwise noticed. The classic example of this is the retailer who noticed that diapers and beer are often bought together. In our examples, we are not working with retailer information, but instead working to classify news articles under various topics.

An important part of determining clusters is defining how distance will be measured. Mahout includes definitions of multiple types of distance measurements. In this paper we use the Euclidean distance measure to determine distances between points and cluster centers across all our tests as done in [2].

A. Clustering using Hadoop

Hadoop computations have execute-once semantics and are stateless. Clustering computations expressed in Hadoop need to account for these execute-once and statelessness constraints. At the end of an iteration, every computation must also store the state such that the subsequent iterations can retrieve this information as part of its initialization. A new computation must be launched for each round of execution, and this computation must reconstruct state that it had saved in a previous iteration from disk, typically using HDFS. Often, the original data splits also need to be loaded into the computation. In the case of clustering algorithms which often have multiple, successive rounds of execution this can lead to overheads and increased execution times.

B. Clustering Using Granules

Depending on their specified scheduling strategy Granules computations stay dormant when conditions for their execution have not been satisfied. Computations are activated from dormancy once data is available on one or more of their input datasets. The activation overhead for computations once data is available for processing is in the order of 700 microseconds. Computations in Granules can have multiple rounds of execution and the runtime manages their lifecycles. Individual computations also retain state across these multiple rounds of execution.

In Granules computations can also be expressed as directed graphs that can have cycles in them. When a computation pipeline comprising multiple stages have been set up, it is possible for portions of a set of computations (spanning multiple machines) to have feedback loops in them. The execution breaks out of these feedback loops once the necessary conditions for progress have been satisfied.

Granules allows computations to enter a dormant state between rounds of execution. Due to this ability, running an iterative Map-Reduce application – such as the machine learning algorithms within the Mahout library – within Granules should be more efficient than in a runtime that requires all data to be written to file between rounds of execution.

In our setting involving Granules, each Mapper works with a subset of the original dataset. The Mapper is then responsible for clustering these points throughout the lifetime of the algorithm. For every iteration, the Mapper loops through the points it is responsible for and aggregates all cluster information before sending this data on to the reducer. The reducer is activated when it receives outputs from individual mappers. Once the reducer has received inputs from all mappers, it is able to determine global adjustments to the clusters and send this information back to the mappers to start the next round of clustering. Implementing these distributed clustering algorithms as Granules computations have a few advantages that could translate into faster execution times. Computations can gain from:

(1) Not having to reinitialize state from the disk
(2) Streaming results between intermediate stages of a computation pipeline rather than having to perform disk I/O.
(3) Fast activation of dormant computations as data streams continue to become available.

C. Code Modifications

In this work, we attempted to run Mahout with the Granules DSPS instead of Hadoop. To do so, we needed to modify the drivers for the clustering algorithm as well as some semantic changes to the map and reduce code. The actual clustering algorithms were not touched at all, and it is important to keep in mind that the bulk of our code modification, in the drivers, would generally be modified for something as small as a change in the type of data being clustered.

The I/O format of the code is similarly untouched. In our Granules runs, we kept the HDFS backend for initial loading of points and clusters – while we cannot take advantage of rack-locality in Granules, this did enable us to keep the runtimes comparable. The real changes were made to slightly tweak the map and reduce code, to fit the different programming paradigm of Granules. Hadoop demands run-once semantics – the map and reduce code is called for every line of data that is read by Hadoop. In Granules, computations can retain state during successive rounds of execution and multiple lines of data can be processed at a time.

Both Hadoop and Granules use different strategies to move data between different stages of a computation pipeline. In the case of Hadoop this involves disk I/O and polling to determine if the data is available within HDFS. In the case of Granules, data is streamed between the different stages and computations are activated from their dormancy when such data is available.

D. Fault Tolerance

Hadoop uses a *pull based* scheme wherein the outputs stored by individual mappers, on their local disks, are pulled by the appropriate reducer. While intermediate files produced by the mappers are not managed by HDFS, the outputs generated by the reducer are. Along with the original split of the data points, the reducer outputs (cluster centers) serves as input to the Mappers that are launched for every iteration of the algorithm. This inherent checkpointing scheme provides a natural way to cope with failures. If there is a mapper failure, that mapper could be relaunched at the nodes that hold copies of the original input split and the reducer output. In the case of a reducer failure, the newly launched reducer could either pull outputs from the mappers or simply have the mappers redo processing for the iteration at which the reducer failed.

Granules relies on a *push based* scheme wherein the mappers stream their intermediate outputs to the reducer and the reducer streams its output back to the mappers. Since computations in Granules retain state, mappers maintain the original input data splits in memory and do not need to be reinitialized. Since there is no checkpoint, if there is a failure recovery is not possible. This is something that we will incorporate in the future and is beyond the scope of the current paper. To implement this fault tolerance feature, we will rely on interleaving I/O and CPU processing so that the data clustering completion times are not negatively impacted. Once the reducer's output has been streamed to the mappers we will checkpoint this information. In case of a mapper failure, the reducer can supply both the original input split and information about the cluster centers to relaunch map processing. In the case of a reducer failure, we will simply launch a new reducer and have the mappers stream their last outputs to the reducer.

IV. EXPERIMENTAL SETUP

To force Mahout to use the MapReduce version of an algorithm (as opposed to the sequential, in-memory version), datasets need to be stored within HDFS. Because of this, all our approaches (both Hadoop and Granules) initially read data from an HDFS cluster.

All tests are run on 2.4 GHz quad-core machines running Fedora 14 with 12GB RAM and a gigabit network connection. Each distributed run contains 25 mappers and a reducer (Mahout clustering algorithms are set to run with a single reducer, eliminating boundary conflicts). To properly compare Mahout performance across runtimes, when testing with Granules, the Granules resources are running on the HDFS worker machines.

Among the 25 machines, one was also responsible for acting as a namenode, and tasktracker, while five were acting as Brokers (for the Granules runs). In both cases, the Mahout operation was submitted from a machine outside the cluster.

A. Clustering Setup

For each clustering method we analyze: k-means, fuzzy k-means, dirichlet, and latent dirichlet allocation, we first use Mahout to generate a random set of starting clusters. We use the same set of starting clusters when contrasting the performance of Hadoop and Granules. Canopy clustering is a technique used to jumpstart other clustering algorithms, and usually only runs for a limited number of iterations, we will not be analyzing canopy clustering in this work.

V. DATASET

For the clustering example, we used the Reuters-21578 text categorization collection data set [9]. This data set contains 21,578 documents which appeared on the Reuters newswire in 1987. This dataset was generated following the ACM SIGIR '96 conference when it was decided the Reuters-22173 dataset should be cleaned and standardized in order to achieve more comparable results across text categorization studies. We processed the dataset to convert it to a format that Mahout can handle, based on the guidelines in [2]. This produced vectors of normalized bigrams from the input data with *term frequency – inverse document frequency* (TF-IDF) weighting. The TF-IDF weighting helps to lower the importance of words which occur often across all documents, such as "the", "he", "she", or "a."

We are clustering across all news documents in the dataset, so we have 21,578 points to cluster. There are 95,000+ dimensions of bigrams, or unique pairs of words. Since no single document contains all possible bigrams, these are stored internally using Mahout's `SparseVector` format.

VI. K-MEANS CLUSTERING

Mahout supports several different clustering algorithms. We initially start with k-means clustering, a clustering algorithm where the user estimates how many clusters are required (k) to adequately group all data. The algorithm then runs with this number kept constant – no clusters are added or removed during the computation. This algorithm was first introduced as a technique for pulse-code modulation [10].

K-means is the most basic clustering algorithm supported by Mahout, and operates on the principle that all data can be separated into distinct clusters. K-means requires the user to specify a k value, and the output can vary drastically based on not only the number of clusters chosen, but the initial starting points of all clusters. With respect to our dataset, when looking for very broad topic categories, a small value of k would be chosen (10-20). When looking for very small and finely honed categories, we would need to drastically increase k (1,000).

K-means clustering is a good choice when it is believed that all points belong to distinct groups. It can also a good choice when initially approaching a new dataset. K-means runs quickly, and can find large distinctions within data.

In the Hadoop implementation, the input data is separated into a number of files. This data consists of the points which will be clustered. Each map process is responsible for looping through its assigned file, and assigning the points to the nearest cluster. The mappers output a file which contains a cluster ID connected to the point which is assigned to it. The reducer will read in each file generated by the mappers, and move through the list of clusters assigning each point to it. Once this is complete, the reducer then computes new cluster centers, and generates a file containing the new clusters. The mappers read in the cluster data, as well as the points and the entire process repeats.

Our implementation in Granules follows the original Hadoop implementation. Each mapper node is responsible for loading a set of points into memory, and is responsible for clustering those points. In each iteration, a set of current clusters is made available to all mappers. Once each mapper has finished clustering their points, a set of `ClusterObversations` for each cluster is sent to the reducer. The reducer combines `ClusterObservations` from each mapper, and uses this information to update the cluster centers. This modified set of clusters is then sent to each mapper for the next round of iteration.

One major difference between the Hadoop and Granules versions is where the completion point is computed. The Hadoop version will calculate the maximum number of iterations in the mappers as well as in the reducer, while in Granules the mappers are unaware of the overall point in execution and the reducer is responsible for keeping track of rounds of execution.

A. K-Means Runtime Analysis

K-means is the simplest clustering algorithm we have benchmarked. In the Hadoop implementation, each iteration a Mapper is responsible for loading the current set of clusters from disk. Once the clusters are in memory, the Mapper then reads through the list of points assigned to it one at a time, and identifies which cluster the point belongs to. Once a point has been assigned to a cluster, it is written out to file. The overall cost of the Hadoop Map operation is $CR_D + NCR_DW_D$ where C is the number of clusters, N is the number of points a given mapper is responsible for clustering and R_D and W_D are read and write times to disk. It is important to note that these values include seek time as well as the time to actually read and write the data.

The Hadoop Reducer will first read in the current set of clusters from file, then read in the outputs from all the mappers. As the reducer reads in data, it modifies the clusters in constant time and writes out a modified cluster once it has finished processing all points assigned to the cluster. The overall runtime of the reducer is $CR_D + MNR_DW_D$, where M is the number of Mappers in the system.

In the Granules Mapper, we can take advantage of state retention by keeping the points to be clustered in memory, so we only need to read in the new states for every round of execution. We can also use state to make sure we send less data to the Reducer, helping cut down on the amount of work the Reducer needs to perform. The Granules Mapper runtime is: $CR_S + NC + CW_S$ where R_S refers to the cost of reading streaming data over a socket while W_S refers to writing data to a socket. It is important to note that this includes the cost of the streaming substrate overhead.

The Granules Reducer needs to read in the input from all the mappers, and send back out the newly computed clusters

for the next round of computations. The running time of this operation is $MCR_S + CW_S$.

Comparing the Mapper runtimes for the Granules and Hadoop Implementations, it is clear that both can be boiled down to an $O(NC)$ operation, and the major difference between them is simply the constants around that function. The Reducers have a different overhead by an order of N, yet we're not seeing a commensurate speedup in our benchmarks. This is because the majority of the computation is spent in the Mappers, while the role of the Reducers is relatively small in the overall computation.

B. K-means Clustering Results

In both Hadoop and Granules, we ran 20 rounds of k-means on 88 clusters for 100 iterations. The same set of starting clusters was used for each implementation. The results of these tests are displayed in Figure 1. and summarized in TABLE I.

Another observation is the difference in standard deviation of the running times, where Granules deviates by about 14 seconds, Hadoop varies by over 21 seconds.

TABLE I. K-MEANS CLUSTERING IN SECONDS

	Mean	Min	Max	SD
Granules	1749.76	1737.91	1772.52	13.696
Hadoop	3948.14	3922.475	3995.501	21.203

From these results, we can clearly see that the Granules implementation can outperform the Hadoop implementation by decreasing the amount of disk accesses necessary to complete the operation. Granules can even outperform the Hadoop version when both are running on top of HDFS, where the Hadoop workers can take advantage of data locality. Hadoop uses data locality to start a mapper on the node which holds the data the mapper needs to work with. By eliminating network I/O this locality speeds up access time.
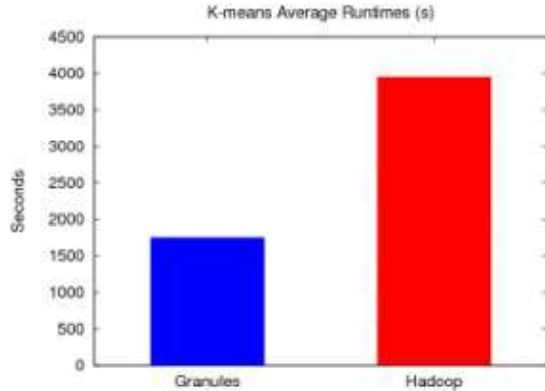

Figure 1. K-means Average runtime in Seconds

VII. FUZZY K-MEANS

Fuzzy K-Means [11] operates on a similar principle as k-means: The user chooses an initial set of k clusters, and allows the algorithm to run and adjust their centers as points are assigned to them. Fuzzy k-means allows an extra degree of freedom by allowing a point to belong to more than one cluster.

Using our dataset as an example, k-means is able to find the broad and overarching topics and group the articles accordingly; however, k-means cannot handle data points that span multiple topics. For example, a news article may discuss oil prices in the Middle East. With k-means, this article can either be clustered with articles about the Middle East, or articles discussing the prices of raw materials, but not both. Fuzzy k-means would allow the article to be associated with both topics, thus revealing a link between data that k-means could not show. Not only will fuzzy k-means show this overlapping of topics, it will also describe the degree to which the article is related to each topic.

Fuzzy k-means operates in roughly the same manner as k-means, with the modification that instead of each point belonging to a single cluster, each point is assigned a probability of belonging to every cluster. After this step, the reducer then goes through each probability, and adjusts cluster centers with respect to those points with the highest probability of belonging to the cluster.

A. Fuzzy K-Means Runtime Analysis

Fuzzy k-means is a slightly more complex clustering algorithm than k-means, and requires much more data to be sent between Map and Reduce phases. As fuzzy k-means computes the probability that each point belongs to every node, the Mapper now will pass NC information to the reducer instead of just N. The overall running time of the Hadoop Mapper is $R_DC + R_DNCW_D$. Again, R_D and W_D refer to reading from and writing to disk – including seek time, N is the number of points a Mapper is responsible for clustering, and C is the number of clusters.

The runtime for the fuzzy k-means Hadoop Reducer is very similar to the k-means version – it simply has to handle more data. The runtime is: $MNCR_D + CW_D$. Where M is the number of Mappers in the system.

The Granules fuzzy k-means Mapper has an overhead of $R_SC + NC + CW_S$, where R_S and W_S are the times to read stream data from and write it to sockets – including the streaming overheads. A major difference between this and the Hadoop Mapper is that Granules can retain state information and can aggregate outputs, so it only needs to send C to the Reducer, instead of NC.

The Granules Reducer takes advantage of the partial aggregation done by the Mappers, and needs to read in far less data than its Hadoop counterpart (by a factor of N). The runtime of the Granules Reducer is: $MCR_S + CW_S$, again with M being the total number of reducers.

Both the Granules and Hadoop approaches are bounded by the NC computation to compute the probability of each point belonging to every cluster, essentially bounding the runtime at $O(N)$. While we do see a big difference in the Reducer behavior, it is another computation where the work done by the Reducers is insignificant when compared to the work performed by the Mappers. Despite the Granules implementation have a quicker Reducer runtime by a factor of N, their overall runtimes are still very similar.

B. Fuzzy K-Means Results

Since the fuzzy k-means algorithm allows points to span multiple clusters it is very long running. We ran 20 rounds of fuzzy k-means for 25 iterations before halting the computations. These tests were run with 44 clusters. The same set of initial clusters was used for both Hadoop and Granules

versions. In Figure 2. the mean execution times can be seen. Hadoop is taking just a bit more time than Granules to finish processing the data. More detailed results are shown below in TABLE II. We can see that on average, the Granules implementation is finishing over 700 seconds (almost 12 minutes) sooner than Hadoop.
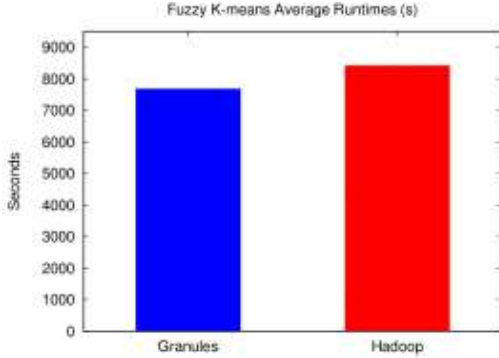

Figure 2.   Fuzzy k-means Average Runtime in Seconds

While we do not see the same increase in speed we get with k-means, the Granules implementation still manages to outperform Hadoop implementation by a small margin simply by cutting out the need for repeated reads/writes from file. From analyzing the source code, it appears that fuzzy k-means has a far higher ratio of CPU-to-I/O bound processing than the other clustering algorithms we discuss here, accounting for the difference in speedup..In order to isolate the bottleneck in the Granules implementation, we timed each step of the algorithm. Through this method, we found that the biggest bottleneck in our system was in the reducer.   Each round of execution, several seconds were lost with output from mappers waiting in the reducer's queue while it was processing previous inputs.

TABLE II.   FUZZY K-MEANS CLUSTERING IN SECONDS

	Mean	Min	Max	SD
Granules	7685.74	7676.47	7695.79	5.567
Hadoop	8423.78	8414.22	8431.15	5.812

VIII.   DIRICHLET CLUSTERING

This clustering algorithm [12] differs drastically from k-means. Most notably, there is no *k*. Dirichlet clustering may add and remove clusters as it deems necessary, and additionally can support different shapes of models. K-means and fuzzy k-means both assume that all clusters have normal distributions around a central point (in the case of 2-dimensional data it is circular). They cannot handle a distribution where clusters match a different model. Mahout currently supports models such as GaussianCluster, NormalModel, and SampledNormalDistribution; also allowing the user to define more models as needed.

Because of its complexity, Dirichlet clustering can take far longer to run than k-means or fuzzy k-means. Due to the many iterations it may go through, Mahout allows the user to specify the number of iterations to move through before writing cluster information to file – though this is only available for local in-memory runs.

Dirichlet clustering is a good initial clustering algorithm as it can help to determine an appropriate k to give the faster running k-means algorithms, or even help show why k-means may be having problems e.g.: if the data does not fit the circular model that k-means expects, Dirichlet should be able to cluster data where either k-means or fuzzy k-means fails.

A.   Dirichlet Clustering Runtime Analysis

As mentioned above, Dirichlet clustering follows a different paradigm than k-means or fuzzy k-means. For Dirichlet, the number of models *D* is a parameter. This algorithm also requires state to be passed between each iteration. In the Hadoop Mapper the state is first read in, then the algorithm is run as the points to cluster, *N*, are read in. This leaves the overall Mapper runtime at $DR_D + R_DNDW_D$, where R_D and W_D are read and write to disk respectively, and N is the number of points to cluster.

Each mapper writes data for every point, for every model, meaning that the Hadoop Reducer then has to read in all this information from every mapper, as well as load state. Once the reducer has finished processing all data for a model, it then writes the model information to disk to be read in as state information for the next round of computation. The overall runtime of the Reducer is $R_DD + R_DMND + W_DD$, where *M* is the number of Mappers in the setup.

The Granules Mapper follows the same approach as the Hadoop Mapper, but manages to cut down slightly on the overheads by aggregating data, and sending a smaller amount of data to the Reducer. This also helps to cut down on the Reducers runtime. For the Granules Mapper we see an overhead of $R_SD + ND + DW_S$, where R_S and W_S are read and write overheads for sending data to sockets, including the streaming overhead. Because of the partial aggregation of data at the mappers, the value associated with the write is only *D* instead of *ND*. Additionally, the Granules Mapper does not need to read in the points, completely removing a read operation.

The Granules Reducer takes advantage of the partial aggregation by having a much reduced read-in time: $R_SMD + DW_S$. Additionally, the Granules Reducer has no need to read in the current state, since it saves the state generated in the previous iteration. This cuts down significantly on the I/O time that Granules needs for this algorithm.

For this algorithm, both the Map and Reduce in the Hadoop implementation are bounded by $O(ND)$. In Granules, the Map has $O(ND)$, but the reducer only has $O(MD)$, and *M* is usually several orders of magnitude smaller than *N*. As we see in the next section, however, we are not seeing a speedup in Granules of several orders of magnitude. This is because the work done in the map and reduce portions of the algorithm are not balanced – the Mapper does far more work than the Reducer, so lowering the runtime of the Reducer drastically does not have a great effect on overall runtime.

B.   Dirichlet Clustering Results

We ran 20 iterations of Dirichlet clustering which ran for 40 iterations each. The results of these tests in both Granules and Hadoop are displayed below in TABLE III. Granules runs Dirichlet clustering to completion about five times faster than Hadoop, which is also visualized in Figure 3.

Dirichlet clustering relies heavily on state, and does not require as much processing of data points as fuzzy k-means. From these results, it seems clear that the majority of the processing time Hadoop spends is loading the state from file every step.

Again, we see a difference in standard deviation between Granules and Hadoop implementations. Hadoop varies by almost a minute, while Granules is just at 10 seconds.
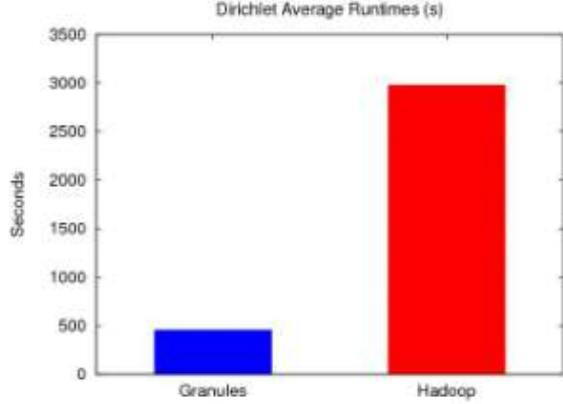

Figure 3.   Dirichlet Average Runtime in Seconds

TABLE III.   DIRICHLET CLUSTERING IN SECONDS

	Mean	**Min**	**Max**	**SD**
Granules	456.78	437.61	481.64	10.474
Hadoop	2980.24	2830.94	3068.12	46.794

IX.   LATENT DIRICHLET ALLOCATION

Latent Dirichlet Allocation (LDA) [13] is a clustering method similar to Dirichlet clustering. It is a generative model, so it starts off with a known model, and tweaks parameters to fit the model to the data. LDA can cluster words into "topics" by defining all documents as a mixture of all topics with a given probability. Much like k-means, LDA needs to be given a k, which identifies the number of topics in the dataset. The LDA classifier then attempts to discern the separate topics, and cluster each document into the appropriate topic. The algorithm reads through every word in every document, and calculates the probability that each word belongs to a topic. Based on the number of words in the document belonging to each topic, the overall topic of the document can be determined. LDA runs until the maximum number of iterations have been reached, or once the model has stabilized i.e. the amount of change between iterations has fallen below a given threshold.

Where algorithms such as k-means are very adept at grouping data with patterns not always apparent to humans, LDA can achieve results very similar to what would happen if we asked a human to cluster documents by topic [2]. The cost of this is that the algorithm takes many iterations to reach that level. LDA allows the process to be sped up by modifying a number of parameters which should help to cut down on the number of necessary iterations, such as automatically detecting stopwords and removing them from future calculations. LDA is a good clustering algorithm when one is looking for clustering that is human-understandable.

A.   Latent Dirichlet Allocation Runtime Analysis

LDA runtimes depend on the number of topics to be clustered by (T), as well as the dimensionality of the data ($|P|$). LDA analyses the number of times given bigrams appear in a document to determine the probability that that document belongs to a given group. In this algorithm state is passed between rounds of execution. This state primarily consists of a matrix which defines the relationship between all the bigrams and every topic. This means that the mappers need to load a $T|P|$ size array into memory before every run. This array is changed by the reducer between every round of execution, so even the granules approach hits this cost.

After loading the state, the mapper then needs to create an inference for every point to be clustered – this involves looping through the dimensions of the point to be clustered, and assigning weights based off of the state information gathered in the first step. The Hadoop Mapper performs this step as it reads in points, and writes out information as soon as it has calculated adjusted probabilities for every topic. This results in a runtime of $R_D T|P| + R_D N|P|T W_D$, where R_D and W_D are read and write to disk (including seek time), T is the number of topics, N the nodes to be clustered, and $|P|$ the dimensionality of every point in N.

The Hadoop Reducer has a far simpler task than the Mappers, it simply needs to read in data output by the Mappers and aggregate probabilities. The aggregated probabilities are then read in as the state table by the Mappers in the next iteration. The Hadoop Reduce runtime is $NMT|P|R_D + T|P|W_D$, with M being the number of Mappers.

As mentioned above, the Granules Mappers also need to load in state at the beginning, so it does not gain much improvement over the Hadoop implementation there. Granules can improve on the Hadoop implementation, however, by again performing partial aggregation at the Mapper. Instead of pushing out output immediately, the Granules Mapper can hold the information in memory until the algorithm has completed and only needs to write $T|P|$ data to the Reducer instead of $NT|P|$ data. The overall runtime of the Granules Mapper is $T|P|R_S + N|P|T + T|P|W_S$, with R_S and W_S being overheads for reading and writing data to sockets, including the streaming overhead.

The Granules Reducer can again take advantage of the decreased size of input and has a runtime of $MT|P|R_S + T|P|W_S$. While this is not as great of an increase as we saw with Dirichlet clustering, it still allows the Granules implementation to gain an edge over Hadoop.

The Hadoop Mapper and Reducer are both essentially bound by $O(NT|P|)$. The Granules Mapper has almost the exact same runtime, but the Granules Reducer is only bound by $O(T|P|)$. Again, this looks like it should lead to a much larger margin in performance between the Hadoop and Granules implementations, but the disparity between workloads holds true for LDA as well: even if we speed up the Reducer, the Mapper is still slowing us down too much for it to be noticeable.

B.   Latent Dirichlet Allocation Results

In our tests, we ran LDA for 40 iterations clustering into 10 topics. While this was not enough iterations to allow the model to converge, this is enough iterations to give us a good idea of

how the algorithm runs. We ran the full 40 iterations with both Granules and Hadoop versions, and compared the running time of each below in TABLE IV.

TABLE IV.   LDA CLUSTERING IN SECONDS

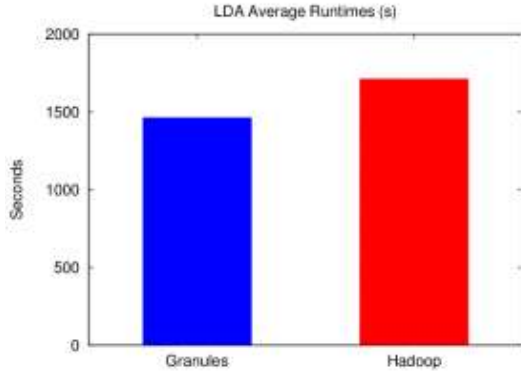|	Mean	Min	Max	SD
Granules	1463.61	1430.64	1468.93	8.477
Hadoop	1712.64	1682.19	1756.12	32.595

Figure 4.   LDA Average Runtime in Seconds

Figure 4. shows a direct comparison of mean execution times of LDA for Granules and Hadoop. On average, Granules finished about 4 minutes earlier than the Hadoop implementation. Again, this seems to be a direct result of Granules' ability to stream data between stages, instead of needing to write to disk between every step.

Interestingly, the difference between standard deviations is again very high in our experiments with LDA. LDA requires a mixture of overhead to read in state, as well as significant processing time needed to build probability tables once the state has been read in. It is interesting to see that it has a very similar profile to fuzzy k-means, the other algorithm we examine with a heavy CPU processing load. Both feature relatively close execution times with a large difference in standard deviation.

X.   RELATED WORK

Distributed machine learning has been a big topic for several years, not only on multicore machines [14], but also GPUs [15]. While many works mention machine learning applications running in a distributed environment [1], [16] they do not go into depth about the details of their implementations, and have not made the libraries available to the public.

Mahout offers access to many varied machine learning algorithms, but it is geared towards developing enhanced recommenders which can use multiple different clustering and classification algorithms to help generate recommendations. The Twister Iterative Map-Reduce runtime [17], on the other hand, has been developed to help biologists leverage the many parallel algorithms available for bioinformatics research. It allows biologists to specify a high-level workflow without needing a strong background in high performance computing.

XI.   CONCLUSIONS AND FUTURE WORK

Our results demonstrate the feasibility of using the Granules runtime for clustering algorithms. Since Granules supports computations that can execute multiple, successive

rounds of execution while retaining state it is particularly well suited for clustering algorithms that are inherently iterative. An added feature in Granules of streaming results between stages of an execution pipeline and activating computations when such (intermediate result) streams are available allows us to support distributed implementations of the clustering algorithms in an efficient fashion. Our benchmarks bear this out for the majority of our tests.

Our future work in this area will target two aspects. First, we plan to implement support for fault tolerance as outlined in section III.C. Second, we will focus on implementing Mahout's classification algorithms in Granules and profiling their performance.

REFERENCES

[1]   J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," *ACM Commun.,* vol. 51, pp. 107-113, Jan. 2008 2008.

[2]   S. Owen*, et al.*, *Mahout in Action*: Manning Publications, 2011 (est.).

[3]   T. White, *Hadoop: The Definitive Guide*, 1 ed.: O'Reilly Media, 2009.

[4]   S. Pallickara*, et al.*, "Granules: A Lightweight, Streaming Runtime for Cloud Computing With Support for Map-Reduce," in *IEEE International Conference on Cluster Computing*, New Orleans, LA., 2009.

[5]   S. Pallickara*, et al.*, "An Overview of the Granules Runtime for Cloud Computing," in *IEEE International Conference on e-Science*, Indianapolis, 2008.

[6]   D. Borthakur. (2007). *The Hadoop Distributed File System: Architecture and Design*. Available: http://hadoop.apache.org/common/docs/r0.18.0/hdfs_design.pdf

[7]   M. Isard*, et al.*, "Dryad: distributed data-parallel programs from sequential building blocks," in *2nd ACM SIGOPS/EuroSys European Conference on Computer Systems*, Lisbon, Poutugal, 2007.

[8]   K. Ericson*, et al.*, "Analyzing Electroencephalograms Using Cloud Computing Techniques," in *IEEE Conference on Cloud Computing Technology and Science*, Indianapolis, USA, 2010.

[9]   D. D. Lewis, "Reuters-21578 text categorization test collection, Distribution 1.0," A. T. L.-. Research, Ed., 1.0 ed: UCI Machine Learning Repository, 1997.

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

[11] J. C. Bezdek, *Pattern Recognition with Fuzzy Objective Function Algorithms*. Norwell, MA: Kluwer Academic Publishers, 1981.

[12] P. McCullagh and J. Yang, "How many clusters?," *Bayesian Analysis,* vol. 3, pp. 101-120, 2008.

[13] D. M. Blei*, et al.*, "Latent dirichlet allocation," *J. Mach. Learn. Res.,* vol. 3, pp. 993-1022, 2003.

[14] C.-T. Chu*, et al.*, "Map-Reduce for Machine Learning on Multicore," in *Advances in Neural Information Processing Systems (NIPS)*, Vancouver, Canada, 2006.

[15] B. Catanzaro*, et al.*, "Fast support vector machine training and classification on graphics processors," presented at the Proceedings of the 25th international conference on Machine learning, Helsinki, Finland, 2008.

[16] Y. Yu*, et al.*, "DryadLINQ: a system for general-purpose distributed data-parallel computing using a high-level language," presented at the Proceedings of the 8th USENIX conference on Operating systems design and implementation, San Diego, California, 2008.

[17] C. Hemmerich*, et al.*, "Map-Reduce Expansion of the ISGA Genomic Analysis Web Server," IEEE CloudCom 2010, Indianapolis, USA, 2010.