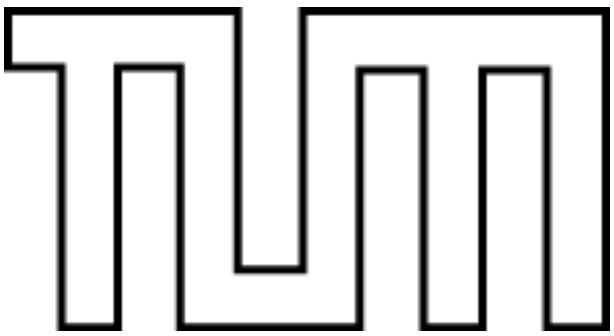


Technische Universität München

Faculty of Computer Science

Iterative MapReduce and its applications

Guided research

Gennady Shabanov

Supervised by: Dipl.-Ing. Jose Adan Rivera Acevedo

Advisor: Prof. Dr. Hans-Arno Jacobsen

Contents

1 Introduction

The current rise in interest in distributed machine learning applications can be explained by the evolution of hardware architectures and software frameworks that make it easy to manage the large amount of data and process information in a distributed fashion. In the most cases behind a ML algorithm an optimization problem has to be solved where an optimization task takes the prominent part of the computation time. As a result, distributed solution methods are highly desirable. In this paper we discuss the leveraging the MapReduce framework for parallelizing machine learning methods and optimization problems. We will focus on the Alternating Direction Method of Multipliers (ADMM) algorithm for distributed convex optimization and how it can be implemented in a MapReduce framework. The main difficulty is that ADMM is an iterative algorithm, but MapReduce tasks are not designed to be iterative ,i.e., they do not preserve state in the mappers across iterations. Thus, the main research goal is to investigate implementation caveats of MapReduce applied to machine learning problems. The research tasks can be formulated as follows:

- Review of MapReduce frameworks
- Reproduce state-of-the-art PageRank in MapReduce (this is the main representative of iterative algorithms) and compare implementations in different frameworks
- The implementation of ML algorithms in MapReduce using ADMM and results discussion

2 MapReduce programming model

MapReduce is a popular model of distributed computations. It allows to a user define map and reduce functions which process data in parallel. These methods have been borrowed from functional programming where **map** applies function to every item of data and returns a list of results. The **reduce** function applies function of two arguments cumulatively to items of two arguments so that the result is reduced to one single value. For example, reduce(lambda x, y: x+y, [1, 2, 3, 4, 5]) calculates ((((1+2)+3)+4)+5).
Let us consider workflow of MapReduce process on a cluster. Data set is split into data blocks then each mapper reads corresponding data and performs processing. The results from different mappers then merged with the use of reducers. Each mapper and reducer are represented as working machines in the cluster. There are several advantages of the MapReduce approach:

- MapReduce abstraction hides underlying complexity of distributed software, hiding the messy details of parallelization, fault tolerance, data distribution and load balancing

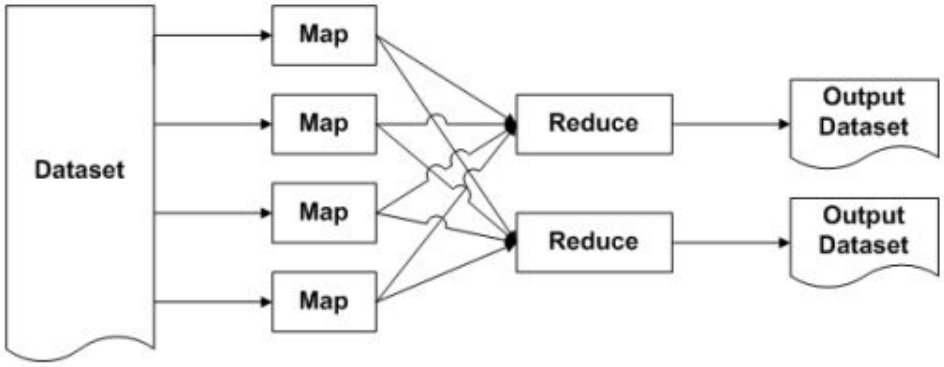


Figure 1: Mapreduce work flow

- The amount of cloud infrastructure available for MapReduce computing can make it convenient to use in practice, especially for large problems
- MapReduce procedure comprises map and reduce functions which are simple to understand, thus it allows to develop distributed applications without special knowledge of parallelization
- Wide range of free software frameworks implementing the computational paradigm

3 Overview of mapreduce software frameworks

3.1 Apache Hadoop

Apache Hadoop is a well-known open source software framework for cluster distributed computing. Hadoop is written in the Java programming language and uses Hadoop Distributed File System (HDFS). The core architectural goal of HDFS is to provide automatic recovery from faults in the presence of thousands server machines. HDFS can be characterized as having the following features:

- Allows to store data reliably in the presence of failures in large clusters
- Provides cluster rebalancing (automatic move of data from one cluster to another)
- Provides data integrity (stores checksums for each block of the file)
- Write-once-read-many access model for files
- Support of replication

Although Hadoop have been widely applied to large-scale data analytics, it is inefficient for an important class of applications in which the data is reused across multiple computations. For example, in many iterative machine learning algorithms, such as PageRank,

K-means clustering, and logistic regression. The main problem is that the result of computations are saved in HDFS, then this stored data should be retrieved again for the next iteration. Another issue is that the path where the previous data is stored needs to be known. Since HDFS stores results in the disc frequent data reuse produces high I/O rate which significantly influence the performance of the system. Let us consider PageRank algorithm implemented in Hadoop.

Algorithm 1 PageRank, Apache Hadoop

```
1: class PageRank
2:     method Initialize
3:         Initialize initial PageRank values to 1
4:     end method
5:
6:     method Map(Data (node, neighbors))
7:         for all Neighbor outlink ∈ List neighbors do
8:             Emit(outlink, rank = node.pagerank/neighbors.size)
9:         end for
10:    end method
11:
12:    method Reduce(Data (link, rank)) //Reduce by key
13:        for all links do
14:            pagenrank = SUM rank with the same link value
15:            Emit(link, 1 − DF + DF ∗ pagenrank)
16:        end for
17:    end method
18:
19:    method Main(inputPath)
20:        for i = 1 to MAXITERS do
21:            outpuPath = new Path
22:            run MapReduceJob(inputPath, outpuPath)
23:            inputPath = outpuPath
24:            proceed until a convergence criteria satisfied or MAXITERS achieved
25:        end for
26:    end method
27:
28: end class
```

PageRank has the following formula:

$$PR(p_i) = 1 - d + d \left(\sum_{p_j \in M(p_i)} \frac{PR(p_j)}{L(p_j)} \right) \tag{1}$$

where $PR(p_i)$ is the PageRank of the document p_i, $M(p_i)$ is the set of pages that link to p_i, $L(p_j)$ is the number of outbound links from the page p_j, and d is a damping factor

wich signifies the probability that the user will not jump to a random page. Usually d is set to 0.85. On the map phase, the PageRank of a node evenly divided by the number of its neighbors and each piece is passed to each neighbor, producing pair [neighbor, piece of the node's PageRank]. On the reduce phase pieces of the links with the same URL are summed up and a new page rank is recomputed. Then the results saved to a new location and on the next iteration the output path is assigned to the input path. Also, at the end of iteration we end up with exactly the same data structure as the beginning, which is the requirement for iterative algorithms to work.

3.2 Apache Spark

The main goal of Spark open source cluster computing system is to provide for users fast data analitics. The key difference in comparison with Hadoop is that the results of computations can be stored in RAM memory rather then using disk I/O. That is why Spark out-performs Hadoop on iterative algorithms. Spark is written in the Scala programming language and provides API in Python, Scala, Java. Also, Spark supports

Algorithm 2 PageRank, Spark (simplified)

```
1: class PageRank
2:     method Initialize
3:         Initialize initial PageRank values to 1
4:     end method
5:
6:     method Map(RDD (node, neighbors))
7:         for all Neighbor outlink ∈ List neighbors do
8:             Emit(outlink, rank = node.pagerank/neighbors.size)
9:         end for
10:    end method
11:
12:    method Reduce(RDD (link, rank)) //Reduce by key
13:        for all links do
14:            pagenrank = SUM rank with the same link value
15:            Emit(link, 1 − DF + DF ∗ pagenrank)
16:        end for
17:    end method
18:
19:    method Main(RDD nodes = [(n1, neighbors_1), ..., (n_N, neighbors_N)])
20:        for i = 1 to MAXITERS do
21:            nodes.ranks = nodes.Map.Reduce
22:            proceed until a convergence criteria satisfied or MAXITERS achieved
23:        end for
24:    end method
25:
26: end class
```

multithread execution. The core Spark feature which allows in-memory querying of data is Resilient Distributed Datasets (RDDs). RDD is immutable, partitioned collection of objects. It can be created through transformations on either data in stable storage or other RDDs. The example of transformations include map, filter, join. To provide fault tolerance Spark logs transformation on data (lineage), rather than storing the actual dataset. It means that if a partition of a RDD is lost, the RDD has enough information to derive the partition by relaunching the sequence of transformations. Let us consider an example of PageRank algorithm (Spark implementation in Algorithm 2). In the main loop (lines 20-23) the MapReduce phase is invoked over RDD on each iteration keeping the data in-memory. In the case of lack of the RAM memory Spark spills the data into the disk. In comparison with Hadoop Spark keeps the RDD reference of previously computed results. Thus, a programmer does not set explicitly where the previous results are stored.

3.3 GraphLab

GraphLab is a new framework which is designed for large-scale distributed problems and also support multicore computations. It shows faster results than Hadoop when running iterative algorithms. GraphLab gives more flexibility in writing distributed applications than MapReduce paradigm providing more API functions and allowing to define the stucture of communication between nodes. For example, MapReduce provides only two functions (map, reduce) and communication is accomplished only between workers and a master node. While GraphLab provides fold, apply and merge functions and the structure of communication can be encoded using graph abstraction. The main drawback of GrapgLab is that the distributed fault tolerance is not implemented yet, thus the framework can be used only on small clusters. In the paper we do not make performance measurements of GraphLab.

4 ADMM implementations in MapReduce

4.1 SVM

For support vector machine (SVM) the optimization problem may be formulated as follows:

$$\begin{aligned} &\text{minimize } f_0(w,b) = \frac{1}{2}||w||_2^2 \\ &\text{subject to } f_i(w,b) = y_i(w^T x_i + b) - 1 \leq 0 \ i = 1, ...m \end{aligned} \tag{2}$$

where w, b are the parameters to be found, x_i is ith data point, $y_i \in \{-1, 1\}$. The goal is to find a separating hyperplane with a maximum margin. We can write down an equivalent problem by using a hinge loss:

$$\text{minimize } \frac{1}{2}||w||_2^2 + \lambda \sum_j max(Ax + \mathbf{1}, 0) \tag{3}$$

where $A = [-y_j x_j - y_j]$ and $x = (w, b)$. The problem can be solved in ADMM in the following way:

$$\begin{aligned} x_i^{k+1} &:= \text{argmin}_{x_i}(\mathbf{1}^T(A_i x_i + \mathbf{1})_+ + \frac{\rho}{2}||x_i - z^k + u_i^k||_2^2) \\ z^{k+1} &:= \frac{\rho}{(1/\lambda) + N\rho}(\bar{x}^{k+1} + \bar{u}^k) \\ u_i^{k+1} &:= u_i^k + x_i^{k+1} - z^{k+1} \end{aligned} \tag{4}$$

where $\rho > 0$ is the augmented Lagrangian parameter, λ is dual variable, $\bar{x} = \frac{\sum_i^N x_i}{N}$, and $\bar{u} = \frac{\sum_i^N u_i}{N}$. As stated in [1] the following stopping criteria can be choosen:

$$r^k = x^k - z^k \tag{5}$$

$$s^k = -\rho\left(z^{k+1} - z^k\right) \tag{6}$$

If these two resudiuls are small then objective must also be small. Then we can choose two parameters ϵ^{pri} and ϵ^{dual} as convergence paramteres, i.e.,

$$||r^k||_2 \leq \epsilon^{pri} \tag{7}$$

$$||s^k||_2 \leq \epsilon^{dual} \tag{8}$$

where $\epsilon^{pri} > 0$ and $\epsilon^{dual} > 0$. The values can be assigned such that:

$$\epsilon^{pri} = \sqrt{n+1}\epsilon^{abs} + \epsilon^{rel} max(||x^k||_2, ||z^k||_2) \tag{9}$$

$$\epsilon^{dual} = \sqrt{n+1}\epsilon^{abs} + \epsilon^{rel}||\rho u^k||_2 \tag{10}$$

where $\epsilon^{abs} > 0$ is an absolute tolerence and $\epsilon^{rel} > 0$ is a relative tolerence. A value for ϵ^{rel} we have choosen is 10e-12 and for ϵ^{abs} is 10e-14.

The first expression of (4) can be solved in parallel on different machines (SVM is fitted to the local data A_i). Machine j loads local data A_j in memory for ADMM iterations. Then each machine uses efficient algorithms (L-BFGS-B in our case) to solve a subproblem with the input being the local data and the broadcasted vectors u and z. After solving the subproblems, all machines aggregate the local solutions and send to the master node for the next round of iterations. We can see that if we have extremely high-dimensional data, e.g., having more than billions of features, would create significant overhead in distributed system due to network bandwidth constraints.

For our numerical experiments it is important to solve the subproblems as efficient as possible, thus let us consider improvements [3] of ADDM that can reduce number of iterations and running time while achieving the same accuracy.

Over-relaxation. x^{k+1} is used to update z^{k+1} and u^{k+1}. The convergence can be improved if we add z^k to the x^{k+1}:

$$x^{k+1} = \alpha x^{k+1} + (1 - \alpha)\, z^k \tag{11}$$

Algorithm 3 MapReduce SVM

```
1: class SVM
2:
3:     method INITIALIZE
4:         RDD rdd = loadDataFromClusters
5:         Initialize z^k, u^k
6:     end method
7:
8:     method MAP(partitionid id, RDD rdd)
9:         for all Array A ∈ Partition rdd[id] do
10:            x_i^{k+1} = argmin_x(1^T(Ax + 1)_+ + (ρ/2)||x − z^k + u_i^k||_2^2)
11:            EMIT(LocalSolution x^{k+1})
12:        end for
13:    end method
14:
15:    method REDUCE(LocalSolutions [x_1^{k+1}, x_2^{k+1}, ..., x_N^{k+1}])
16:        x^{k+1} = concatenate(x_1^{k+1}, x_2^{k+1}, ..., x_N^{k+1})
17:        EMIT(Solution x^{k+1})
18:    end method
19:
20:    method MAIN(RDD rdd)
21:        for i = 1 to MAXITERS do
22:            BROADCAST z^k, u^k
23:            x^{k+1} = rdd.Map.Reduce
24:            Compute z^{k+1}, u^{k+1}
25:            proceed until a convergence criteria satisfied or MAXITERS achieved
26:        end for
27:    end method
28:
29: end class
```

α is the adjustable parameter.

Random permutation of data. The worst case is when the data of only the same class is located on a cluster. If we randomly shuffle the data across clusters it gives significant improvement in terms of number of iterations and execution time.

Warm start in solving subproblem. In our experiments we used Broyden-Fletcher-Goldfarb-Shanno (L-BFGS) algorithm. It requires the initial guess as input parameter. It is not necessary to start always from some fixed values (e.g., zeros). We can use the previous value x^k as an initial guess to obtain x^{k+1}.

Inexact minimization (early stopping) of subproblem. An exact solution of a problem might require a lot of time and iterations. The problem can be solved approximately. Thus, in our case we can limit the maximum number of iterations in L-BFGS-B algorithm.

Let us investigate the advantages of applying these improvements. We consider four

datasets (*dataset1*, *dataset2*, *dataset3*, *dataset4*). Each dataset has equal size (50000), but different amount of features (2,3,4, and 5 correspondingly). All the datasets are seperable by a hyperplane. We assume that the distibuted algorithm is running on 8 machines. In the serial implementation we have an inner loop that runs 8 times on each ADMM iteration (at line number 23 in Algorithm 3 we should have a loop). We take the maximum running time among these 8 iterations and set it as a running time of the ADMM iteration. The machine has an Intel Core I5-2450M (at 2.50GHz) with 8 GB RAM. From the convergence plots (Figure 3) we can see that it takes a prominent amount of time and iterations to get the primal and dual residuals satisfied the convergence criteria. But very high precision can achieved earlier. For example, on the figures Figure 3(a) and Figure 3(b) you can see that on the iteration 113 the error rate is around 0 (0.0098), so the algorithm can be stopped at this point. We can observe similar behavior on figures Figure 3(c) and Figure 3(d). On the figures Figure 3(e), 3(f), 3(g), 3(h) we can see that the low error rate were achieved upon completion of several iterations, but to satisfy the convergence criteria thousands iterations have been carried out. To avoid such behavior we can apply a heuristic. Upon completion of a certain amount of iterations we can compute a training error rate (or validation error rate) on each cluster and if it small enouth, stop the training.

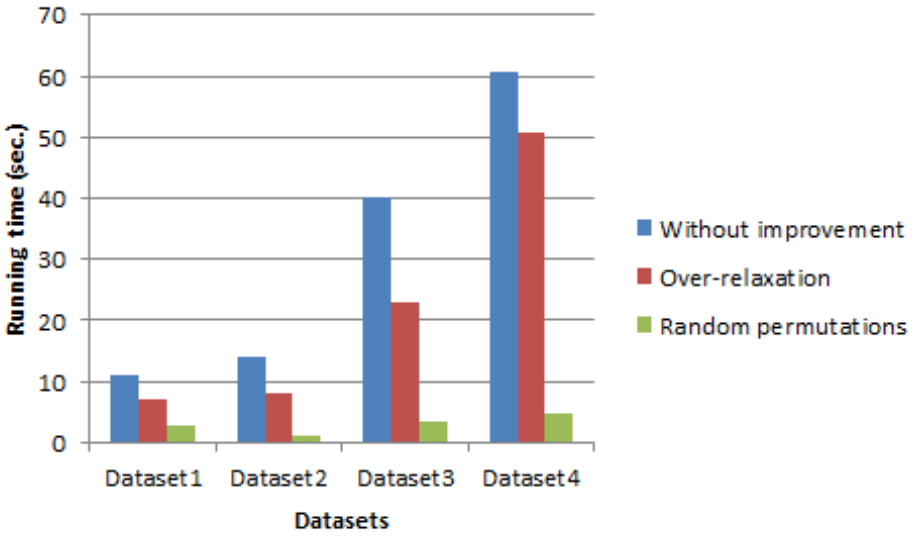
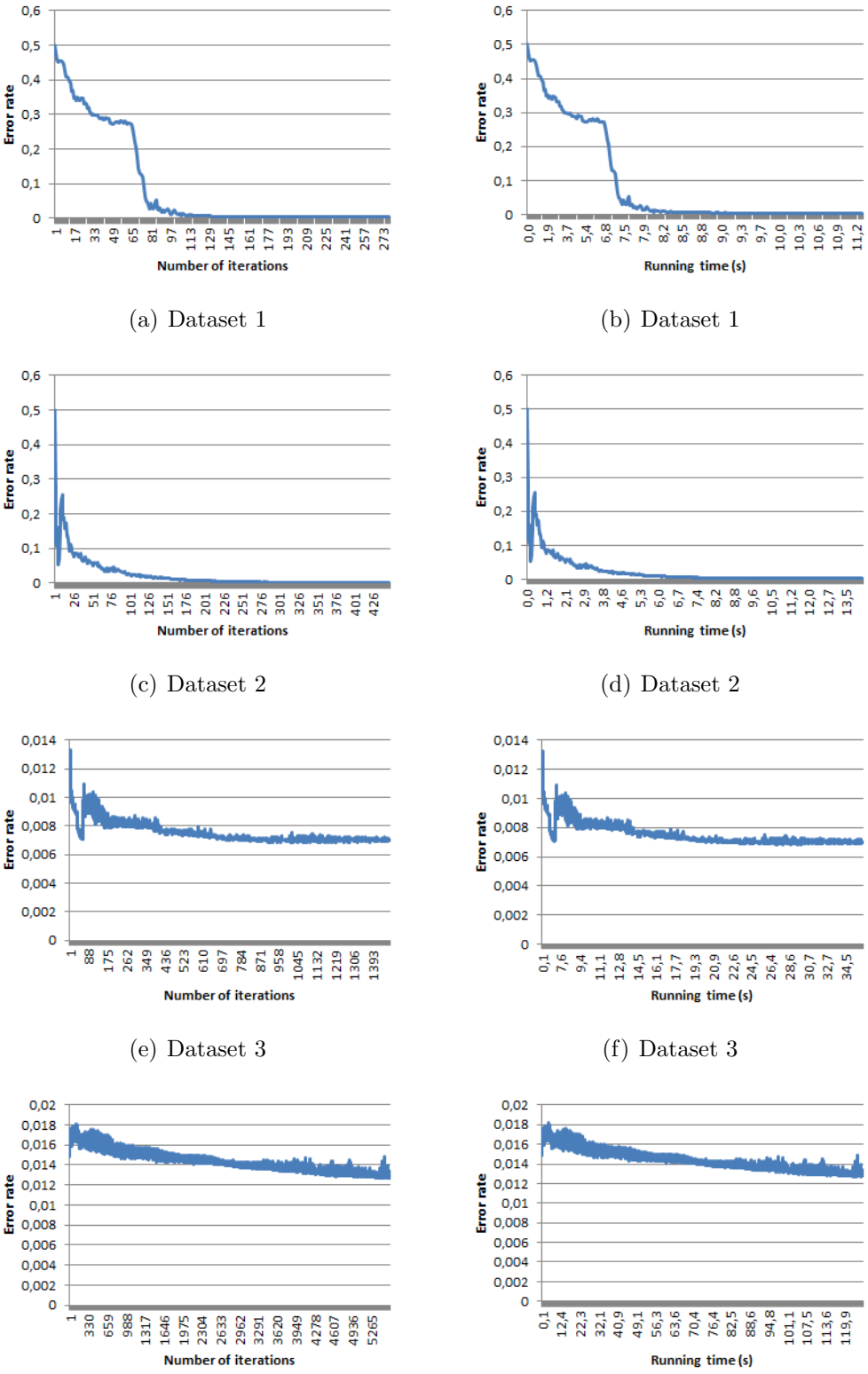


Figure 2: The result of the improvements on different datasets



(a) Dataset 1

(b) Dataset 1

(c) Dataset 2

(d) Dataset 2

(e) Dataset 3

(f) Dataset 3

(g) Dataset 4

(h) Dataset 4

Figure 3: Convergance of four datasets

From the Figure 2 we see that the worst case when only the data of one class located on the cluster. If data of the different classes is shuffled and distributed evenly across the clusters then the convergence is faster. Also, we see that over-relaxation always provides an advantage. The problem with the over-relaxation is that the parameter α has to be fitted. In our experiments we used $\alpha = 1.7$. As reported in [1] usually $\alpha \in [1.5, 1.8]$.
With warm starting we did not achieve convergence of the algorithm. It can be explained by the peculiarities of the L-BFGS-B method.
For the scalability experiments we will use the data consisting of 500000 items and 2 features. We will mesure the speedup in terms of data size and number of clusters.

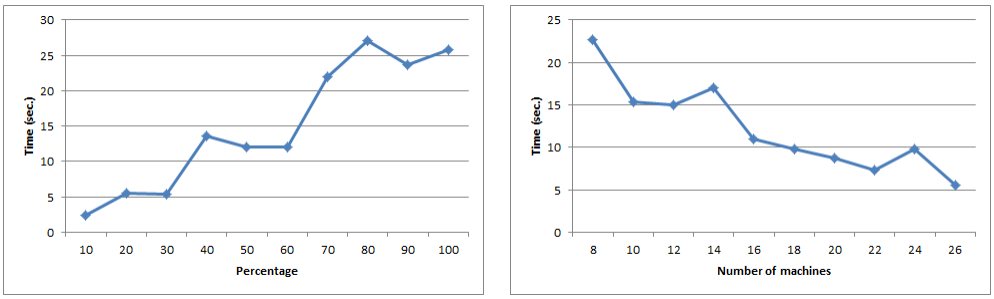


Figure 4: Running time in terms of data size and number of clusters

As we can see from Figure 4 SVM implemented with ADMM can achieve an almost linear speedup.

4.2 Valley filling

Let us consider an application of mapreduce in the optimal electrical vehicles (EV) charging problem. Alternating Direction Method of Multipliers (ADMM) allows formulate the charging task as decentralized optimization algorithm. It means that we can solve optimal fleet charging using a MapReduce framework. As an example, consider valley filling problem.

1. For each EV $i = 1, ..., N_{EV}$:

$$\begin{aligned} x_i^{k+1} &= \underset{x_i}{\text{minimize}}\ \gamma f_i(x_i) + \frac{\rho}{2}||x_i - x_i^k + \bar{x}^k + u^k||_2^2 \\ &\text{subject to } x_i \in \mathbb{X}_i \end{aligned} \tag{12}$$

2. For the aggregator:

$$\begin{aligned} x_N^{k+1} &= \min_{x_N} \delta||D - x_N||_2^2 + \frac{\rho}{2}||x_N - x_N^k + \bar{x}^k - u^k||_2^2 \\ &= \frac{\rho}{\rho - 2\delta}\left(x_N^k - \bar{x}^k + u^k\right) - \frac{2\delta}{\rho - 2\delta}D \end{aligned} \tag{13}$$

3. Incentive signal update:

$$\begin{aligned} \bar{x}^{k+1} &= \frac{1}{N}\sum_{i=1}^{N} x_i^{k+1} \\ u^{k+1} &= u^k + \bar{x}^{k+1} \end{aligned} \tag{14}$$

Here x_i the EVs profiles to be optimized under constraints $\mathbb{X}_i$. Let us consider the implementation in Spark MapReduce framework. The EVs profiles are distributed across the cluster nodes.

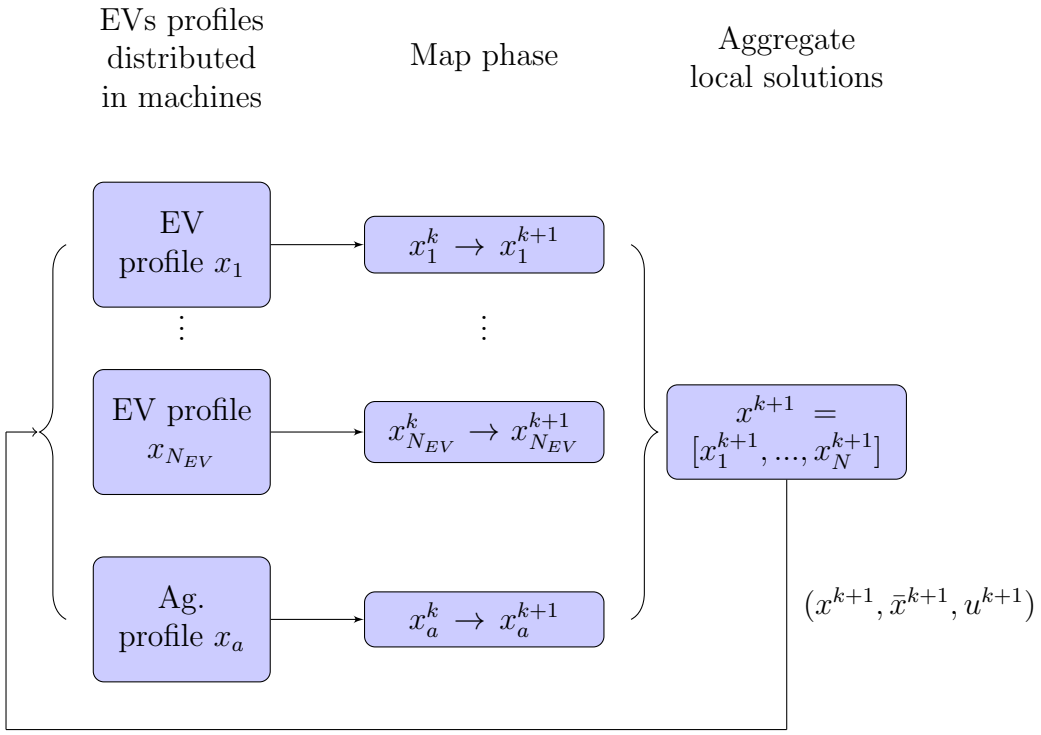


Figure 5: An illustration of distributed Valley Filling algorithm

Each partition of the RDD contains a single EV profile. Then map phases are launched over the RDD partitions (EV profiles) to find an optimal profile for the EV locally. Then on the reduce phase all machines aggregate the profiles, compute incentive signals $\bar{x}$, u, and send to the master node for the next iteration. In our numerical experiments we will show that EV ADMM has a linear computational complexity with respect to the number of agents.

Algorithm 4 Spark, Valley Filling

class VALLEY-FILLING

  method INITIALIZE
    RDD $rdd =$ initialize N_{EV} profiles
    Initialize z^k, u^k
  end method

  method MAP(partitionid id, RDD $evprofiles$, Boolean $is-ev-profile$)
    if $is-ev-profile$ **then**
      $x_i^k = evprofiles[id]$
      $x_i^{k+1} = \underset{x_i}{\text{minimize}}\ \gamma f_i(x_i) + \frac{\rho}{2}||x_i - x_i^k + \bar{x}^k + u^k||_2^2$
      EMIT(EVProfile x_i^{k+1})
    else
      $x_N^k = evprofiles[id]$
      $x_N^{k+1} = \frac{\rho}{\rho - 2\delta}\left(x_N^k - \bar{x}^k + u^k\right) - \frac{2\delta}{\rho - 2\delta}D$
      EMIT(AgProfile x_N^{k+1})
    end if
  end method

  method REDUCE(LocalSolutions $[x_1^{k+1}, x_2^{k+1}, \ldots, x_{N_EV}^{k+1}, x_N^{k+1}]$)
    $x^{k+1} = concatenate(x_1^{k+1}, x_2^{k+1}, \ldots, x_N^{k+1})$
    $\bar{x}^{k+1} = \frac{1}{N}\sum_{i=1}^{N} x_i^{k+1}$
    $u^{k+1} = u^k + \bar{x}^{k+1}$
    EMIT$(x^{k+1}, \bar{x}^{k+1}, u^{k+1})$
  end method

  method MAIN(RDD rdd)
    for $i = 1$ to $MAXITERS$ **do**
      BROADCAST z^k, u^k
      $x^{k+1} = rdd.Map.Reduce$
      Compute z^{k+1}, u^{k+1}
      proceed until a convergence criteria satisfied or MAXITERS achieved
    end for
  end method

end class

From the picture (Figure 6) we see that the algorithm is scalable. We used MATLAB environment for simulations. As optimization method we used CVXGEN which gives greater performance than L-BFGS-B (CVXGEN is faster since it can be compiled into C code).

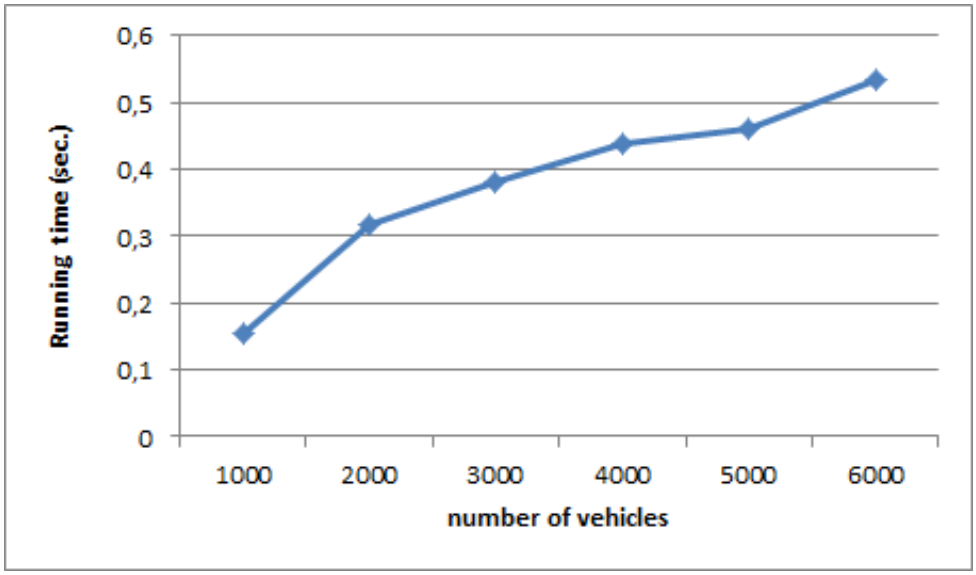


Figure 6: Running time of the valley filling problem

5 Conclusion

In this work we investigated and compared the implementations of iterative algorithms in MapReduce frameworks. Reproduced state-of-the-art PageRank, implemented SVM, and Valley Filling problem in SPARK using ADMM. In comparison with Hadoop SPARK does not require to define the path to the input and output data on each iteration. Thus, owing to RDD abstraction, a programer does not need to care about data location. ADMM does not give a speedup in runnning computations and can be only applied in the case when data can not fit in memory or when it is not allowed to transfer the data to the central node in a cluster because of privacy issues. If increasing running time of a machine learning task in the main concern then it is recommended to use PyLearn library which has already implemented ML algoritms in CUDA providing fine-grained parallelism. ADMM is targeted toward cluster systems.

We considered ADMM improvements that can give a significant increase in performance. For example, from experiments we see that if data uniformly located on each cluster then the convergence time is less than if the data of only one class is presented on each cluster. If over-relaxation parameter is manually fitted well then it also gives a significant advantage. With warm start improvement we did not achieve the convergence.

In the case of linear classification ADMM convergence criteria is not efficient, e.g., when the classification error rate is low the algorithm still does not satisfy the stopping criteria. The execution time can be reduced using a heuristic. While running a classification task, upon completion of predefined amount of iterations training error rate (or validation error rate) can be calculated and if the classification precision is high enough then stop the training.

In our experiments we used a serial implementation that performs the update of x_i se-

quentially. The reported time is the maximum time required to solve a subsystem. It means that we do not take into account the latency, time required to keep foult-tolerence (recovery from failures and etc.) and time for master-slaves coordination in a distributed system. We got a linear speedup in running time, but it necessary to check, how it works on real cluster because above mentioned problems.

For further research it will be worth to investigate the performance on a cluster system, consider the execution with the presence of failures and how much time it takes to recover from these failures.

Also, it is interesting to consider in more details relatively new GraphLab framework. It provides more flexible API (in comparision with MapReduce), supports multicore computantions and it also shows significant speedup over Hadoop in running iterative algorithms. At the mom Currently there is no running time comparisons of state-of-the-art algorithms in SPARK and GraphLab.

References

[1] S. Boyd, N. Parikh, E. Chu, B. Peleato, and J. Eckstein. Distributed optimization and statistical learning via the alternating direction method of multipliers. *Foundations and Trends in Machine Learning,* 3:1122, 2011.

[2] J. Grauzam, B. Prodhomme, J. Reilly. Cache-Reduce: Implementing Collaborative Filtering on a Cluster with the Spark Framework, 2012.

[3] C. Zhang, H. Lee, and Kang G. Shin. Efficient Distributed Linear Classification Algorithms via the Alternating Direction Method of Multipliers.In *Proceedings of the 15th International Conference on Artificial Intelligence and Statistics (AISTATS), JMLR WCP 22,* 2012.

[4] M. Kraning, E. Chu, J. Lavaei, and S. Boyd. Message Passing for Dynamic Network Energy Management, 2012.

[5] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, I. Stoica. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. In *NSDI'12 Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation,* 2-2, 2012.

[6] P. A. Forero, A. Cano, G. B. Giannakis. Consensus-Based Distributed Support Vector Machines. In *Journal of Machine Learning Research,* 11:1663-1707, 2010.

[7] D. Borthakur. The Hadoop Distributed File System : Architecture and Design. In *Journal of Machine Learning Research,* 2007.

[8] J. Dean, S. Ghemawat. Mapreduce: simplified data processing on large clusters. In *Proceedings of the 6th conference on Symposium on Opearting Systems Design Implementation,* 6:1010, 2004

[9] R. Lammel. Googles MapReduce programming model Revisited. In *Science of Computer Programming,* 70:1-30, 2008

[10] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, C. Kozyrakis.Multicore and Multiprocessor Systems. *Symposium on High Performance Computer Architecture (HPCA),* 2011.