A Distributed Computing Framework for Machine Learning Applications

Yongbiao Ai
Department of Computer Science & Engineering
The Chinese University of Hong Kong
ybai@cse.cuhk.edu.hk

ABSTRACT

I implemented a distributed computing framework for machine learning applications. The framework is a parameter server like system which distributed the data and workloads over worker nodes, and the model parameters are over the server nodes, represented by the vectors or matrices. This framework is able to handling the asynchronous request between different machine nodes and support fault tolerance.

To demonstrate the performance and the fault tolerance for the framework, I show the experimental results on the popular machine learning algorithm with gigabyte real data.

1. INTRODUCTION

Over the past decades, the data generation have reached a extreme level. With the technology evolution such as mobile Internet and IoT, massive and many different types of data have been generated. We are surrounded with the data at the moment, for every seconds, mammoth amount of data is being generated all over the world.

Finding a solution for processing and storing the **Big Data**, known with the 3V characteristic: Volume, Velocity and Variety, is a critical issue in research and industry companies. The typical solution (Figure. 1) for Big Data from bottom to top is: Data Collection, Data Storage, **Data Processing** and graphical user interface (GUI). Data is usually generated by electronic devices such as smartphone, computer and other intelligent hardware, these devices will sent regular report to the centralized server, which then preprocess these data and save it into data storage system for safety. Most type of these data will then sent to data processing system by chunk for batch processing. After data was being analyzed, the result will be used to improve commercial strategy for the customers.

Machine Learning algorithms are popular and useful tool-kits for data processing, it can take large amounts of data and generate insights to help the organization. Some of the common applications of machine learning include following: Web Search, spam filters, recommender system and fraud detection. Distributed computing systems like Hadoop MapReduce [1], Apache Spark and FlumeJava [2] have been widely used in the industry, however, none of them were design for machine learning algorithms. Realistic quantities of training data is massive, and as the model complexity increased, model parameters saved in a single Hadoop or Spark node usually will increate the computational workloads and the communication cost.

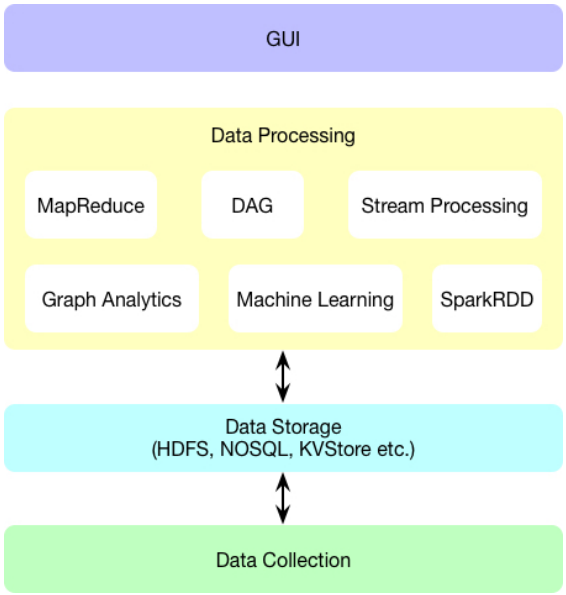

Figure 1. Big Data typical solution

Parameter Server [3] is a popular distributed machine learning framework in recent years, it mainly divide machine nodes into worker node and server node, while the training data and workloads are distributed over worker nodes, and model parameters are maintained by server nodes. The framework manages data communication between different nodes, provide data consistency, data partition and fault tolerance to developers.

In this paper, I conduct a systematic study of existing optimization methods and their relationship between different machine learning algorithms. I implemented a general algorithmic, distributed machine learning framework, which is designed to implement various combinations machine learning algorithms based on gradient optimization method. I made following contributions:

- A minimum Parameter Server framework for distributed machine learning data processing. The framework supporting work-server nodes interaction based on ZMQ [19] using various consistency model such as BSP, ASP and SSP, it also implement data partition strategy for load balancing, multi-task schedul-

ing on working node and vector/map data calculation for model parameters.

- A efficient fault tolerance mechanism with checkpointing, heartbeat detecting and reload data from HDFS on the framework, the system must not require a full restart of a long-running computation. I give a comprehensive analysis of existing fault tolerance implementation and their comparison on the state-of-art distributed computing systems.

- Multiple machine learning algorithm applications with implementation of various existing optimization methods under the framework. A extensive evaluation of the performance of applications and framework.

In the following sections, I first present the background on the evolution of the distributed computing systems, discuss why the parameter server is a efficient framework for machine learning applications (Section 2). Then I illustrate different mathematical optimization methods that can be used in for machine learning application implementation, and show the difference between them (Section 3). And I demonstrate the core components in the framework and how it was designed (Section 4). The applications that have been implemented on the framework will be introduced (Section 5). The evaluation will demonstrate the performance of the framework with other computing systems, and the influence of fault tolerance on the iteration time (Section 6). Followed by the future works in the next semester (Section 7) and conclusions (Section 8).

2. BACKGROUND

In this section, I briefly introduce the the evolution of the distributed computing systems, including state-of-art system such as Hadoop MapReduce, FlumeJava and Spark. I give a comprehensive analysis on what is the parameter server, how it works with machine learning optimization methods and why it is suitable for machine learning data processing.

2.1 Distributed Processing Systems

With the Google **MapReduce** purposed, it had a far-ranging impact on the distributed computing industry, the MapReduce illustrate a concept that some type of data processing procedures could be divided into mapping and reducing phases, the developer could easily overcome the massive data processing difficult. However, the MapReduce have several fatal defects, which make the system cannot run efficiently on particular type of jobs. It have following drawbacks:

- **No Caching** : MapReduce cannot cache the intermediate data in memory for a further requirement, which will diminishes the performance such as iterative tasks (These tasks need each output of the previous task be the input of the next stage). HaLoop and SparkRDD have make improvements on this, as them will accesses data from RAM instead of disk,

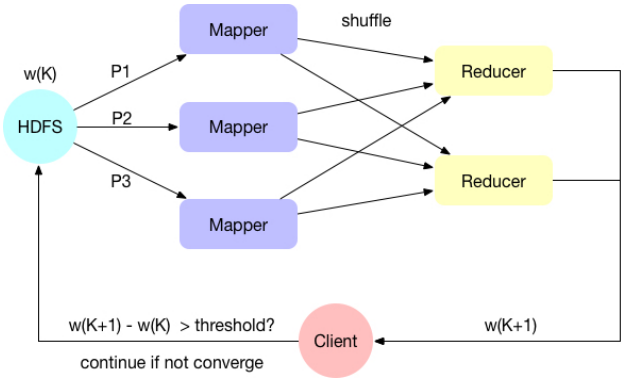

Figure 2. Hadoop MapReduce have no caching function for the iterative task. The intermediate data have to be stored into DFS and reload in the next Job.

which dramatically improves the performance of iterative algorithms that access the same dataset repeatedly.

- **Slow Processing Speed** : When MapReduce process large datasets with different tasks, it will requires a lot of time to perform map and reduce functions, thereby increasing latency. This can be sloved by Dryad [4] and FlumeJava based on the Directed Acyclic Graph (DAG), which use a graph holds the track of operations. DAG will converts logical execution plan to a physical execution plan, which helps in minimize the data shuffling all around and reduce the duration of computations with less data volume, eventuallly increase the efficiency of the process with time.

- **No Real-Time Data Processing** : Hadoop MapReduce is designed for batch processing, which means it take a huge amount of data in input, process it and produce the output. Although batch processing is very efficient for processing a high volume of data, but the output can be delayed significantly. Which will cause the MapReduce is not suitable for Real-time data processing. Naiad purposed a timely dataflow computational model, which support continuous input and output data. It emphasizes on the velocity of the data and it can be processed within a small period of time.

- **Support for Batch Processing Only** : Hadoop MapReduce only support batch processing, it does not able to process streamed, graph and machine learning data, hence overall performance is slower. Husky have purposed a unified framework, which support different kind of tasks with multiply purposes. Which can achieve high performance and support user firendly API among C++, python and Scala.

2.1.1 In Memory Processing

Hadoop MapReduce is not designed for iterative task like Machine Learning algorithm K-Means. (Figure. 2) Every intermediate data will have not saved into persistent

storage. **HaLoop** [5] is a great extension for Hadoop as it provides support for iterative application. In order to meet these requirement, several main changes that are made in Hadoop to efficiently support iterative data analysis. It provides a new application programming interface to simplify the iterative expressions and an automatic generation of MapReduce program by the master node using loop control module until the loop condition is met. Also, it create a new task scheduler that supports data locality in these application in order to efficiently perform iterative operations.

RDDs [6] stands for "Resilient Distributed Datasets", it is the fundamental data structure of Apache Spark. RDD in Apache Spark is an immutable collection of objects which computes on the different node of the cluster. As Hadoop MapReduce makes the iterative computing such as Logistic Regression, K-Means and PageRank slower. Although HaLoop guarantee faster computing with caching extension, the fault tolerance and other questions still exist. RDDs try to solve these problems by enabling fault tolerant distributed In-Memory. Compare with HaLoop, when the worker node in Spark goes down, the system will use Lineage, a track of all the transformations that has to be applied on that RDD including from where it has to read the data, to re-compute the lost partition of RDD from the original one.

2.1.2 DAG Processing

Hadoop MapReduce restricts all computations to take a single input set and generate a single output set, which will cause extra overhead in solving tasks with multiply stages.

In **Dryad** [4], each job will be represented with a **Directed Acyclic Graph** (DAG), the intermediate vertices were written to channels, and more operation than map and reduce will be used, such as join and distributed. With data-flow, the developer do not need to worry about the global state of the computing system, just need to write simple vertices that maintain local state and communicate with other vertices through edges. Compare with DAG, MapReduce is just a simple form of data-flow, with two types vertices: the mapper and the reducer. Compare with MapReduce, Dryad provides explicit join, combines inputs of different types, developers are able to understand and run complex jobs faster.

FlumeJava [2] is a library used in Google, based on expressive and convenient small set of composiable primitives. By using deferred evaluation and optimization, the API could automatically transformed into an efficient execution plan. It is also a run-time system that executing optimized plans, which can transform logical computations info efficient programs. Operations in FlumeJava are executed lazily using deferred evaluation, in order to enable optimization, each PCollection object is represented internally either in deferred or materialized state. When a operation like parallelDo() is called, it just create a parallelDo deferred operation object and return a new deferred PCollection points to it. The result of executing a series operations is thus a DAG of deferred PCollections and operations, it is also called the execution plan. In order to actually trigger the evaluation of a series of parallel operations, the developer need to call FlumeJava.run(), this will first optimizes the execution plan and visits each of the deferred operations in the optimized plan. When a deferred operation is evaluated, it will convert its result PCollection into a materialized state, FlumeJava will automatically deletes any temporary intermediate files.

2.1.3 Stream Processing

Hadoop MapReduce was designed to support batch processing, it is not suitable for streaming data processing. Stream processing should enable users to query continuous data stream and detect conditions fast within a small time period from the time of receiving the data, the detection time period may vary from few milliseconds to minutes.

The **Naiad** [7] project is an investigation of data-parallel data-flow computation like Dryad, but with a focus on the low-latency streaming and cyclic computations. It introduces a new computational model called , which combines low-latency asynchronous message flow with lightweight coordination when required. Naiad's most notable performance property, when compared with other data-parallel data-flow systems, it its ability to quickly coordinate among the workers and establish that stages have completed. Naiad support efficient implementations of a variety of programming patterns, including nested iterative algorithms and incremental updates to iterative computations.

Popular stream processing framework includes Spark Streaming, Storm and Flink.

2.1.4 General Purpose Platform

System like Hadoop and Spark have been widely adopted for big data processing, however, sometimes over-simplified API stop developers from more find-grained control and designing more efficient algorithms, but using sophisticated DSLs may result in development cost and bug-prong programming.

A general research platform called **Husky** [8] is able to help developers implement applications of different characteristics, for example, coarse-grained and fine-grained, iterative and non-iterative, synchronous and asynchronous workloads, and achieves performance close to or better than specialized systems and programs.

2.2 Parameter Server

Many machine learning problems reply on large amounts of data for training, companies nowadays training algorithms with terabytes or petabytes of data, and create models out of it. Such models consist of weights that will optimize for error in inference for most cases. The number of weights/parameters run into orders billions to trillions. In such big model, learning on a single machine is not possible. It is useful to have a framework that can be used for distributed learning as well as inference.

Parameter Server have been purposed for solving machine learning algorithms such as LDA and L-BFGS efficiently. The history of the development of parameter server:

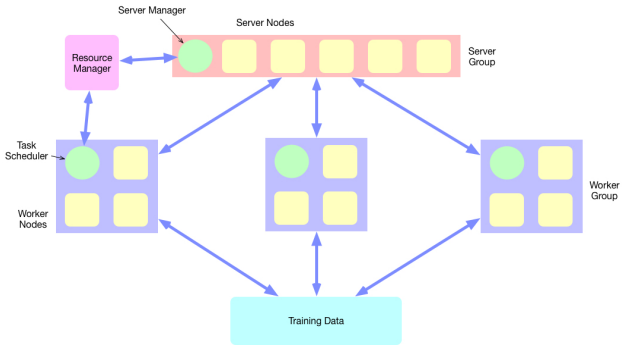

Figure 3. Parameter Server Architecture; Server nodes store model parameters; Worker nodes maintains and computes training data; Server manager maintain meta data of server nodes; Task scheduler allocate jobs and monitor the worker nodes' process.

- In 2010, Alex Smola purposed a parallel-LDA computing framework, which is the first generation parameter server, which use memcached as the parameter storage system. It can successfully training LDA model in parallel, but still lack of efficiency and flexibility.
- Jeff Dean from Google purposed the second generation parameter server called DistBelief [9], which stores massive deep learning model parameters into the global parameter server nodes. It efficiently solve the SGD and L-BFGS algorithm training problem in parallel.
- Mu Li purposed the third generation parameter server called **ps-lite** , which is a more general platform support flexible consistency models, elastic scalability, and continuous fault tolerance.

Ps-lite [3] is the third generation (Figure. 3) , lightweight and efficient implementation of parameter server framework. It provides clean and powerful APIs for model parameter query and updates:

- Push(keys, values): Push a list of key-value pairs to the server nodes
- Pull(keys): Pull the values from servers for a list of keys
- Wait(): wait until a push or pull finished

2.3 Related Work

Apache Mahout is a project under the Apache Software Foundation, which is machine learning tool-kits base on Apache Hadoop and uses the MapReduce paradigm. Mahout implements popular, scalable and distributed machine learning algorithms that are focused in the areas of clustering, collaborative filtering and classification. It contains Java libraries for common math algorithms and operations that focused on statistics, linear algebra and primitive Java collections. However, most machine learning algorithms adopt an iterative-convergence approach, the Hadoop MapReduce system will lead those algorithm operations cost high computation and cause extra communication overheads, as a result, Mahout is not able to provide high efficiency.

Bösen [10] is a data-parallel distributed key-value store parameter server system, it is also a critical component in Petuum. Bösen is not work purely asynchronously, it adopt a consistency model called SSP, which can provide better performance than a complete synchronous model. The Machine Learning applications running on the system was linked with client libraries to update model parameters, the libraries in the run-time will launch a group of background threads to synchronized the local parameters with other servers, and also launch multiple user threads to execute algorithm logic on training data. Bösen also adopt leaky bucket to reduce network bandwidth consumption.

Angel [11] is a flexible and powerful parameter server jointly developed by Tencent and Peking University, it have wide applicability and stability after repeatedly tuned under the massive data from Tencent. The core design philosophy revolves around the model, it rationally splits high-dimensional large models into multiple parameter server nodes, and easily implements various efficient machine learning algorithms through efficient model update interfaces and arithmetic functions, as well as flexible synchronization protocols. Since Angel was developed with Java and Scala, it can be scheduled directly on the Yarn platform.

The parameter server architecture is also used in popular deep learning systems such as MXNet and TensorFlow as the distributed management module. Low-level communication efficiency between worker and server were focused on the normal parameter server systems, while deep learning systems concentrate more on high-level abstraction for users to build a data-flow graph, and also focus on graph execution optimization.

In conclusion, among the state-of-art works of distributed computing systems, from Hadoop MapReduce, HaLoop, Spark to Naiad, these system have their advantages in linear data processing, In-Memory processing or processing based on DAG, however, the parameter server systems are the most efficient for large-scale machine learning algorithms. The popular PS system such as Bösen and PsLite focus more on the scalability, fault tolerance and support of various consistency protocols.

3. MATHEMATICAL OPTIMIZATION

Mathematical Optimization has become the heart of Machine Learning. As each ML algorithm has three core components: Representation, Evaluation and Optimization. Usually representation is done once with some parameter values, then the data should evaluated using the parameters, optimization actions will iterate until a certain condition is met based on the evaluation result. Among various optimization techniques, gradient-based algorithms have been extensively studied and are highlighted by their wide applicability in different machine learning problems and their iterative properties. In order to design and implement a distributed machine learning framework that allows flexible combination of machine learning algorithms and gradient-

base optimization methods, it is necessary to understand optimization methods and how they are applied to machine learning problems.

3.1 Basic Gradient Descent Methods

In the calculus, the partial derivative of the parameter of the multivariate function is obtained, and the partial derivative of each parameter obtained is written in the form of a vector, which is a gradient. In the machine learning algorithm, when the loss function is minimized, it can be solved step by step by the gradient descent method to obtain a minimized loss function and model parameter values. Conversely, if we need to solve the maximum value of the loss function, we need to iterate by the gradient ascent method. The gradient descent method and the gradient descent method are mutually transformable. For example, we need to solve the minimum value of the loss function $f(\theta)$, then we need to use the gradient descent method to iteratively solve. But in fact, we can reverse the maximum value of the loss function $-f(\theta)$, and the gradient rise method comes in handy.

To find the local minimum of a function using the gradient descent method, iterative search must be performed to the specified step distance point in the opposite direction of the gradient corresponding to the current point on the function. There are some definition:

- Learning Rate: Determines the length of each step along the negative direction of the gradient during the iterative process of the gradient descent.

- Feature: The input part of the sample, such as two single feature samples $(x_0, y_0), (x_1, y_1)$, then the first sample feature is x_0, The first sample output is y_0.

- Hypothesis function: In supervised learning, the hypothesis function used to fit the input sample is denoted by $h_\theta(x)$. For example, for m samples of a single feature $(x_i, y_i)(i = 1, 2, ...m)$, the fitting function can be used as follows: $h_\theta(x) = \theta_0 + \theta_1 x$.

- Loss function: In order to evaluate the fit of a model, the loss function is usually used to measure the degree of fit. The loss function is miniaturized, which means that the degree of fitting is the best, and the corresponding model parameters are the optimal parameters. In linear regression, the loss function is usually the square of the difference between the sample output and the hypothesis function. For example, for m samples $(x_i, y_i)(i = 1, 2, ...m)$, using linear regression, the loss function is:

$$J(\theta_0, \theta_1) = \sum_{i}^{m} (h_\theta(x_i) - y_i)^2$$

  Where x_i represents the i-th sample feature, y_i represents the output of the i-th sample, and $h_\theta(x_i)$ is a hypothesis function.

- Iterative update parameters: Let α and $J(\theta)$ be the learning rate and loss function, then the parameters update as follows:

$$\theta_{t+1} = \theta_t - \alpha \frac{\partial J(\theta)}{\partial \theta}$$

3.1.1 Batch Gradient Descent

Batch Gradient Descent (BGD) is the most primitive form of the gradient descent method. The specific idea is to use all the samples to update each parameter. The mathematical form is as follows:

$$\theta_{t+1} = \theta_t - \alpha \frac{1}{m} \sum_{i=1}^{m} (h_\theta(x^i) - y^i) x_t^i$$

It can be noticed from the above formula that it obtains a global optimal solution, but each step of the iteration uses all the data of the training set. If the number of samples m is large, calculating each θ value requires traversing to calculate all the samples. When the amount of data is very time consuming.

3.1.2 Stochastic Gradient Descent

Stochastic Gradient Descent [12] (SGD) method is similar to the principle of the batch gradient descent method. The difference between the gradient and the gradient is not used for all m samples, but only one sample j is used to find the gradient. The corresponding update formula is:

$$\theta_{t+1} = \theta_t - \alpha (h_\theta(x^i) - y^i) x_t^i$$

Compared to the batch gradient descent method, the two algorithms are two extremes, one using all data for gradient descent and one for one sample for gradient descent. Their respective advantages and disadvantages are naturally very prominent. For the training speed, the random gradient descent method is iterative because it only uses one sample at a time, and the training speed is not satisfactory when the sample gradient is large. For accuracy, the stochastic gradient descent method is used to determine the direction of the gradient with only one sample, making the solution most likely not optimal. For the convergence speed, since the random gradient descent method iterates one sample at a time, the iterative direction changes greatly and cannot converge to the local optimal solution very quickly.

3.1.3 Mini-batch Gradient Descent

The **Mini-batch gradient descent** (MBGD) method is a compromise between the batch gradient descent method and the stochastic gradient descent method, that is, for m samples, we use x samples to iterate, where 1 <x <m. Generally, x=10 can be taken. Of course, according to the data of the sample, the value of this x can be adjusted. The corresponding update formula is:

$$\theta_{t+1} = \theta_t - \alpha \frac{1}{m} \sum_{k=i}^{i+m-1} (h_\theta(x^k) - y^k) x_t^k$$

3.2 Accelerated Methods

One disadvantage of the normal gradient descent method is that its update direction is completely dependent on the gradient calculated by the current batch and is therefore very unstable. Momentum and Nesterov accelerated gradient methods are two widely used methods to accelerate SGD convergence by reducing oscillations and accumulating useful parameter displacements.

3.2.1 Momentum

The **Momentum** [13] algorithm borrows the concept of momentum in physics, which simulates the inertia of an object's motion, that is, it retains the previously updated direction to some extent when updating, and uses the current batch's gradient to fine tune the final update direction. In this way, stability can be increased to a certain extent, so that learning is faster, and there is a certain ability to get rid of local optimum:

$$m_t = \gamma m_{t-1} + \alpha \bigtriangledown_\theta J(\theta)$$

$$\theta_{t+1} = \theta_t - m_t$$

The Momentum algorithm observes the past gradient m_{t-1}. If the direction of the current gradient is consistent with the historical gradient (indicating that the current sample is unlikely to be an abnormal point), the gradient in this direction will be enhanced. If the current gradient is inconsistent with the historical ladder direction, the gradient will be attenuated. However, when the model approximates local optimum, the momentum method has a convergence problem because the calculated gradient does not respond to parameter updates due to momentum and causes the learning rate to be too large.

3.2.2 Nesterov Accelerated Gradient Method

The **Nesterov Accelerated Gradient** [14] (NAG) method consists of a gradient descent step followed by something that looks like a momentum item, but not exactly the same as the classic momentum. I am calling it the "momentum stage" here. It is worth noting that the parameter that NAG minimizes is given the symbol γ by Sutskever, not θ. You will see that θ is the updated parameter symbol for the gradient descent phase, but before the momentum phase. Here are two phases:

$$m_t = \gamma m_{t-1} + \alpha \bigtriangledown_\theta J(\theta - \gamma m_{t-1})$$

$$\theta_{t+1} = \theta_t - m_t$$

Compared to the momentum method, NAG differs in the order of calculation. In each iteration, NAG first applies the momentum term to the parameter and then calculates the gradient based on the new parameter, while the momentum method first calculates the gradient using the old parameters before applying the momentum. Therefore NAG provides better convergence.

3.3 Adaptive Learning Rate Methods

Researchers have long recognized that learning rates are certainly one of the hyper-parameters that are difficult to set because they have a significant impact on the performance of the model. Losses are usually highly sensitive to certain directions in the parameter space and are not sensitive to others. Momentum algorithms can alleviate these problems to some extent, but at the cost of introducing another hyper-parameter. In this case, naturally there is no other way to ask. If we believe that the direction sensitivity is axis-aligned to some extent, then it is reasonable to set different learning rates for each parameter and automatically adapt to these learning rates throughout the learning process. The Delta-bar-delta algorithm [15] is an early heuristic method for adapting the learning rates of model parameters during training. The method is based on a very simple idea that if the loss keeps the same sign for the partial derivative of a given model parameter, then the learning rate should increase. If the sign for the parameter changes the sign, then the learning rate should be reduced. Of course, this method can only be applied to full-batch optimization. Recently, some incremental (or small batch based) algorithms such as AdaGrad, RMSProp and Adam have been proposed to adaptive the learning rate of model parameters.

3.3.1 AdaGrad

The **AdaGrad** [16] algorithm is capable of independently adapting the learning rate of all model parameters, scaling each parameter inversely proportional to the square root of the sum of all its gradient history squared values. The parameter with the largest partial derivative of loss has a rapidly decreasing learning rate, while the parameter with small partial derivative has a relatively small decrease in the learning rate. The net effect is that a more gradual tilt in the parameter space will result in greater progress.

$$G_{t,i} = G_{t-1,i} + (\bigtriangledown_\theta J(\theta_t, i))^2$$

Where θ_t, i is the i-th parameter in iteration t, and $\bigtriangledown_\theta J(\theta_t, i)$ is the gradient of the objective function relative to the i-th parameter in iteration t. The update formula for the adaptive learning rate containing each parameter is:

$$\theta_{t+1,i} = \theta_{t,i} - \frac{\alpha}{\sqrt{G_{t,i} + \varepsilon}} \bigtriangledown_{\theta_t} J(\theta_{t,i})$$

Where α denotes a general learning rate, and ε is a smoothing factor to avoid division by zero.

3.3.2 RMSprop

The **RMSProp** algorithm [17] is based on AdaGrad modifications to make it better for non-convex settings. The variable gradient is accumulated as an exponentially weighted moving average. AdaGrad is designed to quickly converge when applied to convex problems. When applied to a non-convex function training neural network, the learning trajectory may pass through many different structures and eventually reach a region where the local convex is a bowl.

AdaGrad shrinks the learning rate based on the entire history of the squared gradient, which may make the learning rate too small before reaching such a convex structure. RMSProp uses exponential decay averaging to discard the history of the distant past, enabling it to converge quickly after finding the convex structure.

$$G_{t,i} = \gamma G_{t-1,i} + (1-\gamma)(\bigtriangledown_{\theta_t} J(\theta, i))^2$$

Compared to AdaGrad, the use of moving average introduces a new hyper-parameter γ, which is used to control the length range of the moving average. The update stage is same as AdaGrad.

$$\theta_{t+1,i} = \theta_{t,i} - \frac{\alpha}{\sqrt{G_{t,i} + \varepsilon}} \bigtriangledown_{\theta_t} J(\theta_{t,i})$$

3.3.3 Adam

Adam [18] is another learning algorithm for learning rate adaptation. The name "Adam'" is derived from the phrase "adaptive moments". In the context of early algorithms, it might be best to think of a combination of RMSProp and momentum with some important differences. First, in Adam, momentum is directly incorporated into the estimate of the gradient first moment. The most intuitive way to add momentum to RMSProp is to apply momentum to the scaled gradient. There is no clear theoretical motivation for combining momentum with scaling. Second, Adam includes an offset correction that corrects the estimates of the first and second moments initialized from the origin. RMSProp also uses second-order moment estimation, but the correction factor is missing. Therefore, unlike Adam, the RMSProp second-order moment estimate may be highly biased at the beginning of training. Adam is generally considered to be quite robust to the selection of hyper-parameters, although the learning rate sometimes needs to be modified from the suggested defaults.

$$g_t = \bigtriangledown_\theta J(\theta)$$
$$m_t = \beta_1 m_{t-1} + (1-\beta_1) g_t$$
$$v_t = \beta_2 v_{t-1} + (1-\beta_2) g_t^2$$

Where m_1 is the first moment of iteration t, v_t is the second moment, and β_1 and β_2 are the corresponding attenuation factors. When the decay rate is small (β_1 and β_2 are large), the moment is biased toward the initial value. The revised estimate is used to resolve this issue in the form:

$$\hat{m}_t = \frac{m_t}{1-\beta_1^t}, \hat{v}_t = \frac{v_t}{1-\beta_2^t}$$

Parameters are updated iteratively as follows:

$$\theta_{t+1} = \theta_t - \alpha \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \varepsilon}$$

4. ARCHITECTURE AND IMPLEMENTATION

In this section, I introduce architecture of the framework, I also presents the programming model, data parallelism, request handling, consistency control and implementation of fault tolerance in the framework.

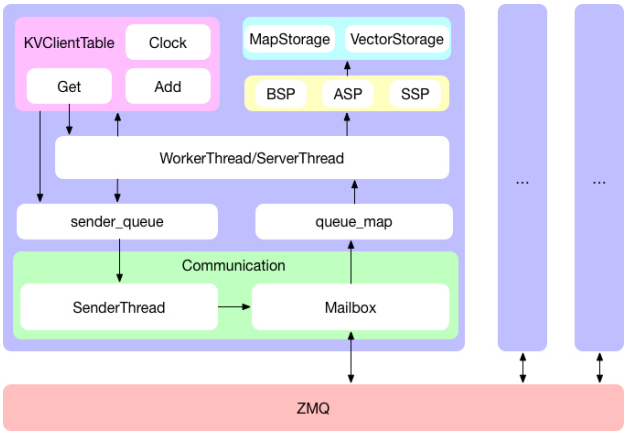

Figure 4. System Architecture; Multiple machines interconnected with each and form a parameter server cluster. WorkerThread/ServerThread plays the worker and server role in parameter server; Mailbox provides inter-process communication services.

4.1 System Architecture

The framework adopts a typical master-slave architecture, as shown in Figure. 4. Each machine nodes running on cluster was communicated with each other using ZeroMQ library via TCP/IP protocol, each nodes have a Engine module responsible for start, run and stop the global services. The KVClientTable can be called by developer with different operations such as Get, Add and Clock. The role of worker and server are different from the typical parameter server such as ps-lite, I implement workers and servers with threads rather than processes in different machines. On the one hand, this can allow multiple worker threads on the same machine able to share the loaded data and avoid repetitive pull of the same model parameters, on the other hand, the thread based design made starting or killing workers or servers cheaper than processes based system.

There are a brief introduction about components in the framework:

- **Engine** : The core manager in the framework, interconnected with other module such as Mailbox, Model and Server/WorkerThread in the framework.
- **KVClientTable** : Provide core parameter server APIs such as Get, Add and Clock. Other interfaces like CheckPoint and HeartBeat for fault tolerance, Send for communication.
- WorkerThread: Sets the callback when KVClientTable.Get() is called. When ServerThread gets the data successfully and sends it back, it will send the callback to return the data to KVClientTable.
- ServerThread: The service thread running in the node, providing the response service of Add, Get, Clock.
- SenderThread: A service used to provide communication information between sending nodes, a stand actor-mailbox model is used.

- PartitionManager: Provides data partition services according to number of nodes in the cluster for data parallel computing.

- IdMapper: Allocate worker and server threads' id based on the node configuration. There are maximum 1000 threads running on each machines, the server threads id are range from 0 to 50, while the worker threads id are range from 50 to 100. The id is critical for the communication between different worker and servers.

- Communication Pattern: In the process of communication, the id of sender and receiver is an important identifier, and all ids are allocated according to the id of the machine node to ensure uniqueness. Each id is associated with each WorkThread, ServerThread queue, and the id is used to find the queue of the corresponding thread.

- **Mailbox** : Encapsulates a set of interfaces for sending and receiving information from ZMQ. By maintaining a map to insert messages into the queues of WorkThread and ServerThread, a typical producer-consumer model.

- Message: The basic data element for communication, it contains important information such as flag that indicate the message's type, the sender and receiver's node id, and the updated parameter data.

- Consistency Model: There are three core operations Add, Get and Clock respectively. The first two are used to update and acquire the model parameters. The latter one is to facilitate the synchronization of each node to ensure data consistency. The three commonly used models are BSP, ASP, SSP.

- ProgressTracker: The model parameters clock progress on different machines, based on the tracker, consistency model such as BSP and SSP is able to control the global progress.

- Data Storage: The training data is stored on the distributed storage system called HDFS, each file is partitioned into blocks and may have several replicas on multiple machines for fault tolerance.

- Parameter Storage: Implement two common computer Maps and Vectors for machine learning data processing.

- HDFSManager: Provide useful APIs for developers to load data from HDFS with the classical producer-consumer paradigm.

- **Master** : The master node on the cluster for fault tolerance, when slave machines crashed, the master is able to detect the error with heartbeat mechanism, and restart the machine and recover the training progress from last checkpoint.

- MLTask: This model highly related to the machine learning application, which is used to describe the information about the application's task, it contains data about the table id, worker allocation and running callback.

- Launch Utils: The machine learning applications are running on multiple machines, the framework will allow users launch processes on multiple machines according to the node configuration by python scripts. The script is also able to control the debug or provision environment for test convenience.

4.2 Programming Model

The framework provides a KV-store APIs similar to other parameter server systems. Get and Add methods for application to read and update the model parameters stored in servers. Techniques details such as communication patterns between different machines, consistency control and fault tolerance are hidden from developers. I use a Logistic Regression application to illustrate how to use this framework to develop a distributed machine learning application.

Typically, most application have a similar steps as shown in the code snippet below. In Step 1, we need to load the node configuration from the file and parse it for IdMapper and PartitionManager initialization. Also, HDFSManager and DataLoader need be called to load training data from HDFS. In Step 2, we pass the loaded configuration information and data into the Engine. The Engine is helpful to start important components such as WorkerThread, ServerThread and SenderThread in the machine, also will initialize Mailbox and heartbeat service, which will provides inter-communication and fault tolerance to application. In Step 3, create KVClientTables for parameter management, the data storage and consistency models will be created according to users' configuration. Step 4, Construct MLTasks and use gradient descent based optimization methods to process data, update model parameters after every iterations. In Step 5, run the MLTask with engine, the engine will automatically trigger the training callback with multi-tasking, the configuration will be stored in a model called Info as context for training usage. In Step 6, the final step after training, need to stop everything in each machines, exit the application safely.

```
// Step 1: Load data
std::vector<Node> nodes = ParseFile();
std::vector<SVMItem> data;
HDFSManager::Config config;
loader.load(config, nodes, data);

// Step 2: Start Engine
Engine engine(nodes);
engine.StartEverything();

// Step 3: Create KVClientTable
engine.CreateTable(model_type);

// Step 4: Construct MLTask
```

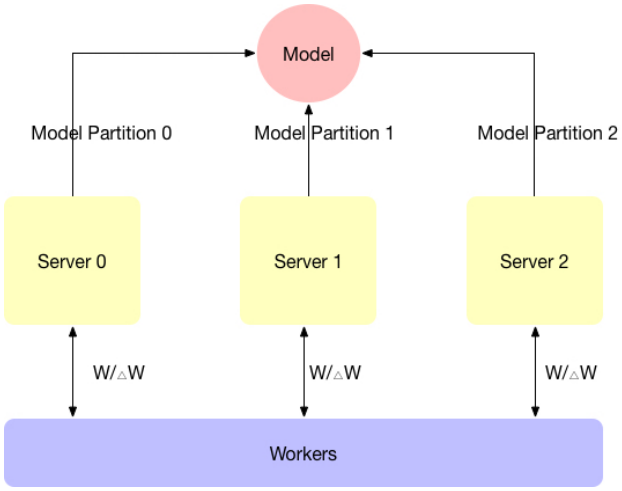

Figure 5. Model Parallelism; The perceptron model is segmented using model parallelism. In this method, workers are responsible for accepting the input and multiplying it by the associated weight Wi. After the multiplication, the result is sent to the each server and summed to obtain the model.

```
MLTask task;
task.SetLambda([](){
        // Get model parameters
        // Calculate gradient
        // Update parameters
});

// Step 5: Run MLTask
engine.Run(task);

// Step 6: Stop and exit
engine.StopEverything();
```

4.3 Computing Parallelism

Parallelism in simple definition, is to share workload with multiple machines when the performance of a single machine have been bottlenecked. There are two parallelism in the machine learning distributed systems: model parallelism and data parallelism.

4.3.1 Model Parallelism

In **model parallelism** , individual models are distributed across multiple machines. The performance improvement effect of parallel training of machine learning algorithm models on multiple machines depends mainly on the structure of the model. Models with a large number of parameters typically get more CPU cores and memory, so parallelizing large models can significantly improve performance and thus reduce training time.

As a simple example, suppose you have a perceptron, as shown in Figure. 5. To be more effectively parallelized, we can think of a neural network as a Dependency Graph with the goal of minimizing the number of synchronization mechanisms, assuming we have unlimited resources. In addition, the synchronization mechanism is required only when a node has multiple variable dependencies. Variable dependency is a dependency that can change over time. For example, bias is a static dependency because the value of the deviation remains the same over time. In the perceptron shown in Figure, parallelization is very simple. Calculations involving multiple inputs W (w0, w1, w2...) can be performed in parallel.

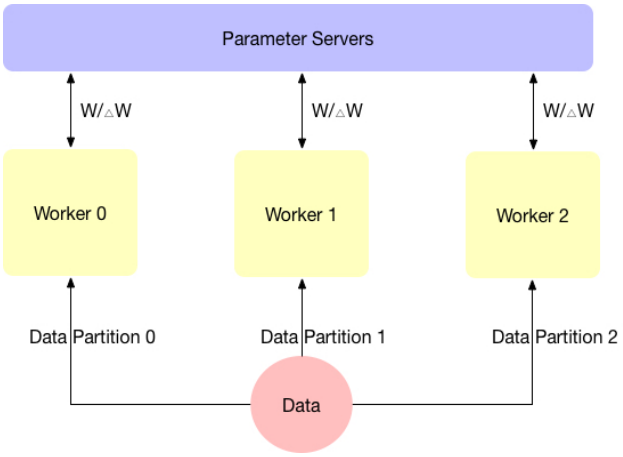

Figure 6. Data Parallelism; For example, n workers are used, and data partitions of the data set are assigned to each worker. Using this data block, each worker i will traverse a mini-batch data to generate a gradient $\triangle W$. Then, the gradient is sent to the parameter servers, and the parameter server PS collects the gradients and updates them according to a specific update mechanism.

4.3.2 Data Parallelism

Data parallelism is another completely different parameter optimization method. The idea for data parallelization to reduce training time is to optimize the parameters of a central model by using n workers to simultaneously compute n different data partitions. In this case, put n copies of the model on n processing nodes, that is, each node has a copy of the model. Each worker then trains a local model copy using the specified data block. However, all workers can work together at the same time to optimize the same goal.

The goal of the parameter server is to collect the parameter updates from each worker and to process the parameter requests from each worker. The distributed learning process first divides the data set into n shards. Each individual block will be assigned to a specific worker. Next, each worker samples the mini-batches data from its shards for training a local model. After one (or more) mini-batches, each worker will communicate with the parameter server to send a variable update request. This update request is generally a gradient $\triangle W$. Finally, the parameter server collects the gradients of all these parameter updates, averages them, and sends the updated values to workers. Figure. 6 shows the above process.

4.3.3 Implementation

In the framework, both data parallelism and model parallelism were implemented. Each machine nodes will load

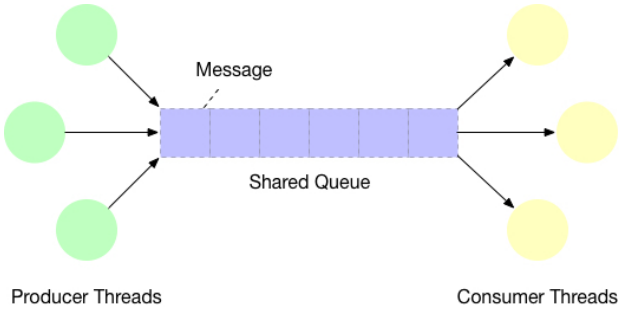

Figure 7. Producer-Consumer Pattern; The queue was shared by multiple threads, the producer use the queue to communicate with consumer, the consumer threads keep reading top message in the queue.

part of the training data with HDFSManager and full node configuration from local file system. For data loading, the framework define a global **HDFSBlockAssigner** to assigns blocks to the threads according to data locality. In the starting phase, the cluster will select the node with id 0 as the master of reading data from HDFS, the node will directly access the storage system and provide socket services to other nodes. Compared with other PS systems like ps-lite, the data is not read directly accessed from the remote storage system, but send requests to the HDFSBlockAssigner in the node with id 0. The Assigner plays a master role, responsible for connect and read from storage system, reply the data by via TCP/IP protocol among different nodes.

For Data Parallelism, I implement a **RangePartitionManager** to slice the data partitions among different nodes based on the node size in the cluster. For example, a training data set contains 15 partitions, there are 5 nodes waiting for data allocation and compute the task with parallelism, then node 0 will be allocated with data partition from 0 to 4, and node 1 will be allocated with data partition from 5 to 9, and so on. The developer have to implement optimization methods to training different data block, for each iteration, each nodes will send gradients to the parameter servers.

For Model Parallelism, I implement a **AbstractStorage** for parameters store and calculation in memory based on certain range, the range was initialized by the Engine on the starting phase according the node configuration. For example, the VectorStorage which stores model parameters represented with vector type, every time the Get or Add operations sent from different nodes, the VectorStorage will first convert the parameter index by the range, and then update gradients based on the converted index.

4.4 Request Handling

Request Handling is important as it need to ensure the data consistency problem among different machines, also it is important to avoid synchronization problem in different threads. A common property for parameter server like systems is that all of those system have Push and Pull operation for parameters. Push method is a synchronize operation which will return immediately after function call, while Pull method is a asynchronous operation which will return until the server node send response back to the worker.

I implements the framework with different type of requests, as follows: (1) Add: Push gradients from worker to server after calculation. (2) Get: Pull the model parameters from server to worker when iteration starts. (3) Clock: Called from worker to server indicates the iteration has end. (4) CheckPoint: Called from worker to server at fixed interval, global dump parameters and configuration for fault tolerance. (5) HeatBeat: Called from slave nodes to master node for heat-beat detecting.

Where Add, Clock and HeatBeat functions are asynchronous methods, which are easy to implements. However, Get and CheckPoint are synchronize methods, which are challenging to implement since the execute sequence between different nodes are hard to control. A novel way have been used for implementation, the core steps and sample code as follows:

```
void Get(keys, vals) { // or Clock
    Slice(keys);
    RegisterRecvFinishHandle(vals);
    NewRequest();
    Send();
    WaitRequest();
}
```

1. Register the reply callback for the Get or CheckPoint method, only triggered after the server have sent message back.
2. Initialize the request with the counter of expected response (according to the number of machine nodes) and the counter of current response (the value is 0 in this step).
3. Send the request message from worker to server.
4. Wait the response from servers. This is implemented with C++ condition library, only when the two counter in the second step is equal, the worker thread will continue to work. To note that one server's reply can only increase the counter of current response by 1, which means the worker thread will not awake until all the server send response back.

The **Producer-Conusmer** Pattern (as shown in Figure. 7) solves the problem of strong coupling between producers and consumers through a container. The producer and the consumer do not communicate directly with each other, but communicate through the blocking queue, so the producer does not have to wait for the consumer to process after the data is produced, and directly throws it into the blocking queue. The consumer does not ask the producer for the data, but Taken directly from the blocking queue, the blocking queue is equivalent to a buffer, balancing the processing power of producers and consumers. This blocking queue is used to decouple producers and consumers. Looking at most design patterns, a third party will be found for decoupling. For example, the third party in the factory pattern is the factory class, and the third party in the template pattern is the template class.

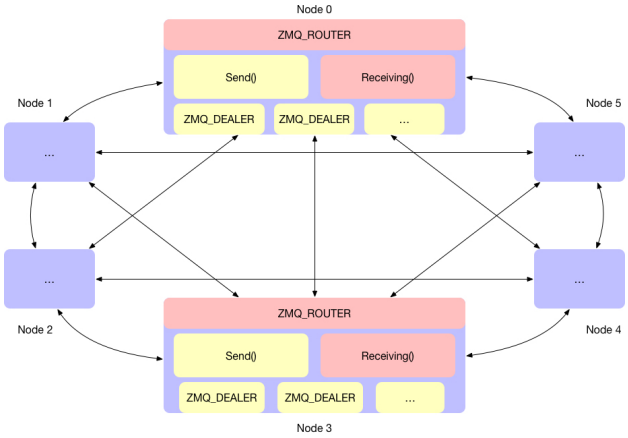

Figure 8. The Communication Design with ZeroMQ in the framework

In the framework, the element in the shared queue between producer and consumer is Message, which is the data communication across different nodes via TCP/IP. There are multiple types for Message, for example, kExit for stopping processing the data; kAdd for push data from worker to server; kGet for pull data from server; kCheckpoint for dump the server side parameters for fault tolerance etc. Those types are indicated with a enum type. The Message also contains other meta data such as the id of sender and receiver node, the consistency model id and the data vector for gradients update. In order to ensure the producer and consumer which located in different nodes to communicate, only the queue's id will be specified, the consumer will insert the Message in the producer's queue and wait for processing.

It is necessary to mention that the worker and server located on different machines, and the communication between them is based on the network. A messaging queue called **ZeroMQ** [19], it is a high-performance distributed message queue written in C++. It is a very simple and easy to use transport layer, making Socket programming easier, simpler and more efficient.

The inter-node, inter-process communication uses the dealer and router sockets in ZeroMQ (as shown in Figure. 8), both of which are upgrades to the request/reply mode. Messages sent using the dealer socket are looped at the peer end and queued equally. When router receives the message, it will remove the first frame in the message before forwarding it to the application process. Using ZeroMQ's request/reply mode, you can easily build a cluster that can communicate with each other. Each node in the cluster listens through router to provide services, and maintains a dealer queue to send messages.

4.5 Consistency Control

In a distributed computing system, because the calculation progress of multiple computing nodes is not completely consistent, it will lead to waiting for those nodes with slow calculation speed when summarizing the results, that is, slow nodes will slow down the progress of the entire computing task, waste computing Resources.

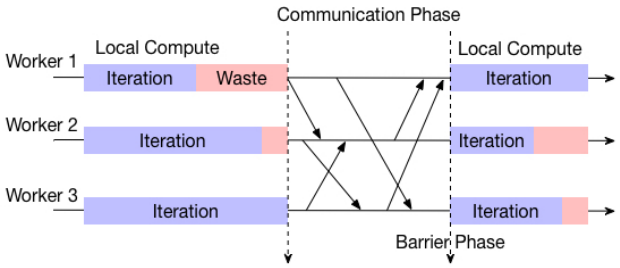

Figure 9. The BSP Model; Three phases in the BSP model, faster worker have to wait global barrier until enter next iteration (super-step), so there are waste after compute in these faster workers.

Considering the particularity of machine learning, the system can actually relax the synchronization limit. It is not necessary to wait for all the computing nodes to complete the calculation every round. For some workers who run fast, in fact, it is possible to push the trained increments first. Then proceed to the next round of training runs. This reduces latency and makes the entire computing task faster. Therefore, asynchronous control is one of the most important functions in distributed machine learning systems.

The framework provides three levels of asynchronous control protocols: BSP [20] (Bulk Synchronous Parallel), SSP [21] (Staleness Synchronous Parallel) and ASP (Asynchronous Parallel), whose synchronization limits are relaxed in turn. In order to pursue faster calculation speeds, the algorithm can choose a more relaxed synchronization protocol.

4.5.1 Bulk Synchronous Parallel

The goal of the **Bulk Synchronous Parallel** [20] model is to provide a theoretical model basis independent of the specific architecture for existing and future parallel architectures, so it is also called the Bridging Model with three important parts:

- Computing Components (at least consisting of a processor and a memory).
- Router that provides a communicable network for each computing component to enable peer-to-peer messaging between computing components.
- Barrier Synchronisation with interval T.

In the BSP model, the entire calculation process consists of a series of calculations separated by a barrier synchronizer (as shown in Figure.9), including the following phases:

1. Local Computing Phase: The computing node calculates the local data, stores the calculated result in the local memory, and stores the message data that needs to be sent to other computing nodes to the local message queue, waiting to be sent.
2. Global Communication Phase: communication between nodes in a point-to-point manner

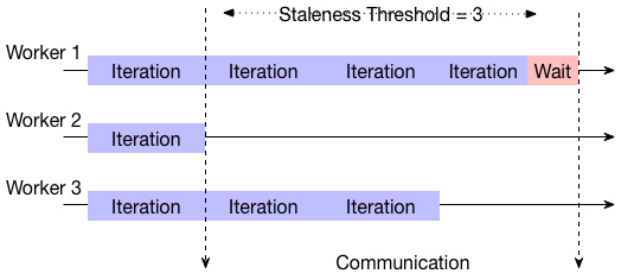

Figure 10. The SSP Model; The staleness threshold is 3, for every iteration application can call Clock function to continue, not need to wait for other nodes unless the threshold have been exceed.

3. Barrier Synchronization Phase: super step takes the barrier synchronization as the end point. The data communication of this super step takes effect after the barrier synchronization ends, and is used for the next super step. After ensuring that the data exchanged during the communication is transmitted to the destination computing node, and each computing node completes the calculations and communications performed in the current super-step, it can enter the next super-step, otherwise it stops waiting for other nodes to complete the calculation and communication.

In summary, the BSP model can be understood as parallel in the different nodes and serial in different iterations, it has a wide range of applications, and each round of iterations has high convergence quality. However, every iteration needs to wait for the slowest task, and the overall task takes a long time to calculate (straggler problems [22]).

4.5.2 Staleness Synchronous Parallel

The **Staleness Synchronous Parallel** [21] is a bounded asynchronous bridge model that not only can solve the straggler problem in the BSP model while preserving synchronization. Assume that the SSP model uses a master-slave architecture, the principle is as follows: suppose there are P slave nodes, and the machine is optimized in an iterative manner to solve learning problems $\triangle$, F , each node maintains an iteration counter t and model as a local view of parameter A. After completing a round of iterative calculations, each compute node submits the parameter update $\triangle$, and then do the following:

1. Call the Clock() function to indicate that the iteration calculation is completed.
2. Increase iteration counter t by 1.
3. Notify the master node to propagate the parameter update $\triangle$, of the node to other calculation sections, so that other compute nodes update their local view of model parameter A.

The Clock() function here is similar to the barrier synchronization in the BSP model, except that in the SSP model, since each node is asynchronously calculated, the update

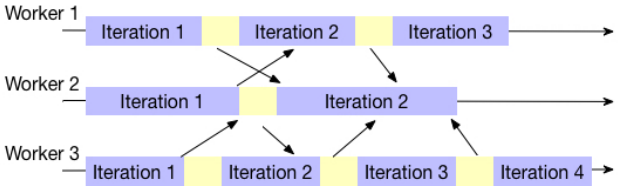

Figure 11. The ASP Model; Workers will not wait for each other, the iteration will continue directly until the end of the job in local node.

$\triangle$, F submitted by one compute node is not immediately passed to other nodes. As the other nodes may only receive partial parameter updates for the next iteration, the local view of the model parameters saved by the nodes at the p-th node in the t-th round becomes stale. For a given stale threshold s (s represents the iteration rounds difference between nodes), parallel systems based on the SSP model must satisfy the bounded stale conditions (as shown in Figure. 10), which is the difference between the number of the fastest and slowest compute nodes must be no more than s, otherwise the fastest compute node will be forced to wait for the slowest compute node, and also the model state guarantees that when a computing node calculates the update $\triangle$ in the t-th round, it needs to ensure that the node receives all the model parameter updates in the [0, t - s -1] round.

4.5.3 Asynchronous Synchronous Parallel

The Asynchronous Synchronous Parallel (ASP) model consists of a distributed processor with local memory and a total node that manages global parameters. The computational model is shown in Figure. 11.

The ASP process is as follows: 1) Each node uses local data to calculate all the model parameters, after the node calculates a round, the model can be updated at the master node. 2) The latest global parameters are obtained from the master node for the next round of updates and each node does not need to wait for each other.

4.5.4 Implementation

In order to implement BSP, ASP and SSP as consistency control in the framework, PendingBuffer and ProgressTracker have been defined to store the Message into the buffer, and control the iteration progress in the each nodes. The details for each consistency models as follows:

- BSPModel: ProcessTracker is necessary for this model, in a certain iteration n, only if all the nodes in cluster finish this iteration and updated local parameters to each other, this model can continue to the next iteration based on the record of the ProcessTracker.
- ASPModel: No need for ProcessTracker in this model, whenever Get or Add methods are called, these operations will be directly executed without any barrier or wait.
- SSPModel: This model need initialized with a variable called staleness as the threshold, actually, is the

staleness is 1, then SSPModel can act like the BSP-Model, when the staleness is ∞, the SSPModel will act like the ASPModel. The ProcessTracker and PendingBuffer will both used in this model, PendingBuffer is used to store reply message until all the nodes have finished in a certain iteration, while ProgressTracker is used to stored global progress. It is necessary to mention that a minimum progress is defined for the framework to know the slowest node.

4.6 Fault Tolerance

One feature of a distributed system that differs from a standalone system is that it can tolerate partial failure (**Fault Tolerance**). Partial failures can occur when a component in a distributed system fails. This failure may affect the correct operation of other components, but it may also not affect other components at all. Failures in non-distributed systems often affect all components and can easily crash the entire application. An important goal in the design of distributed systems is to construct the system in such a way that it can automatically recover from partial failures without seriously affecting overall performance. In particular, when a failure occurs, the distributed system should continue to operate in an acceptable manner while recovering, that is, it should be able to tolerate errors and continue to operate to some extent in the event of an error.

4.6.1 Types of Failures

Failures in fault-tolerant distributed systems can be divided into process failures and communication failures. When it comes to assuming or allowing a fault under a system model, the implication is that it is possible to assume that such a fault is occurring under this system model.

For the **process failure**, the process running normally means that the process operates according to the specified protocol. It is generally assumed that there is at least one running process in the system. If a process is not working properly, it will be defined as failed. Process failures can be classified according to the degree of harm: stop failure, missed failure, and Byzantine failure. Stopping a fault means that a process goes from a normal state to a completely stopped state and is no longer recovered. Missing faults mean that a process misses or misses a message when it should send or receive a message. Byzantine failure means that the process does not operate according to the specified protocol, ie its behavior is completely unpredictable. According to the number of processes that have failed, they can be divided into most faults and a few faults. Most faults refer to the number of processes that have failed in a certain operation of the system exceeds or equal to half; a few faults mean that the number of processes that have failed in a certain operation of the system is less than half.

For the **communication failure** , define the link from process A to B is reliable, it means that the link meets the following two points: First, when both A and B are running normally, A sends The message of B will eventually arrive at B; Second, if B receives a message from A, then the message is indeed sent by A. If a link is not reliable, then we say it has failed. Communication failures can be classified as message loss and any link according to the degree of harm. Message loss refers to a link that violates the first point above, but meets the second point. Any link refers to a link that violates the first or second point above.

4.6.2 Fault Tolerance Mechanism

The most basic mechanism of fault tolerance in distributed system is **checkpoint and rollback** recovery. The application should trigger checkpoint at fixed time interval or at the end of certain stages, the checkpoint will save the state of the systems, it mainly includes two parts, the data generated by different nodes, for example in the parameter system, is the parameters and training; the communication and computing progress of the application. This recovery mechanism includes two important stages, the checkpoint stage which will dump the states into the persist storage such as distributed system or local disk (system like HDFS will be better as those system ensure the fault tolerance to the data block); the rollback stage, which will read the dumped data from the persist storage and notify every nodes in the cluster, recover the progress and the data.

Another common mechanism for fault tolerance is to store multiple copies of data redundantly (**Data replication**), each called a replica. In this way, when a node fails, data can be read from other copies. It can be argued that replicas are the only means of fault-tolerant technology for distributed storage systems. Due to the existence of multiple copies, how to ensure the consistency between the copies is the theoretical core of the entire distributed system.

In the distributed system, checkpoint and data replication have been widely used. For example, HDFS adopt the data replication to guarantee the fault tolerance and load balancing, which can provide high availability to users; The Spark adopt the checkpoint and rollback strategy, backup the Lineage log, and try to generate the origin data with it after detecting the error. It is commonly believe that data replication can achieve better performance in the cluster with massive number of nodes. However, in the parameter server based framework, I adopt the checkpoint and rollback mechanism, the reason for that is data replication on small set of nodes can cause high overhead on the application.

4.6.3 Implementation

I implement a fault tolerance mechanism with checkpoint in the framework (the design shown in Figure. 12). After the application call CheckPoint method with the KVClientTable, the workers and servers in the cluster will start to dump the data and parameters for storage, the Engine will also dump the configuration, this can ensure when the failure happens, the Engine is able to rollback the progress and continue the processing. I detecting the failure in the cluster using a heat-beat based method, to doing that, a master node will be launched to store the information of heatbeat about all the slave nodes in the cluster. For a certain time interval, the slave need to send the heat-beat message, which contains the timestamp and node id data, the master then will update the record. The master will detecting

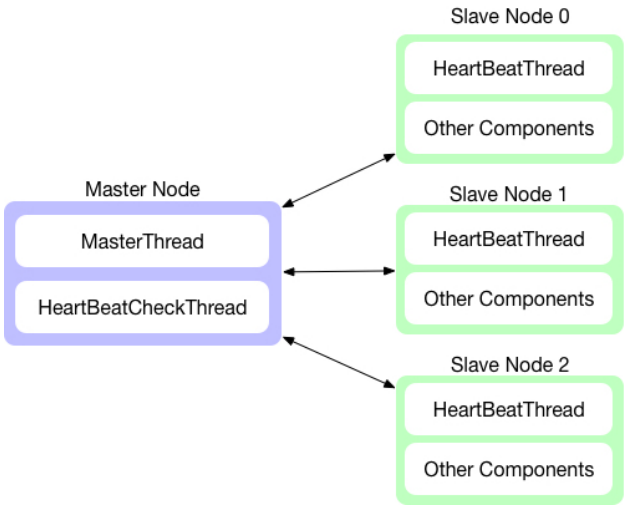

Figure 12. The fault tolerance design in the framework; A typical master-slave pattern, the master is keep detecting on the heat-beat from slave nodes, and restart the failed node once it was detected, the slave nodes keep sending the heat-beat with a fixed time interval.

the failure based on the heat-beat record, if one of the node haven't update the record until the timeout threshold, the master will try to restart the failed node again, and stop the computing progress in other nodes. When the process on the failed node restart successfully and reload all the data from the checkpoint storage, the master is able to receive the heat-beat message from it, and then send messages to other nodes, recover the computing progress in the cluster.

The core components about this design as follows:

- Master Node: Maintain the a heat-beat map, provides service for slave nodes to update their record, keep detecting the timeout among all the nodes in the cluster, restart the failed node and coordinate other nodes when failure happens.
- Slave Nodes: Similar to the parameter server like systems, consist of worker and server type nodes. Sending the heart-beat to the master in a fixed time interval. In order to recover the progress, slave nodes have to call CheckPoint for backup.
- HeatBeatThread: The main component for fault tolerance in the slave nodes, responsible for sending heat-beat to master without interfere other threads.
- MasterThread: The server side of the HeatBeatThread, the message from the slave nodes will directly routed to the MasterThread, the data will be extracted and used to update the overall information.
- HeatBeatCheckThread: The main component in the master node, used to check the slave nodes' heat-beat records, restart the failed node with Python script.

5. APPLICATIONS ON FRAMEWORK

In this section, I present a discussion about three major types of machine learning algorithms: classification, clustering and recommendation. I introduce the abstraction of these algorithms and how they are implemented in the framework.

5.1 Linear/Logistic Regression

Linear Regression [23] is a regression analysis that models the relationship between one or more independent variables and dependent variables using a least squares function called a linear regression equation. This function is a linear combination of one or more model parameters called regression coefficients (the independent variables are all squares). The case of only one independent variable is called simple regression, and the case of more than one independent variable is called multiple regression. The model is denoted as: $h_\theta = \theta^T x$, the loss function θ is denoted as:

$$J(\theta) = \frac{1}{2m}\sum_{i=1}^{m}(h_\theta(x_i) - y_i)^2$$

Where m is the number of training data, the model can also denoted as: $h_\theta = \theta^T x + \epsilon$, where x is the weights in vector form, ϵ is the bias term. In order to minimize the loss, the gradient based optimization method can be used in the framework.

Logistic Regression [24] can be seen as a special case of linear regression. Unlike linear regression, logistic regression processing outputs are binary classification problems. It can be denoted as:

$$h_\theta(x) = \frac{1}{1 + e^{-\theta^T x + \epsilon}}$$

Where y is probability of the output variable that is equal to 1. The loss function can be denoted as:

$$J(\theta) = -\frac{1}{n}\sum_{i=1}^{n} y_i log h_\theta(x_i) + (1 - y_i) log(1 - h_\theta(x_i))$$

This loss function can also be minimized by the gradient-based methods. In the framework, a KVClientTable will be created for storing the weights vector for the logistic regression algorithm, the developer should define the optimization method, and in each iteration, the gradients need be computed and push to the server.

5.2 K-Means

The goal of the **K-means** algorithm is to find a group in the data, the number of clusters being represented by the variable K. Assign each data point to one of the K clusters by iterative operation based on the characteristics provided by the data. The core operations as follows:

1. Initialize the number of clusters K.
2. Then in the feature space, randomly pick k points as cluster centers.
3. Calculate distances from each points to cluster centers.

4. Assign each point to the nearest cluster center.

5. Each cluster will have been classified, and use this to update the new cluster centers.

6. Repeat 3–5 until there is too much change (convergence) in all clusters.

Although K-Means is not a gradient-based algorithm, but the parameter server like framework can also process it with a iterative way. In the framework, in order to implement this algorithm, two MLTask have to be defined, one is for initialize the K cluster centers, another is used to update center during iterations.

5.3 Matrix Factorization

Matrix Factorization [25] have been widely used in the recommendation tasks. In a standard recommendation task, we have m users, n items, and a sparse scoring matrix R ($R \epsilon R^{m*n}$). Each R_{ij} in the R represents the rating of user i for item j. If $R_{ij} \neq 0$, then user i has a rating for item j, and vice versa. Each user i can be represented by the vector $S_i^u = (R_{i1}, R_{i2}...R_{in})$. Similarly, each item j can use the vector $S_i^j = (R_{1j}, R_{2j}...R_{mj})$ to denote. For each side information matrix of the user and the item, it is represented by $X \epsilon R^{m*p}$ and $Y \epsilon R^{n*q}$, respectively.

Let $u_i, v_j \epsilon R^k$, where u_i is the latent factor vector of user i, v_j is the latent factor vector of item j, and k is the dimension of hidden space. Thus, for the user and the item, the corresponding hidden factor vector forms are $U = u_{1:m}$ and $V = v_{1:n}$, respectively. Since $R = UV$, if U and V can be found, then we can find a non-sparse scoring matrix R. Given a sparse scoring matrix R, and edge information matrices X and Y, our goal is to learn U and V to predict missing scores in R. (This is also called UV decomposition of the matrix). For the user-item scoring matrix R, we can decompose it into a user-characteristic matrix, and a feature-item matrix. There are two advantages to doing so:

- Get the user's preferences and the characteristics of the item
- Reduce the dimensions of the matrix

The goal of the optimization task in Matrix Factorization is to minimize the loss function below:

$$\sum_{i=1}^{m} \sum_{j=1}^{n} I_{ij} (R_{ij} - [UV]_{ij})^2$$

Where I_{ij} is the indicator of whether R_{ij} is observed. This function can be optimized using the gradient descent methods and implemented on the parameter server framework with the MapStorage to store the matrix parameters.

The algorithms discussed above, all involve objective functions that can be optimized using gradient-based algorithms. The optimization problem combined with the machine learning algorithm can be expressed as finding parameters to minimize the basic loss function and some regularization terms. Therefore, gradient calculations about parameters can be considered the core of learning problems. Since each sample contributes to parameter displacement independently in these machine learning models, the gradient calculation can be represented by the step of how each sample data and parameters participate in the calculation of new parameters. In addition, because the amount of data and model size are very large, each sample affects only a small portion of the update of a small number of model parameters. These attributes of the machine learning model discussed above are important for data parallel implementation and are useful for parallelizing gradient-based optimization algorithms such as SGD, Momentum and AdaGrad.

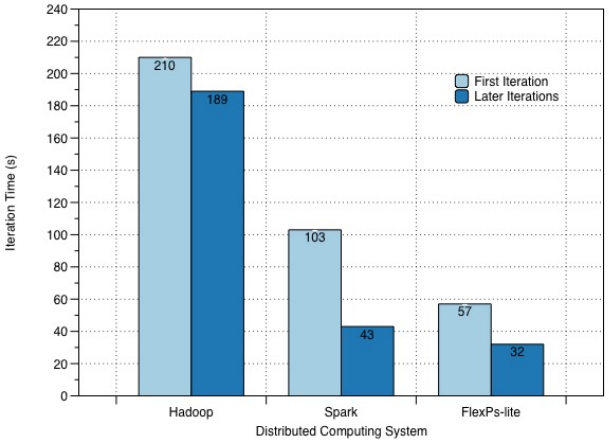

Figure 13. Performance comparison between popular distributed computings systems Hadoop and Spark.

6. EVALUATION

In this section, I evaluate the framework with the popular machine learning algorithm on a common cluster, which shows the performance improvement compared with other parameter server system and distributed computing system. I also evaluate the overhead impacts on the training performance with checkpoint for fault tolerance.

Cluster setup: I evaluated the distributed framework on a cluster with 15 machines connected via 1 Gbps Ethernet. Each machine is equipped with two 2.0GHz E5-2620 Intel(R) Xeon(R) CPU (12 physical cores in total), 48GB RAM, a 450GB SATA disk (6Gb/s, 10k rpm, 64MB cache), and running on 64-bit CentOS release 7.2.

ML Application setup : I use Logistic Regression as the evaluation application, since it is the most representative machine learning algorithm. I adopt a simple range data partitioner strategy, which will partition the training data evenly across the workers. All the test cases will use SSP with staleness threshold 1 (equals to BSP consistency model). The kdd database [1] was chosen as the training and test data for the application.

Performance measure: The evaluations measures the performance, which shows the convergence rate on the training dataset, I compare the framework (denoted as **FlexPs-lite**) with Bösen which is also a parameter server like system. In another evaluation, I compare the framework

[1] www.kddcup2012.org/c/kddcup2012-track2/data

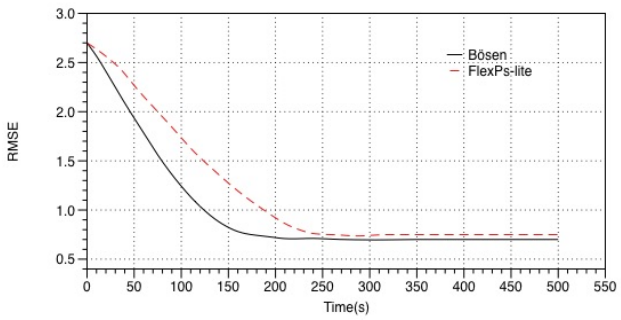

Figure 14. The convergence rate comparison with Bösen based on RMSE.

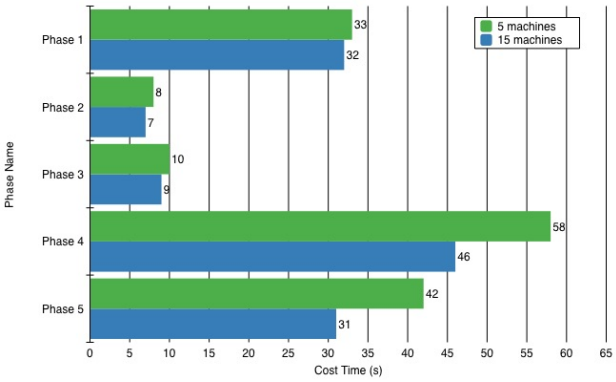

Figure 15. The fault recovery time on the different phases.

with other popular distributed computing systems such as Hadoop and Spark, which will shows the great advantages on the reduction of first and later iteration time.

Fault Recovery Performance : In order to measure the impacts on the performance, I compare the iteration cost with checkpoint function call on and off. To illustrate the time cost on the checkpoint, I divided the checkpoint and rollback operation into different phases, and show the time cost on those phases with different machines on the cluster (5 and 15 machine nodes respectively).

6.1 Performance Measure

I adopt Logistic Regression as the iterative machine learning application, compare the performance on the following distributed systems:

- Hadoop: The Hadoop 2.7.2 stable release.
- Spark: The Spark 2.2.0 stable release.
- FlexPs-lite: The implementation of the parameter server framework.

I ran the algorithm for 10 iterations on the KDD12 datasets using 15 machines. I implement the standard logistic regression application on all the framework, and compared the iteration time cost based on two factors(as shown in Figure. 13):

- First Iteration: In this iteration, the cluster machines need to read training data from distributed storage system HDFS, the computing environment need to be setup. FlexPs-lite is faster than Hadoop and Spark, the main reason is because FlexPs-lite is so light, no extra operation need to be done in the starting phase and iteration setup phase.
- Later Iterations: I compute the average iteration time cost and compare with each platform, the FlexPs-lite is also moderately faster than Hadoop and Spark, because FlexPs-lite have less communication cost, another reason is FlexPs-lite is implemented in C++, which can reduce lots of overhead.

In order to compare the convergence rate on the FlexPs-lite and Bösen, as the Bösen is the starte-of-art parameter server implementation, both systems were written in C++, I measured how efficient a system can compute a result that have more convergence rate based on the same given input.

I adopt the **RMSE** [26] as the measurement. RMSE is a secondary scoring rule that also measures the average magnitude of the error. It is the square root of the mean difference between the predicted and actual observations. This can be calculated as:

$$RMSE = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y_i})^2}$$

The result was obtained as shown in the Figure. 14. It can be seen that performance between Bösen and FlexPs-lite is similar, but FlexPs-lite have better performance, the main reason is FlexPs-lite have a lighter implementation.

6.2 Fault Recovery Performance

I evaluated fault tolerance by randomly kill single running application process running on the machine. When the application process in the machine was killed, the Master node will detect this error as the process no longer send the heat-beat report. Then the master node will call the python script to restart the slave node, after the slave node successfully restart and reload, the cluster will continue to work, the main phases were list as follows:

1. Application Process Failure: One of the application process was failed due to process dead or machine shutdown.
2. Master Node Detected: The master node have detected the exit of the slave process, and able to get the node id of the machine.
3. Restart Failure Node: The master node will call the python script with parameters to restart the machine with the failure node id. The parameters for script was initialized when the cluster start.
4. Node Restarted: The failure node have been restarted in the machine, ready to load the configuration and reload the data.

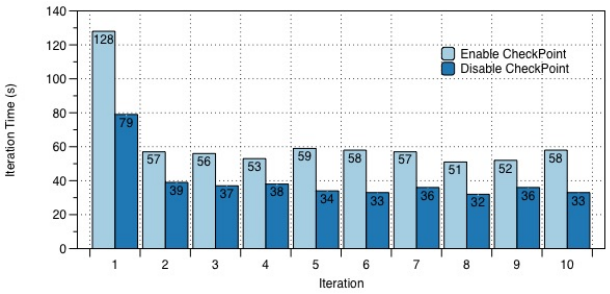

Figure 16. The iteration time cost comparison based on checkpoint enable and disable.

5. Rollback Completed: The restarted node will send heartbeat to the master node, then all the nodes in the cluster will complete the rollback based on the data dumped in the checkpoint stage.

6. Recover Success: All the configuration and data have been reloaded by all the machine nodes in the cluster, the application will continue to work from the last checkpoint.

Phases in the process failure were divided in to 5 stages, the evaluated report about the time cost of these stages are shown in the Figure. 15. It can been seen the time cost of stage 1-3 are similar, which is because these stages were happen on the master node and failure node, while the time cost on the stage 4 and 5 is less, as these stages were operated on the machines that is distributed on the cluster.

I divide the checkpoint and rollback for fault tolerance into 6 stages:

1. The application in the machine node id 0 call the CheckPoint function to other nodes in the cluster.

2. Every workers and servers in the cluster start to dump the parameters and data, the Engine in each will dump the configuration and the iteration progress into storage.

3. One of the machine in the cluster have failed.

4. The master node detect the failure and restart the failed node.

5. Every workers and servers in the cluster start to rollback and reload the data that was dumped in stage 2.

6. The job progress recovered to the checkpoint and continue to computing.

The stage 2 in the checkpoint and rollback operation is the very time consuming step, which can influence a lot to the iteration time, I evaluated the cost time for the iteration enable and disable checkpoint function (as shown in Figure.16). It is obvious that iteration with checkpoint function will cost more time.

7. FUTURE WORKS

I have implemented a distributed machine learning computing framework with baseline and fault tolerance, and implement different type of popular algorithms such as Logistic Regress, K-Means and Matrix Factorization. However, there are several possible direction I can do more on the future works:

- Implement more features on the framework. For example, addressing straggler problems in the consistency model, when a worker that is too slow and slow down the overall cluster, it is possible optimize this with other ways. In is also necessary to optimize the bandwidth management and communication pattern.

- Implement more machine learning algorithms on the framework, such as the Support Vector Machine (SVM) for classification and Latent Dirichlet Allocation for topic modeling.

- Integrate the framework into the popular container such as Kubernetes.

8. CONCLUSIONS

I implement a parameter server framework that provide the basic function such as model accessing, training data abstraction and application logic execution. I introduce the evolution of the distributed computing systems, and illustrate why the parameter server framework is necessary for the machine learning algorithms. I demonstrated the concept on different optimization methods, which can be used on the framework with code examples. The framework architecture and implementation have been well explained with the programming model, computing parallelism, consistency model and request handling. In order to enable the fault tolerance, the design of the checkpoint and rollback method have been illustrated with design and comparison. I evaluate the overall performance on the framework with other computing systems such as Hadoop, Spark and Bösen, and show the influence on the framework with checkpoint and rollback function enabled.

9. REFERENCES

[1] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[2] C. Chambers, A. Raniwala, F. Perry, S. Adams, R. R. Henry, R. Bradshaw, and N. Weizenbaum, "Flumejava: easy, efficient data-parallel pipelines," in *ACM Sigplan Notices*, vol. 45, no. 6. ACM, 2010, pp. 363–375.

[3] M. Li, D. G. Andersen, J. W. Park, A. J. Smola, A. Ahmed, V. Josifovski, J. Long, E. J. Shekita, and B.-Y. Su, "Scaling distributed machine learning with the parameter server." in *OSDI*, vol. 14, 2014, pp. 583–598.

[4] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly, "Dryad: distributed data-parallel programs from sequential building blocks," in *ACM SIGOPS operating systems review*, vol. 41, no. 3. ACM, 2007, pp. 59–72.

[5] Y. Bu, B. Howe, M. Balazinska, and M. D. Ernst, "Haloop: efficient iterative data processing on large clusters," *Proceedings of the VLDB Endowment*, vol. 3, no. 1-2, pp. 285–296, 2010.

[6] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*. USENIX Association, 2012, pp. 2–2.

[7] D. G. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi, "Naiad: a timely dataflow system," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. ACM, 2013, pp. 439–455.

[8] F. Yang, J. Li, and J. Cheng, "Husky: Towards a more efficient and expressive distributed computing framework," *Proceedings of the VLDB Endowment*, vol. 9, no. 5, pp. 420–431, 2016.

[9] J. Dean, G. Corrado, R. Monga, K. Chen, M. Devin, M. Mao, A. Senior, P. Tucker, K. Yang, Q. V. Le *et al.*, "Large scale distributed deep networks," in *Advances in neural information processing systems*, 2012, pp. 1223–1231.

[10] J. Wei, W. Dai, A. Qiao, Q. Ho, H. Cui, G. R. Ganger, P. B. Gibbons, G. A. Gibson, and E. P. Xing, "Managed communication and consistency for fast data-parallel iterative analytics," in *Proceedings of the Sixth ACM Symposium on Cloud Computing*. ACM, 2015, pp. 381–394.

[11] J. Jiang, L. Yu, J. Jiang, Y. Liu, and B. Cui, "Angel: a new large-scale machine learning system," *National Science Review*, vol. 5, no. 2, pp. 216–236, 2017.

[12] H. Robbins and S. Monro, "A stochastic approximation method," in *Herbert Robbins Selected Papers*. Springer, 1985, pp. 102–109.

[13] N. Qian, "On the momentum term in gradient descent learning algorithms," *Neural networks*, vol. 12, no. 1, pp. 145–151, 1999.

[14] Y. E. Nesterov, "A method for solving the convex programming problem with convergence rate o (1/kˆ 2)," in *Dokl. Akad. Nauk SSSR*, vol. 269, 1983, pp. 543–547.

[15] R. A. Jacobs, "Increased rates of convergence through learning rate adaptation," *Neural networks*, vol. 1, no. 4, pp. 295–307, 1988.

[16] M. D. Zeiler, "Adadelta: an adaptive learning rate method," *arXiv preprint arXiv:1212.5701*, 2012.

[17] T. Tieleman and G. Hinton, "Rmsprop: Divide the gradient by a running average of its recent magnitude. coursera: Neural networks for machine learning," *COURSERA Neural Networks Mach. Learn*, 2012.

[18] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.

[19] P. Hintjens, *ZeroMQ: messaging for many applications*. " O'Reilly Media, Inc.", 2013.

[20] L. G. Valiant, "A bridging model for parallel computation," *Communications of the ACM*, vol. 33, no. 8, pp. 103–111, 1990.

[21] Q. Ho, J. Cipar, H. Cui, S. Lee, J. K. Kim, P. B. Gibbons, G. A. Gibson, G. Ganger, and E. P. Xing, "More effective distributed ml via a stale synchronous parallel parameter server," in *Advances in neural information processing systems*, 2013, pp. 1223–1231.

[22] A. Harlap, H. Cui, W. Dai, J. Wei, G. R. Ganger, P. B. Gibbons, G. A. Gibson, and E. P. Xing, "Addressing the straggler problem for iterative convergent parallel ml," in *Proceedings of the Seventh ACM Symposium on Cloud Computing*. ACM, 2016, pp. 98–111.

[23] G. A. Seber and A. J. Lee, *Linear regression analysis*. John Wiley & Sons, 2012, vol. 329.

[24] T. A. Cameron, "A new paradigm for valuing nonmarket goods using referendum data: maximum likelihood estimation by censored logistic regression," *Journal of environmental economics and management*, vol. 15, no. 3, pp. 355–379, 1988.

[25] Y. Koren, R. Bell, and C. Volinsky, "Matrix factorization techniques for recommender systems," *Computer*, no. 8, pp. 30–37, 2009.

[26] C. J. Willmott and K. Matsuura, "Advantages of the mean absolute error (mae) over the root mean square error (rmse) in assessing average model performance," *Climate research*, vol. 30, no. 1, pp. 79–82, 2005.