Lightweight MapReduce Framework

Zihao He (zihaohe), Shiming Zhuang(szhuang)

December 16, 2018

1 Summary

We implemented a lightweight MapReduce framework using C++, and demonstrated several machine learning algorithms (e.g. naïve bayes, logistic regression etc.) on top of it, which shows the feasibility of large scale parallel machine learning algorithms.

2 Background

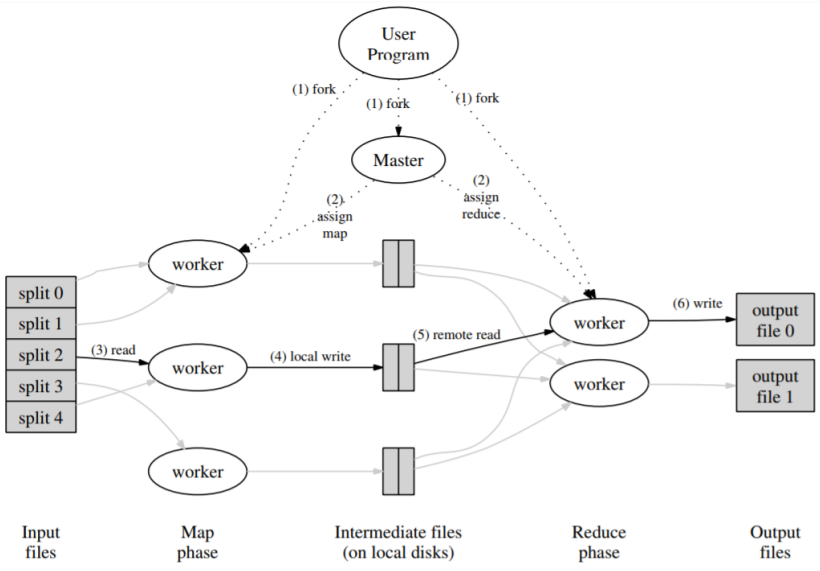

Figure 1: MapReduce framework overview. Image credit: Dean, Ghemawat

As big companies are collecting more and more data (e.g. logging info, clicking data, social network info etc.) from users, they are facing a tough problem of dealing with a large amount of data. The processing and analyzing tasks require huge computation power from hundreds or even thousands of computers. Many of the problems are pretty straightforward such as word counting, naïve bayes classification etc., and the intrinsic difficulty instead

becomes how to distribute and parallelize the workload efficiently. To focus on the business logic itself instead of the general parallelism problem, Jeffreay Dean etc. proposed a programming model named MapReduce[1], where users specify a map function to generate a set of intermediate key/value pairs, and a reduce function to merge all intermediate values with the same key (see diagram above). It turns out that many real world problems can be expressed in this model. And this idea enables great convenience for programmers without any experience with parallel systems to write highly scalable programs.

Our 15618 project is to implement such a MapReduce framework that will take care of the details of partitioning the input data, scheduling and distributing tasks and so on. It is lightweight because it does not handle unexpected situations such as machine failures, network interruptions or hard drive problems etc, and it will take advantage of existing distributed file system (e.g. AFS) for sharing partial results. The framework will support dynamic work distribution by assigning one master which saves the task states and controlling the rest workers.

Furthermore, it is more useful to test it with some real life problems and examine the speedup under different settings (e.g. with how many machines and how much workload on each machine etc.). To start with, we implemented word counter as a "hello world" version of MapReduce task where multiple workers collaborate to find out how many times a word appears in the document. For naïve bayes, two rounds of MapReduce for training and another two rounds for predicting are involved. And for kmeans, number of rounds is based on a threshold which further requires consensus among all the workers. Many of these algorithms demand different numbers of Mappers and Reducers in different rounds, which also requires the master dynamically distribute the workload. The details of the framework as well as the implementation of each machine learning algorithms are described in the following section.

3 Approach

3.1 Workflow

The main workflow is as above in Figure 2. The master is executed by the user and then it will ssh to other machines to bring up all the workers specified in a configuration file by the user. Note that the program is in a single binary file which has the ability to turn into a master as well as a worker. What it will be is determined by the argument passed to it at the beginning. The task may be splitted into multiple stages and requires several rounds of Maps and Reduces (e.g. KMeans has a threshold to stop the iteration, Naïve Bayes has two stages for training, etc.), and the scheduling is solely controlled by the master node. After each stages there is a barrier for all the workers and only then the master will trigger the next stage.

After all the workers have checked in with the master at the beginning, stage 1 starts. At each stage the master would assign each worker to be a mapper or reducer. A mapper

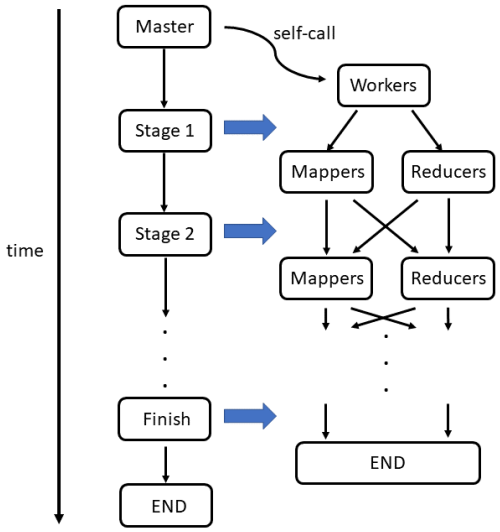

Figure 2: Work flow of a task

will be assigned an input file by the master and notify the master when it is done and get its next input. The output of mappers are called temporary files or partial results which will be used by reducers as input and be deleted after this stage. After all the mappers are done, reducers are assigned to work on the partial results and output the final result of this stage in the end. The stage finishes after all the reducers are done, and then the next stage starts until the end.

The worker is able to turn into a mapper or reducer dynamically, which is beneficial for the whole speedup. Because in some cases the mapper side has much more work to do and it would be helpful to assign more mappers than reducers. Note that a single machine can run multiple workers at the same time, so the MapReduce framework takes advantage of both multi-process parallelism and distributed system.

3.2 Network Communication

We designed our own network communication utility upon TCP using libevent library instead of using OpenMPI for more flexibility. It also bears some features from OpenMPI.

- It is asynchronous based on messages. Handlers are assigned for incoming packets.
- Unexpected network or worker failure would cause task failure because we want to ensure integrity of the final result.
- Communication endpoints are identified by indices in the configuration file.

We designed our own packet header as in Figure 3. *msg_type* includes *LMR_CHECKIN*, *LMR_ASSIGN_MAPPER*, *LMR_MAPPER_DONE* etc. for assigning handlers more conveniently. For example, upon receiving *LMR_CHECKIN* the master will increase the counter

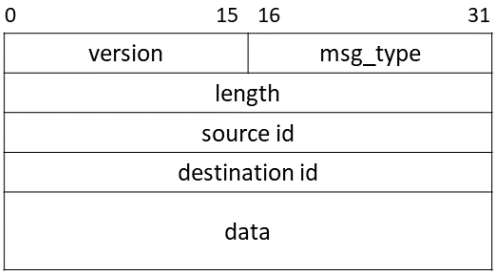

Figure 3: Packet header

and check whether all workers have checked in. Upon receiving *LMR_ASSIGN_MAPPER* the mapper will start working on the input specified in the *data* field.

In the message handler, it will spawn a new thread to do the work as it may take a long time and block later messages. When encountering unexpected worker failure, the TCP connection with the master is down which will cause the master to shut down the whole task.

3.3 MapReduce Version of Machine Learning Algorithms

We adapted several machine learning algorithms from [2,3,4] to our framework as built-in functions. An example code of KMeans clustering is as follows.

A specification is created for a MapReduce instance, and then the machine learning class takes the instance as input for low-level excecution setting. Just call *train* and *predict* and it will parallelize the process automatically. Execution details are stored in *result* for analyzing.

```
MapReduceSpecification spec;
MapReduceResult result;

spec.config_file = "config.txt";
spec.index = (argc == 2) ? atoi(argv[1]) : 0;
spec.num_mappers = 10;
spec.num_reducers = 10;

MapReduce mr(&spec);
ml::kmeans km(&mr);

// 10 inputs, iteration threshold is 0.1, maximum 20 iterations
km.train("kmeans/input_%d.txt", 10, "kmeans/centroids.txt", 0.1, 20, result);
km.predict("kmeans/input_%d.txt", 10, "output/result_%d.txt", result);

// print prediction time cost.
printf("%.3fs elapsed.\n", result.timeelapsed);
```

Take KMeans as an example to show how to parellelize a machine learning algorithm using the idea of MapReduce.

We are going to share some information, i.e. the cluster centroids, across iterations. They are saved in a file on AFS which is accessible to all workers. It at first contains the initial K cluster centroids (or K random data samples) and will be updated after each iteration to contain the latest centroids calculated by Reducer.

For training, we just need to derive this file. So after initialization,

a. The *Mapper* reads this file to get the centroids from last iteration. It then reads the input data and calculates the Euclidean distance to each centroid. It associates each instance with the closest centroid, and outputs (data instance id, cluster id).

b. To compress the size of partial result, we use a *Combiner* to calculates the average of the data instances for each cluster, along with the number of instances. It outputs (cluster id, (intermediate cluster centroid, number of instances)).

c. The *Reducer* calculates the weighted average of the intermediate centroids, and outputs (cluster id, cluster centroid) into the file.

For predicting, we need to associate each data sample with the closest centroid.

a. The *Mapper* reads the centroid file as well as the input data, and then calculates the Euclidean distance to each centroid. It associates each data sample with the closest centroid, and outputs (data instance id, cluster id).

b. The *Reducer* just directly output the input pairs.

Assume there are N data samples with dimension d and K clusters, and training process needs t iterations. Then the time complexity is $O(NKdt)$. In MapReduce, if there are m mappers and r reducers, the complexity is $O((\frac{N}{m}+\frac{m}{r})Kdt+S)$ where S stands for scheduling overhead and is related to the number of mappers and reducers. It shows that the speedup is less than m, but it should be close to m if S, m, r are relatively small.

4 Results

4.1 Results for Different Algorithms

4.1.1 K-Means

The experiment setting is shown in 1. The performance of K-Means on GHC Cluster is shown in 4. The base line is the 4 mappers version of the algorithm. The graph show that the algorithm scales well when the number of mappers is low(the time used by reducer is less than 1 second, thus it is ignored in discussion). However, as the number of mappers increase, the overhead of network communication overweight the number of mappers, thus the performance does not scale well.

Experiment Setting	Value
Dataset	US Census 1990
Platform	GHC Cluster
CPU	Intel(R) Xeon(R) CPU E5-1660 v4 @ 3.20GHz
Data Size	2458285
#Nodes	8
#Iterations	20

Table 1: K-Means Experiment Setting

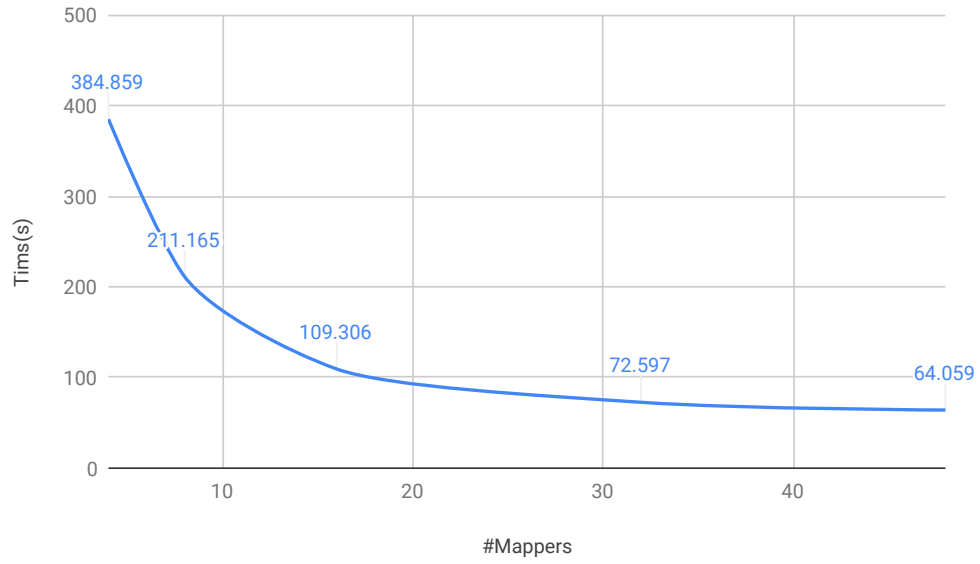

Figure 4: K-Means Performance

Experiment Setting	Value
Dataset	Diabete Dataset
#Data	3686400
#Dimension	39177
Platform	GHC Cluster
CPU	Inter Intel(R) Xeon(R) CPU E5-1660 v4 @ 3.20GHz

Table 2: Linear Regression Experiment Setting

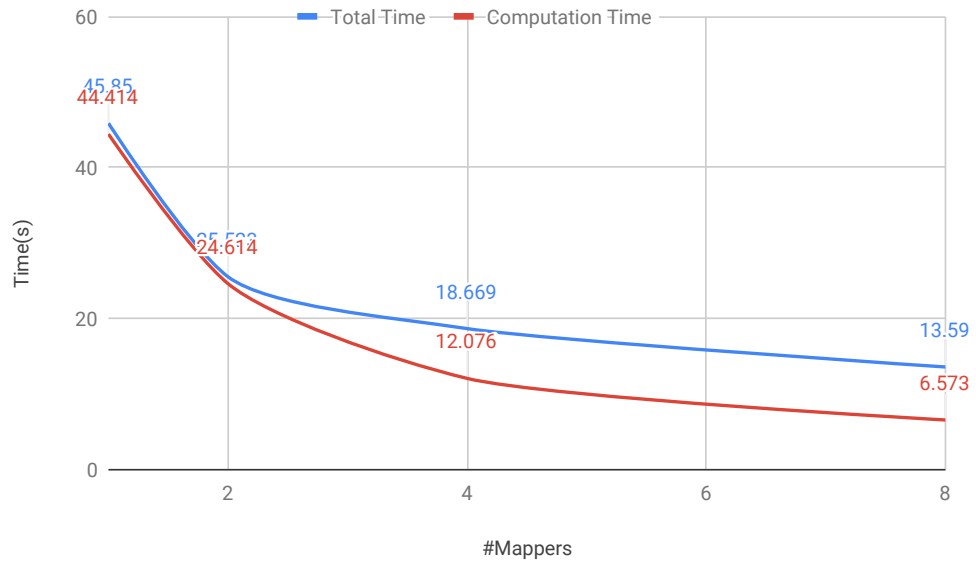

Figure 5: Linear Regression Performance

4.1.2 Linear Regression

The experiment setting is shown in 2. The performance of Linear Regression on GHC Cluster is shown in 5. The base line is the 1 mappers version of the algorithm. The graph show that the computation time scales well(The computation time of reducer is less than 1 second, thus it is ignored in the graph). However, the total time of the algorithm is not ideal. One of the reason is that the framework assign mapper and reducer sequentially. Thus when the number of mappers and reducers increase, the overhead of assignment increase as well. Since the linear regression implemented in this project is a 1-Pass closed-form linear regression algorithm, the assignment of mappers and reducer overweight the computation when its number increase.

4.1.3 Logistic Regression

Experiment Setting	Value
Dataset	Movie Review Dataset
#Data	2300
#Dimension	39177
Platform	Personal Computer
CPU	Intel i5-8259u

Table 3: Logistic Regression Experiment Setting

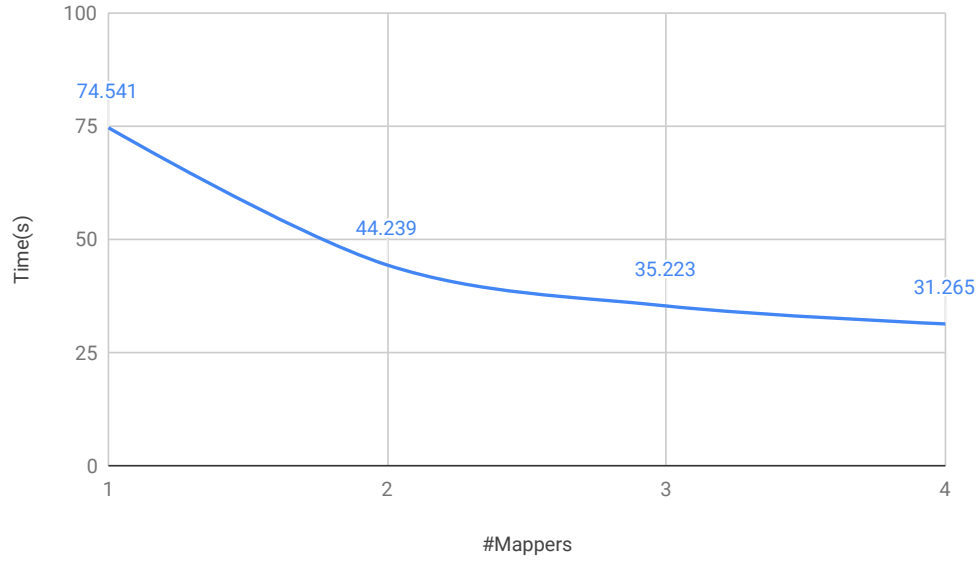

Figure 6: Logistic Regression Performance

The experiment setting is shown in 3. The performance of Logistic Regression is shown in 6. The base line is the 1 mappers version of the algorithm. The graph show that the algorithm scales well when the number of mappers is low(the time used by reducer is less than 1 second, thus it is ignored in discussion). However, as the number of mappers increase, the overhead of network communication overweight the number of mappers, thus the performance does not scale well.

4.2 Analysis for K-Means

K-Means is chosen in this part because it is a multi-iteration algorithm. Although the overhead of mappers and reducers assignment increase as the number of mappers and reducers increase, it can be amortized as the number of iterations increase, thus it can shown more details about the algorithm.

4.2.1 Number of Tasks

The experiment setting is shown in 4. The performance different number of task is shown in 7. The graph shows that the overhead of getting new tasks from the work queue is about 10% of the total run time. However, as the number of tasks increase, the performance is slightly improved. The log of the algorithms show that as the number of tasks increase, some of the mappers tend to execute more tasks. In other word, more tasks give better work-load balance. In this experiment, the input file is partitioned evenly. In real word applications, it is highly possible that the input files have different size. The dynamic schedule of mappers and reducers are able to ipmrove the performance in this situation.

Experiment Setting	Value
Dataset	US Census 1990
Platform	GHC Cluster
Data Size	2458285
#Mappers	16
#Iterations	20

Table 4: Different #Tasks Experiment Setting

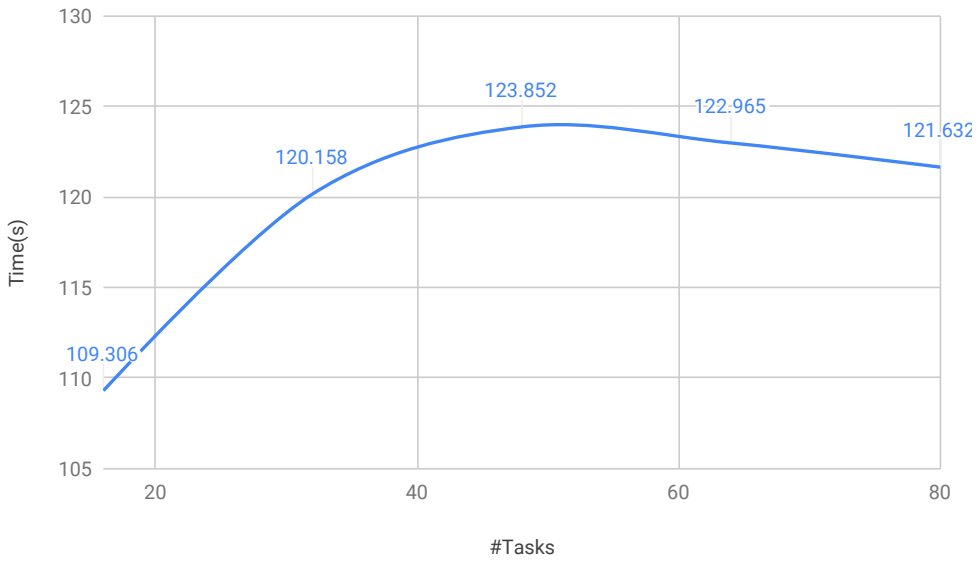

Figure 7: K-Means Performance with Different #Tasks

4.2.2 Number of Nodes

Experiment Setting	Value
Dataset	US Census 1990
Platform	GHC Cluster
Data Size	2458285
#Mappers	16
#Iterations	20

Table 5: Different #Nodes Experiment Setting

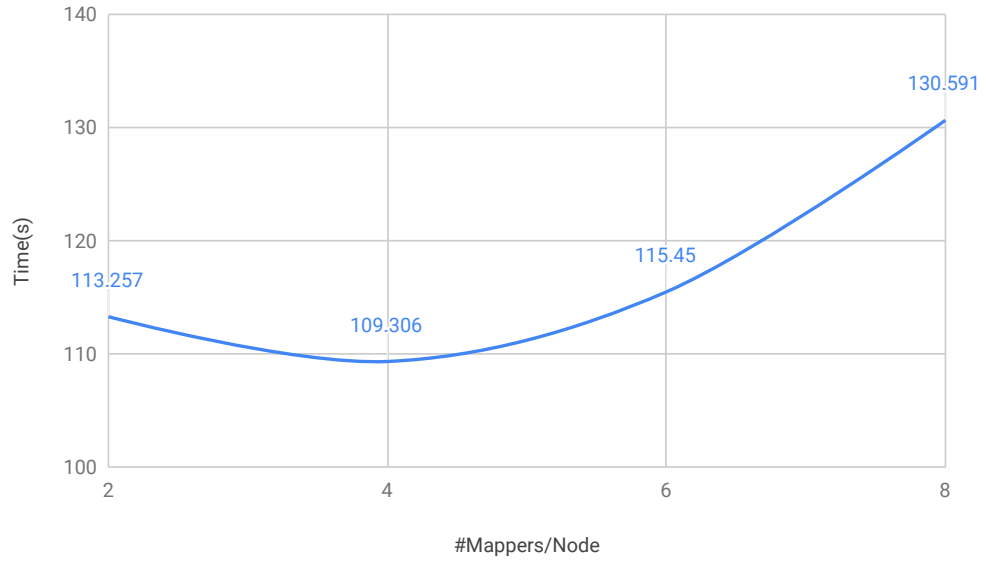

Figure 8: K-Means Performance with Different #Mappers/Node

The experiment setting is shown in table 5. The performance different number of task is shown in 8. The reason why the performance of 8 mappers per node is not ideal is that the CPU have 8 cores. The design of the framework is a multi-thread system. In addition to worker thread which works on the tasks, there are background thread for network communication. For this reason, the mappers are influenced by the system schedule, and the performance is not ideal.

4.2.3 Data Size

The experiment setting is shown in 6. The performance different data size is shown in 9. It is shown in the graph that when the problem size is large enough, the framework is easy to scale.

Experiment Setting	Value
Dataset	US Census 1990
Platform	GHC Cluster
#Mappers	16
#Iterations	20

Table 6: Different Data Size Experiment Setting

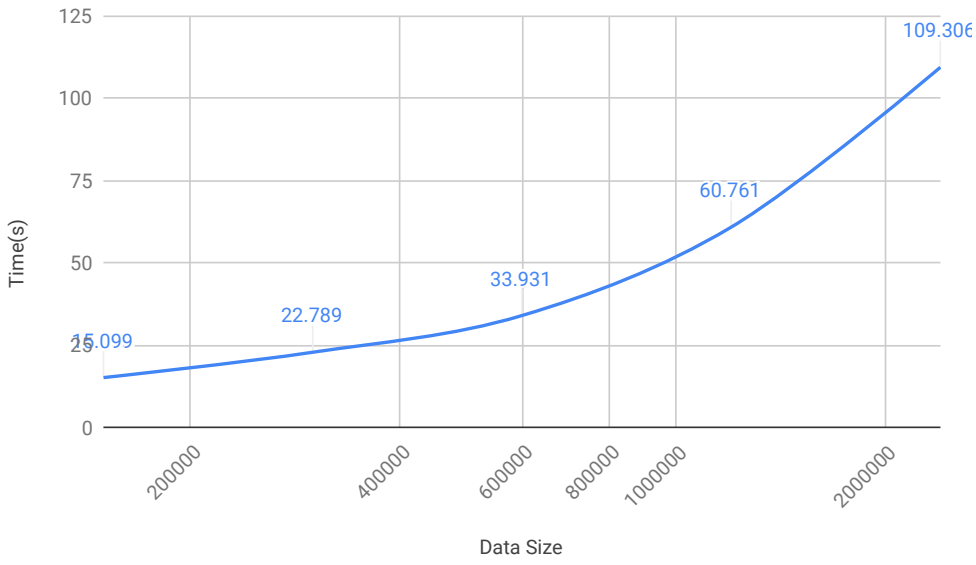

Figure 9: K-Means Performance with Different Data Size

4.3 Result Conclusion

The result of the experiment shows the following features and issues of the framework.

Features:

- Scalability: The applications can be easily implemented on multi-node system, and the scalability of the application is ideal when the problem size is large enough.
- Dynamic Scheduling: The framework support dynamic scheduling for mappers and reducers. When the input files are not partitioned evenly, this feature can improve the work-load balance of the system.
- Connection Reuse: After the connections between master and workers are set, different rounds of MapReduce will reuse these connections to avoid starting overhead.

Issues:

- Starting Overhead: For application with few iterations, the overhead of workers assignment would overweigh the benefit of more workers. It is possible to execute the assignment command in parallel to fix this issue.
- Critical Section: When mappers or reducers send finishing message to master, there is a critical section in master to protect the job queue and finishing counter, which must be done sequentially and may limit the scalability.
- CPU Consumption: In order to perform efficient communication, the libevent library needs to maintain background thread for messages. This overhead is unavoidable for the framework, and it is a trade-off between scalability and performance.

5 Special Thanks

We are inspired by several great projects.

- libevent, an event notification library. https://libevent.org
- libssh, an SSH client library. https://www.libssh.org
- mapreduce-lite, a C++ implementation of MapReduce. https://github.com/wangkuiyi/mapreduce-lite

References

[1] J. Dean and S. Ghemawat, MapReduce: Simplified Data Processing on Large Clusters. Communications of the ACM, 2008.

[2] C. Chu, S. Kim, Y. Lin, Y. Yu, G. Bradski, A. Ng and K. Olukotun. Map-reduce for machine learning on multicore. Proceedings of the 19th International Conference on Neural Information Processing Systems, 2006.

[3] M. Bodoia. MapReduce Algorithms for k-means Clustering. CME 323: Distributed Algorithms and Optimization, Stanford. 2016

[4] W. Cohen. Naive Bayes and Map-Reduce. 2017