A Comparison of Distributed Machine Learning Platforms

Kuo Zhang
University at Buffalo, SUNY

Salem Alqahtani
University at Buffalo, SUNY

Murat Demirbas
University at Buffalo, SUNY

ABSTRACT

The proliferation of big data and big computing boosted the adoption of machine learning across many application domains. Several distributed machine learning platforms emerged recently. We investigate the architectural design of these distributed machine learning platforms, as the design decisions inevitably affect the performance, scalability, and availability of those platforms. We study Spark as a representative dataflow system, PMLS as a parameter-server system, and TensorFlow and MXNet as examples of more advanced dataflow systems. We take a distributed systems perspective, and analyze the communication and control bottlenecks for these approaches. We also consider fault-tolerance and ease-of-development in these platforms. In order to provide a quantitative evaluation, we evaluate the performance of these three systems with basic machine learning tasks: logistic regression, and an image classification example on the MNIST dataset.

1. INTRODUCTION

The goal of machine learning is to "learn" from input data and construct a suitable model by continously estimating, optimizing, and tuning parameters of the model. A recent insight in machine learning is that it is possible to replace complexity in modeling with the use of very large scale datasets for training [11]. With the proliferation of big data, and with the advances in big data processing frameworks, this insight led to very large scale machine learning deployments, and boosted the adoption of machine learning across many application domains. Today, search engines employ machine learning for classification, clustering, and indexing of documents. Recommendation systems and healthcare services employ machine learning to improve their services. In particular, deep learning, an important branch of machine learning, has achieved transformative success in pattern recognition applications, including speech recognition and image recognition.

Large scale machine learning platforms are inevitably built as distributed data processing systems. *Dataflow systems* take a functional programming view of data processing as state transformations [8, 13, 16] and has been adopted widely by distributed data processing systems. Examples of dataflow systems include MapReduce [10], Naiad [15], Spark [21, 20]. When using these dataflow systems, the developer models/represents the computation using a directed graph abstraction composed from the system-provided primitives. The vertex and edge in the directed graph are associated with special meanings which vary from system to system. (For example, in MapReduce a vertex corresponds to a map or reduce task, and an edge corresponds to data communication/shuffle. In Spark, a vertex corresponds to a Resilient Distributed Dataset (RDD) [20], and an edge corresponds to RDD operations.) The dataflow platform then translates this directed graph to a physical execution plan, consisting of scheduled tasks, executes these tasks across a cluster of machines.

Due to the success of dataflow paradigm in big data processing systems, initially dataflow systems were adopted for simple distributed machine learning tasks. For example, Spark is designed as a general data processing framework, and with the addition of MLlib [1], machine learning libraries, Spark is retrofitted for addressing some machine learning problems.

For complex machine learning tasks, and especially for training deep neural networks, the dataflow model fails to scale as mutable state and iteration becomes crucial. *Parameter-server architecture* was proposed to enable in-place updates to very large parameters. In this model, the workers iterate over computation in rounds, and are responsible for computing an update to the model parameters in each round. The parameter server maintains the model parameters using a distributed store structure such as distributed table. Usually there is no communication across workers, and workers only communicate with the parameter server. While data parallel is the most dominant form of parallelism in this approach, model-parallelism is also achievable. Examples of parameter-server architecture includes Google DistBelief [9], Parameter Server [14], and PMLS [19].

Finally more advanced dataflow systems have been developed recently to address distributed machine learning and deep learning problems, including Google TensorFlow [6] and MXNet [7]. These dataflow systems allow cyclic graphs with mutable states and can mimic the functionality of a parameter server. Writing the computation as a dataflow symbolic computation graph enables these platforms to perform graph rewriting, partitioning, and placement to optimize performance over distributed nodes. These platforms also aim to provide flexibility of customizing the parameter-server implementation (with different optimization algorithms,

consistency schemes, and parallelization strategies) at the application layer.

Contributions of the paper. We investigate the architectural design of these distributed machine learning platforms, as the design decisions inevitably affect the performance, scalability, and availability of those platforms. More specifically, we compare and contrast the strengths and drawbacks of the dataflow and parameter-server approaches for building distributed machine learning frameworks. We take a distributed systems perspective, and analyze the communication and control bottlenecks for these approaches. We also consider fault-tolerance and ease-of-development of these approaches.

In order to provide a quantitative evaluation, we perform experiments with representative systems: Spark for dataflow, PMLS for parameter-server, and TensorFlow and MXNet as examples of more advanced hybrid dataflow systems. We evaluate the performance of these systems with the same basic machine learning tasks: logistic regression, image classification using feed-forward neural network on the MNIST dataset [2]. We also employ Ganglia system monitoring tool [3] to inspect the network, CPU, and memory utilization during training in order to reveal potential system bottlenecks.

Our experiments show that while Spark performs good for simple logistic regression, its performance dips for more involved machine learning tasks. Spark does not have a parameter-server abstraction and this limits Spark's scalability, especially when facing a machine learning task containing a large volume of model parameters. Keeping the model as an RDD drags the performance of Spark significantly, so in our experiments we maintained and updated the model parameters at the driver. Even with this configuration, Spark's performance falls below the other platforms for image classification task with single and two hidden layers. We also found that the computation speed varies significantly with different numbers of RDD partitions.

Our experiments show that the parameter-server model provides fast iteration and very good performance for training machine learning and deep learning tasks. Since PMLS implements the parameter-server at a low-level using a high performance programming language C++, it achieves the best performance in terms of speed in our experiments. While PMLS has very little overhead, on the negative side, this means that the users of PMLS need to know how to handle computation using relatively low-level APIs.

The advanced dataflow systems developed for machine learning, TensorFlow and MXNet, failed to perform well in terms of speed. This is due to the overhead caused by the high levels abstractions used in these platforms. On the other hand, these abstractions enable these systems to work on multiple platforms and leverage not only CPU but also GPU and other computational devices. While these systems have been shown to scale to hundreds of machines, our experiments were performed with up to 6 workers, so they do not evaluate these platforms at large-scale. In our experiments, we found that asynchronous training of the workers with respect to the parameter-server achieved higher speeds than synchronous training.

On the usability front, the advanced dataflow systems provide several benefits. By adopting symbolic execution graphs, they abstract away from the distributed execution at the nodes level and also enable optimizations by graph rewriting/partitioning when staging the computation on the underlying distributed nodes. They provide, to some extent, flexibility of customizing the parameter-server implementation (with different optimization algorithms, consistency schemes, and parallelization strategies) at the application layer. While support for data-parallel training with parameter-server abstraction is provided, it is still very cumbersome to program model-parallel training using these platforms.

Outline of the rest of the paper. We provide a brief architectural overviews of Spark, PMLS, TensorFlow, and MXNet in Sections 2, 3, 4, and 5 respectively. In Section 6, we evaluate the performance of these platforms, and in Section 7 we present our concluding remarks and identify directions for future work.

2. SPARK DATAFLOW SYSTEM

In order to achieve better performance than its forerunner MapReduce, Spark enables in-memory caching of frequently used data and avoids the overhead of writing a lot of intermediate data to disk. For this Spark leverages on Resilient Distributed Datasets (RDD), read-only, partitioned collection of records distributed across a set of machines.

In Spark, a computation is modeled as a directed acyclic graph (DAG), where each vertex denotes an RDD and each edge denotes an operation on RDD. On a DAG, an edge E from vertex A to vertex B implies that RDD B is a result of performing operation E on RDD A. There are two kinds of operations: *transformations* and *actions*. A transformation (e.g., *map, filter, join*) performs an operation on a RDD and produces a new RDD. An action (e.g., *collect, count*) triggers a *job* in Spark. A typical Spark job performs a couple of transformations on a sequence of RDDs and then applies an action to the latest RDD in the lineage of the whole computation. A Spark application runs multiple jobs in sequence or in parallel.

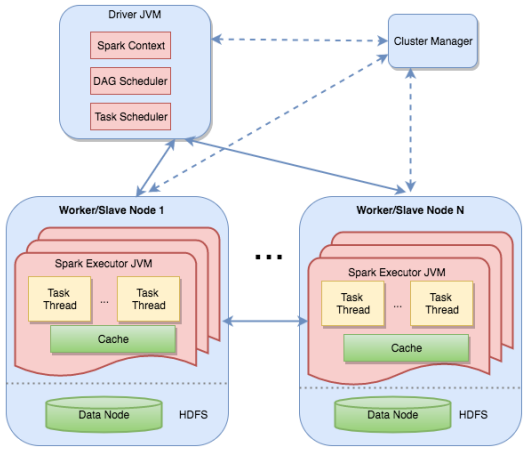

Figure 1: Spark Architecture

Figure 1 shows the architecture of a Spark cluster, which comprises of a *master* and multiple *worker*. A master is responsible for negotiating resource requests made by the Spark *driver* program corresponding to the submitted Spark application. Worker processes hold Spark executors (each of

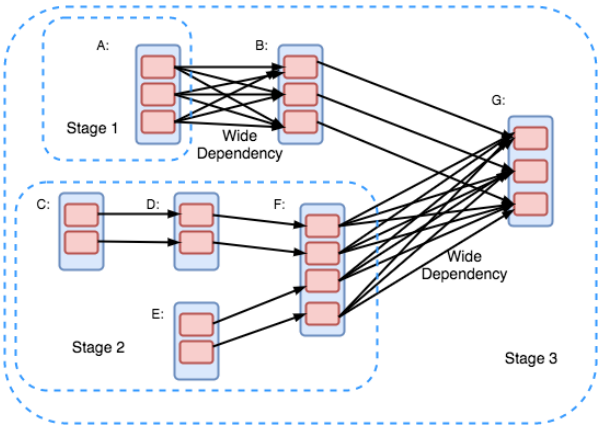

Figure 2: RDD Stages

which is a JVM instance) that are responsible for executing Spark tasks. The Spark driver contains two scheduler components: the *DAG scheduler* and the *task scheduler*. The DAG scheduler is responsible for stage-oriented scheduling, and the task scheduler is responsible for submitting tasks produced by the DAG scheduler to the Spark executors.

Unlike the MapReduce framework that consists of only two computational stages, map and reduce, a Spark job may consist of a DAG of multiple stages. The stages are run in topological order. A stage contains a set of independent tasks which perform computation on partitions of RDDs. These tasks can be executed either in parallel or as pipelined. Spark defines two types of dependency relation that can capture data dependency among a set of RDDs: *narrow dependency* and *shuffle dependency (also called wide dependency)*. Narrow dependency means each partition of the parent RDD is used by at most one partition of the child RDD. Examples include *map*, *filter*, and *union* transformations. Wide dependency means multiple child partitions of RDD may depend on a single parent RDD partition. Examples include *groupby* and *join* transformations. It is the wide/shuffle dependency that defines the boundary of two connected stages. Data exchange across executors only happens between two adjacent stages and the result of shuffled data from the previous stage constitutes the input of the next stage.

Figure 2 is a diagram of how Spark computes job stages. Spark employs a mechanism called "*lazy evaluation*" which means a transformation is not performed immediately. It will wait until the whole computation DAG is built and eventually the execution including that transformation will be triggered by an action in the same DAG. In this scenario, to run an action on RDD G, the Spark system builds stages at wide dependencies and pipelines narrow transformation inside each stage. In other words, narrow dependencies are good for efficient execution, whereas wide dependencies introduce bottlenecks since they disrupt pipelining and require communication intensive shuffle operations.

Fault tolerance. Spark uses the DAG to track the lineage of operations on RDDs. For shuffle dependency, the intermediate records from one stage are materialized on the machines holding parent partitions. This intermediate data is used for simplifying failure recovery. If a task fails, the task will be retried as long as its stage's parents are still accessible. If some stages that are required are no longer available, the missing partitions will be re-computed in parallel. Spark is unable to tolerate a scheduler failure of the driver, but this can be addressed by replicating the metadata of the scheduler.

The task scheduler monitors the state of running tasks and retries failed tasks. Sometimes, a slow straggler task may drag the progress of a Spark job. As the size of cluster and the number of tasks increase, the impact of stragglers become more significant. The task scheduler uses speculative relaunch of straggling tasks in order to reduce the impact of stragglers.

Machine learning on Spark. Spark is not designed specifically for machine learning, however, it has been retrofitted with a machine learning library called MLlib that contains common machine learning algorithms, utilities, and linear algebra operations. In the basic machine learning setup, Spark stores the model parameters in the driver, and the workers communicate with the driver to update the parameters after each iteration. However, for large scale machine learning deployments, the model parameters may not fit into the driver node and they would need to be maintained as an RDD. This introduces a lot of overhead because a new RDD will need to be created in each iteration to hold the updated model parameters. Since updating the model usually involves shuffling data across machines, this limits the scalability of Spark.

3. PMLS PARAMETER-SERVER SYSTEM

PMLS [19] takes a clean-slate approach and starts by identifying the objectives and features of machine learning systems. Most machine learning algorithms work in iterations. In each iteration the system performs computation on the dataset and the current model state and outputs an intermediate result, and secondly updates the model state based on the result. Thus in PMLS, a worker process/thread is responsible for requesting up to date model parameters and carrying out computation over a partition of data, and a parameter-server thread is responsible for storing and updating model parameters and making response to the request from workers. [1]

Figure 3 shows the architecture of PMLS. The parameter server is implemented as distributed tables. All model parameters are stored via these tables. A PMLS application can register more than one table. These tables are maintained by server threads. Each table consists of multiple rows. Each cell in a row is identified by a column ID and typically stores one parameter. The rows of the tables can be stored across multiple servers on different machines. Workers are responsible for performing computation defined by a user on partitioned dataset in each iteration and need to request up to date parameters for its computation. Each worker may contain multiple working threads. There is no communication across workers. Instead, workers only communicate with servers. Note that "worker" and "server" are not necessarily separated physically. In fact server threads co-locate with the worker processes/threads in PMLS.

[1]Originally, PMLS was named Petuum. PMLS consists of *Bösen* sub-system and *Shards* sub-system. In this paper we focus on the Bösen sub-system which provides data-parallel training.

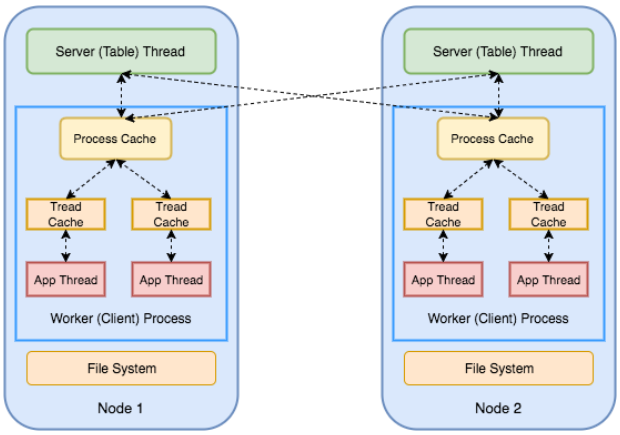

Figure 3: PMLS (Bösen) Architecture

PMLS exploits the error-tolerant property of many machine learning algorithms to make a trade-off between efficiency and consistency. Most machine learning algorithms can tolerate bounded error in their iterative optimization process [12]. In order to leverage such error-tolerant property, PMLS follows Staleness Synchronous Parallel (SSP) [12] model instead of Bulk Synchronous Parellel (BSP) model [18] commonly used in dataflow systems and enables users to set staleness threshold. In SSP model, worker threads can proceed without waiting for slow threads. Fast threads may carry out computation using stale model parameters. Performing computation on stale version of model parameter does cause errors, however these errors are bounded. The usage of SSP model reduces as much the impact of stragglers as possible in a PMLS cluster.

The communication protocol between workers and servers can guarantee that the model parameters that a working thread reads from its local cache is of bounded staleness. With this protocol, PMLS ensures that the fastest working thread can not be s iteration ahead of the slowest working thread, where the staleness threshold s should be configured by a user.

Fault tolerance. Fault tolerance in PMLS is achieved by checkpointing the model parameters in the parameter server periodically. To resume from a failure, the whole system restarts from the last checkpoint. PMLS does not take checkpoint at each iteration, considering the heavy memory and network overhead caused by checkpointing. The checkpointing period is configurable.

Programing interface. PMLS is written in C++. It does not decouple user APIs from its system APIs in its documents explicitly, which means users can use any public methods in a PMLS's core class. To write custom application, a user first needs to define tables to store model parameters. Each table can contain multiple rows of a specific type. One table can be used to store a set of parameters in a machine learning algorithm. By its client APIs, users can access to model parameters in its cache and server threads. For updating parameter, users can either specify an entry of a row to update: $Inc()$ or perform batch update: $BatchInc()$ on a whole row of a table. PMLS system provides $Clock()$ method which is used to inform all servers that current threads has finished a computation round. When $Clock()$ is invoked, the client will release buffered parameter updates to servers. While PMLS has very little overhead, on the negative side, the users of PMLS need to know how to handle computation using relatively low-level APIs.

4. TENSORFLOW

Leveraging their experience with DistBelief [9], a first generation distributed parameter-server system, Google open-sourced TensorFlow [6] in November 2015. Similar to other dataflow systems, in TensorFlow the computation is abstracted and represented by a directed graph. But unlike traditional dataflow systems, TensorFlow allows nodes to represent computations that own or update mutable state. It provides stateful operations *Variable*, which owns mutable buffer, and can be used to store model parameters that need to be updated at each iteration. Nodes in the graph represent operations, and some operations are control flow operations. Values that flow along the directed edges in the TensorFlow graph are *Tensors*, arbitrary dimensionality matrices. An operation can take in one or more tensors and produce a result tensor. In addition, special edges called control dependencies can be added into TensorFlow's dataflow graph with no data flowing along such edges. In summary, TensorFlow is a dataflow system that offers mutable state and allows cyclic computation graph, and as such enables training a machine learning algorithm with parameter-server model.

The Tensorflow runtime consists of three main components: *client*, *master*, *worker*. A client is responsible for holding a *session* where a user can define computational graph to run. When a client requests the evaluation of a Tensorflow graph via a session object, the request is sent to master service. The master then schedules the job over one or more workers and coordinates the execution of the computational graph. Each worker handles requests from the master and schedules the execution of the kernels[2]in the computational graph. The dataflow executor in a worker dispatches the kernels to local devices and runs the kernels in parallel when possible.

If multiple devices are involved in computation, a procedure called *node placement* is executed in a Tensorflow runtime. Tensorflow uses a cost model to estimate the cost of executing an operation on all available devices (such as CPUs and GPUs) and assigns an operation to a suitable device to execute, subject to implicit or explicit device constraints in the graph. TensorFlow supports multiple communication protocols, including gRPC over TCP, and RDMA over Converged Ethernet.

TensorFlow supports sub-graph execution. A single round of executing a graph/sub-graph is called a *step*. A training application contains two type of jobs: parameter server (ps) job and worker job. Like data parallelism in PMLS, TensorFlow's data parallelism training involves multiple tasks in a worker job training the same model on different mini-batches of data, updating shared parameters hosted in a one or more tasks in a ps job.

Figure 4 illustrates a typical replicated training structure called *between-graph replication*, where there is a separate client for each worker task, typically in the same process as the worker task. Each client builds a similar graph containing the parameters (pinned to ps) and a single copy of

[2] The implementation of an operation on a particular device is called a *kernel*.

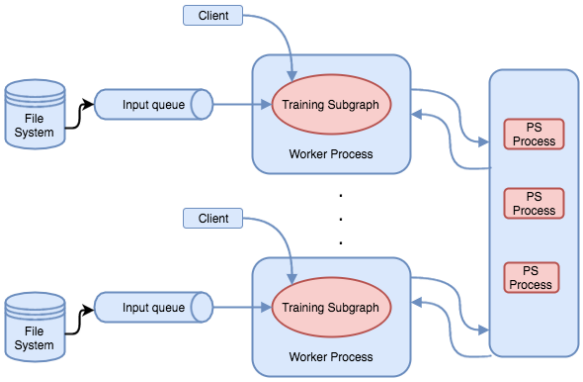

Figure 4: TensorFlow Between-Graph Replicated Training

the compute-intensive part of the computational graph that is pinned to the local task in the worker job. An example of a compute-intensive part is to compute gradient during each iteration of stochastic gradient descent algorithm. Users can also specify the consistency model in the between-graph replicated training as either synchronous training or asynchronous training. In asynchronous mode, each replica of the graph has an independent training loop that executes without coordination. In synchronous mode, all of the replicas read the same values for the current parameters, compute gradients in parallel, and then apply them to a stateful accumulators which act as barriers for updating variables.

Fault tolerance. TensorFlow provides user-controllable checkpointing for fault tolerance via primitive operations: *save* writes tensors to checkpoint file, and *restore* reads tensors from a checkpointing file. Usually a variable is connected to a *save* operation. During the execution of a TensorFlow job, *save* operations are run periodically to produce a new checkpoint. The last checkpoint is used to restart TensorFlow computation. TensorFlow allows customized fault tolerance mechanism through its primitive operations, which provides users the ability to make a balance between reliability and checkpointing overhead.

TensorFlow employs backup workers to mitigate stragglers. The TensorFlow runtime takes the first m of n updates produced at each iteration of the training.

Programming interface. TensorFlow software can be divided into 3 functional layers as in Figure 5. The user client layer provides client APIs in various languages such as Python and C++ as well as language-specific libraries. All the programming APIs in TensorFlow are encapsulated in this layer. TensorFlow provides not only low level math operations APIs, but also many high level operations and optimization algorithms for facilitating machine learning/deep learning. A thin C API layer separates client APIs and TensorFlow core library. The TensorFlow core library is implemented in C++ and contains the core runtime.

Compared with Spark and PMLS, TensorFlow provides more APIs and primitives. That means user can either deploy their machine learning/deep learning algorithms with build-in modules (a set of APIs with a specific purpose) or build their algorithms from scratch by low-level APIs.

5. MXNET

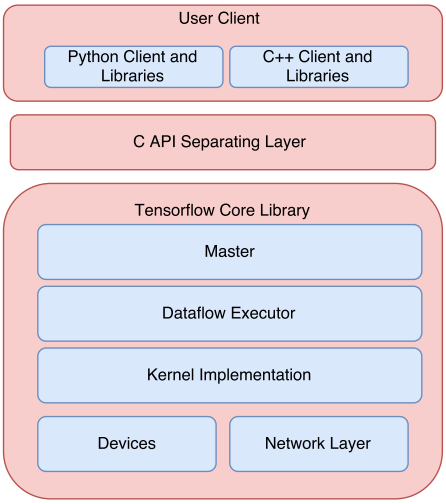

Figure 5: Software Layers in TensorFlow

MXNet [4] is a collaborative open source project that emerged from many deep learning projects such as CXXNet, Minverva, and Purine2 in 2015. Similar to TensorFlow, MXNet is a dataflow system that allows cyclic computation graphs with mutable states, and supports training with parameter server model. Similar to TensorFlow, MXNet provides good support for data-parallelism on multiple CPU/GPU, and also allows model-parallelism to be implemented. MXNet allows both synchronous and asynchronous training [7].

Figure 6 illustrates main components of MXNet. The runtime dependency engine analyzes the dependencies in computation processes and parallelizes the computations that are not dependent. On top of runtime dependency engine, MXNet has a middle layer for graph and memory optimization.

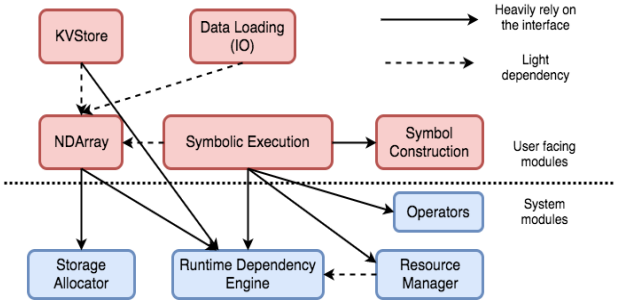

Figure 6: MXNet Components

Fault tolerance. MXNet supports basic fault tolerance through checkpointing, and provides save and load model operations. The save operaton writes the model parameters to the checkpoint file and the load operation reads model parameters from the checkpoint file.

Programming Interface. MXNet uses declarative programming to represent computations in directed graphs and also allows some imperative programming to be used for defining tensor computation and control flow. MXNet provides client APIs written in several languages such as C++, Python, R, and Scala. Similar to TensorFlow, the back-end core engine of MXNet library is written in C++.

6. EVALUATION

In order to provide a quantitative evaluation of Spark, PMLS, TensorFlow and MXNet, we evaluated the performance of these four systems with some typical machine learning tasks: logistic regression and image classification using a neural network. All of our experiments are conducted in Amazon EC2 cloud computing platform using m4.xlarge instances. Each instance contains 4 vCPU powered by Intel Xeon E5-2676 v3 processor and 16GiB RAM. The dedicated EBS Bandwidth of m4.xlarge instance is 750Mbps.

Logistic regression experiments. We implemented a two class logistic regression algorithm on these four platforms. Our synthetically created dataset contains 10,000 data samples and each of the sample has 10000 features. The total size of the dataset is 750MB. On PMLS, we implemented logistic regression using stochastic gradient descent algorithm with batch size 1 and SSP=3. [3] Since Spark is suitable for batch data processing, we used full batch gradient descent (batch size =10000) to train the model. The model parameters are stored in Spark's driver (as they fit there) instead of being stored as RDD (which would kill the performance as we discuss in Section 2). For TensorFlow (TF) and MXNet, we implemented synchronous stochastic gradient descent training with varying batch sizes: 100, 500. The TensorFlow logistic regression was performed by between-graph replicated synchronous training. In these experiments, the cluster of each system contains 3 worker nodes and an extra node is needed to serve as the driver or parameter server (ps) in Spark, Tensorflow, and MXNet. The speed of these systems are shown in Table 1.

For logistic regression experiment, PMLS and MXNet are the fastest two systems and Tensorflow is the slowest one in terms of system speed. Spark comes between them. There are several reasons that lead to this result. First, PMLS is a lightweight system compared to Spark and TensorFlow. It is implemented with high performance C++ programming language compared with Spark which is written by Scala language, running on Java Virtual Machine (JVM). Second, PMLS contains less abstractions compared with TensorFlow which has too many abstractions. Abstractions increase the complexity of a system and lead to runtime overhead.

MNIST image classification experiments. For Spark, TensorFlow, and MXNet, we evaluated an image classification application with different models on MNIST dataset [2]. (This experiment does not include PLMS, as by the time we did this experiment no suitable example code was released from PLMS.) In addition to measuring the efficiency in terms of training speed, we also employed the Ganglia [3] monitoring tool to measure the utilization of CPU, network, memory of both worker and ps node—for Spark ps refers to the driver.

Different models with the same cluster size: We fixed the size of the EC2 cluster to 3 worker nodes and 1 ps node for each system and conducted training with three models: softmax, single-layer neural network (SNN), and multi-layer neural network (MLY), which contains two hidden layers. We used the example codes that are released by these platforms to make the comparison fair. For all the comparison,

[3]The example code of logistic regression in PMLS use batch size 1, and we followed the default setting.

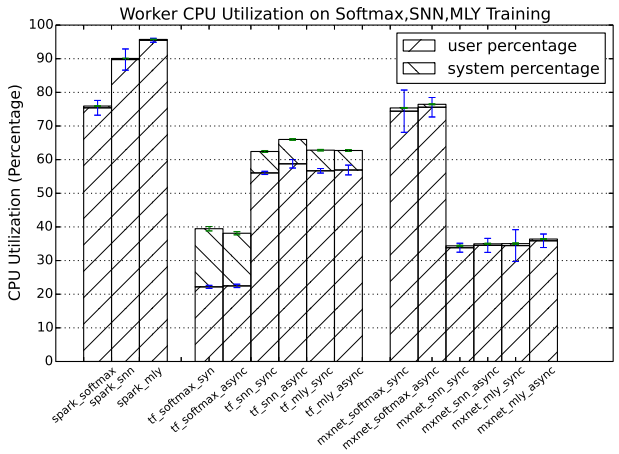

Figure 7: CPU Utilization of Training on Softmax, Single Layer Neural Network, Multilayer Neural Network(2 hidden Layers) 3 workers+1 ps

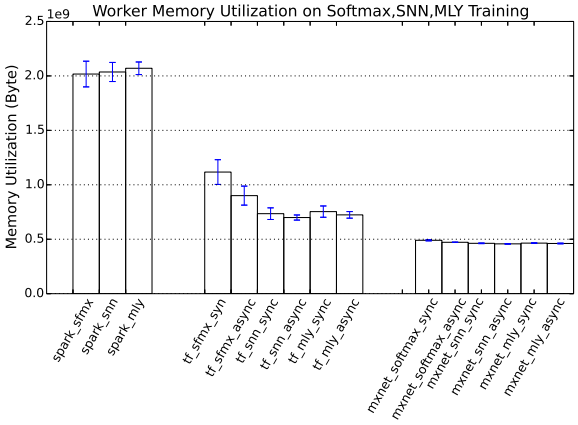

Figure 8: Memory Utilization of Training on Softmax, Single Layer Neural Network, Multilayer Neural Network(2 hidden Layers) cluster size:3 workers+1 ps

we use the same setting and hyperparameters such as learning rate, optimizer, activation function, the number of units in the layer and so on, with only one exception that we used full batch training with Spark. (Using full batch is the default setting in MLlib for training the models as it usually yields the best performance.) For Tensorflow and MXNet, both synchronous training and asynchronous training are implemented. The system speeds of the three systems are shown in the Table 3. We find that MXNet is a little bit faster than Tensorflow except for the asynchronous training on single layer NN and two layers NN. The speed of Spark decreases more significantly than the other two as the model size increases. At the same time, we find that training a larger model in Spark uses more CPU than other two. These two findings indicate a potential CPU-bottleneck for Spark.

Figure 7 and Figure 9, 10, 11 shows the CPU and network utilization performance. Spark uses more CPU than the other two systems but less network per worker. This finding

Table 1: System Speed of Logistic Regression Training (Samples/second)

	Spark	TF_100	TF_500	Mnxet_100	MXNet_500	PMLS_ssp3
System Speed	5,883	403	443	19,277	19,283	21132

Table 2: PMLS Speed with Different SSP

	SSP=0	SSP=3	SSP=6
Speed (records/sec)	20958	21132	21353

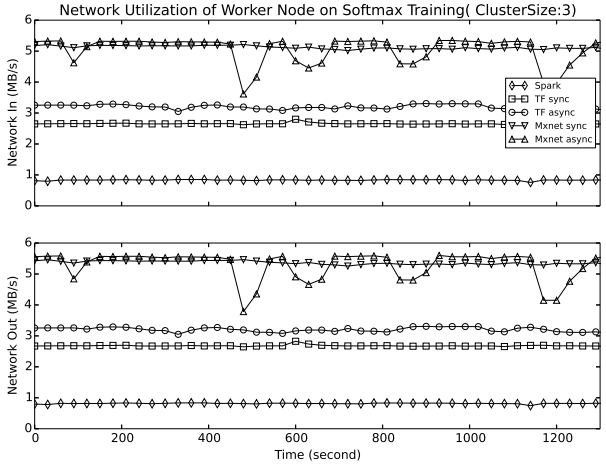

Figure 9: Network Utilization of Softmax Training on the Three Systems, Cluster size =3 (3 workers +1 ps)

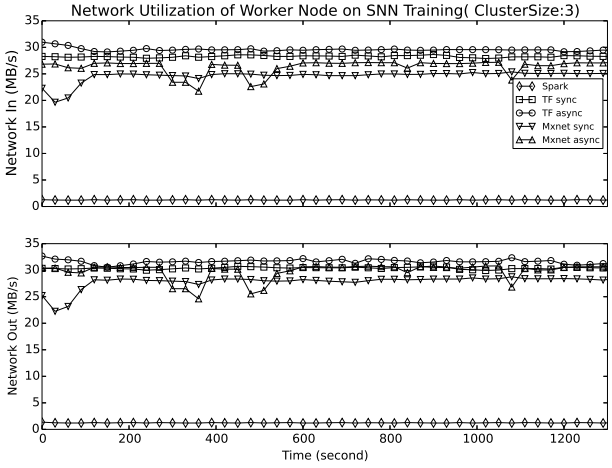

Figure 10: Network Utilization of Single Layer Neural Network (SNN) Training on the Three Systems, Cluster size =3 (3 workers +1 ps)

is in accordance with that in [17]. Researchers found that serialization and some collection operations are CPU intensive in Scala on which Spark is built [5]. Spark's low network utilization is due to the less data communication between the driver and the workers in a given period of time. In this experiment, Spark uses full batch training. Tensorflow and MXNet use a mini-batch size of 500 to train the model. The data communication [4] frequency of Tensorflow and MXNet is much greater than that of Spark. Therefore, given a period of time the worker of Tensorflow and MXNet pushes and pull more data chunk to and from the PS than that of Spark, thus leading to a high network utilization. MXNet uses less CPU when facing a relative bigger model, and we are unable explain this counter-intuitive result.

Figure 8 shows the memory utilization. Spark's memory utilization is relatively higher than Tensorflow and MXNet. This is because Spark needs to cache the whole training data as RDD during the training process. By contrast, for Tensorflow and MXNet the memory utilization of storing data is smaller, as they only need to keep a mini-batch of training data in memory.

Different cluster sizes with the same model: In order to evaluate how the system performance changes as we scale the cluster size, we conduct SNN synchronous training with 1 worker, 3 workers, and 5 workers. Each of the three systems has one ps node. Figure 12 shows the system speed change and the per worker speed change as we increase the size of the cluster. The cost of synchronization for Tensorflow and MXNet is higher than that for Spark. For Tensorflow and MXNet, we can see the per worker speed decreases more significantly as we increase the number of workers.

Figure 13 shows the CPU utilization of per worker and ps as we increase the number of workers. We can see the CPU utilization of the worker is highly correlated with the training speed of worker. The CPU utilization of ps node increases as we increase the number of workers in the cluster. This is because the ps CPU needs to serve more network I/O system calls as the size of the cluster increases.

Figure 15 shows the network utilization of per worker and ps as we increase the number of workers. One thing to note is Spark uses the least network because Spark employs full batch training and has a low frequency of transferring data between worker and ps. Figure 14 shows the memory utilization of both ps and worker. The memory utilization of Spark is twice as high as Tensorflow and MXNet in the SNN synchronous training. In general, MXNet uses the least memory in both worker node and ps node.

7. CONCLUDING REMARKS

In this work we investigated the architectural design decisions in distributed machine learning platforms and their impact on scalability, performance, availability, and even usability of these platforms. We find that for complex machine learning tasks, and especially for training deep neural networks, the basic dataflow model fails to scale due to its lack of support for mutable state and fast iterations. The parameter-server model addresses these requirements and has been adopted widely by the machine learning and deep

[4]In our experiment, the data communication happens when the worker pushes gradient to the ps and fetch the up-to-date model parameters

Table 3: System Speed with Different Models (Images/second)

	Spark	TF Sync	TF Async	MXNet Sync	MXNet Async
Softmax	424,600	113,787	149,759	221,853	259,076
Single Layer NN	24,303	23,485	27,099	24,383	25,644
Two Layers NN	9,133	15,283	22,206	16,122	18,511

Note: system speed means how many images per second can be processed.

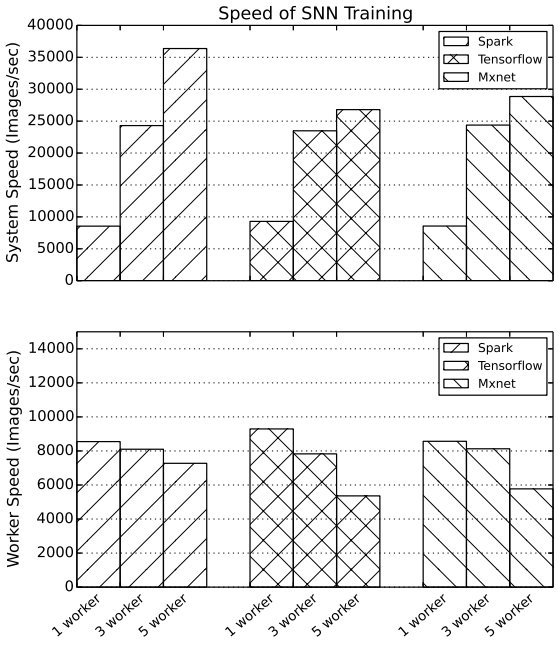

Figure 12: Scalability of the three systems in terms of image process speed

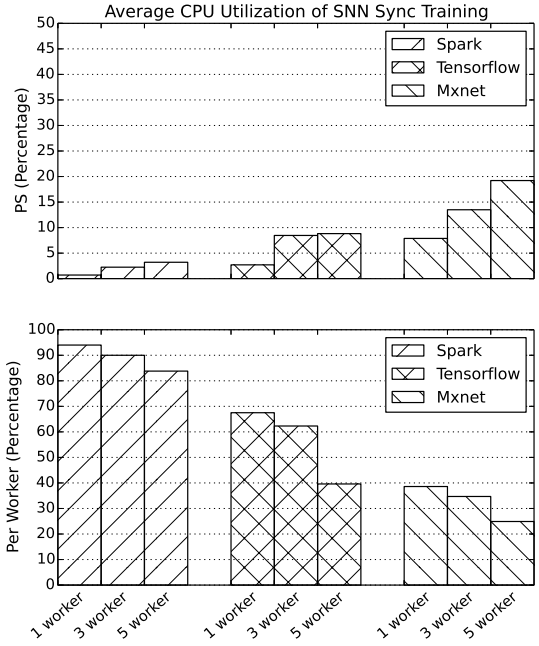

Figure 13: Average CPU Utilization With Different Cluster Size

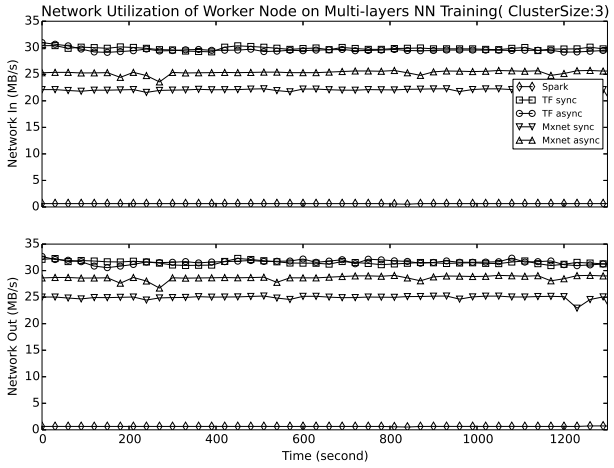

Figure 11: Network Utilization of Multi-layers NN Training on the Three Systems, Cluster size =3 (3 workers +1 ps)

learning platforms. More advanced dataflow systems are developed to allow cyclic execution graphs with mutable states in order to support the parameter-server model.

Below we present some promising directions for future work where the distributed systems researchers can contribute for improving the performance of machine learning and deep learning platforms. Converting the execution graph to an optimized distributed execution is a promising area for research. More advanced graph partitioning techniques that stage the computation graph over the underlying distributed nodes in a fashion that avoids potential bottlenecks would be beneficial. Furthermore, the distributed systems can provide some dynamic scheduling support to the machine learning platforms running atop. Currently, even in TensorFlow and MXNet, the number of parameter-server and workers–and even the number of processes inside a worker– need to be explicitly specified by the application developer. However, the underlying distributed system can monitor/profile the application at runtime and provide an informed provisioning of computation, memory, network resources to the tasks running atop. Finally research is needed on varying degree of consistency for the parameter-server model and on more effective parameter-server worker synchronization mechanisms.

8. REFERENCES

[1] http://spark.apache.org/mllib/.

[2] http://yann.lecun.com/exdb/mnist/.

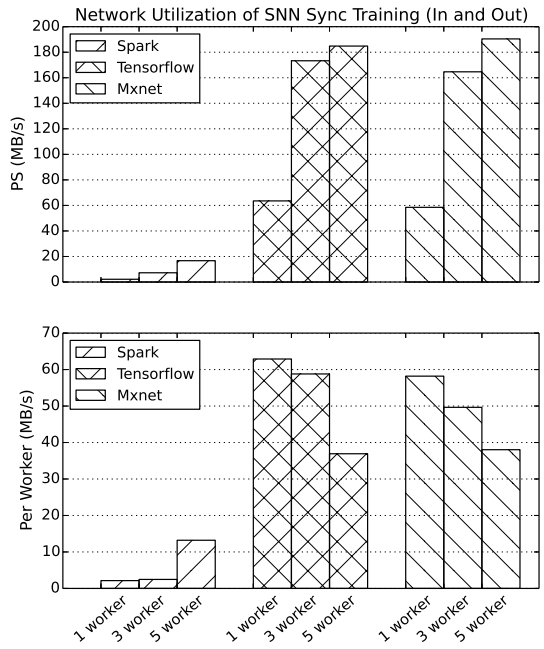

Figure 15: Average Network Utilization with Different Cluster Size

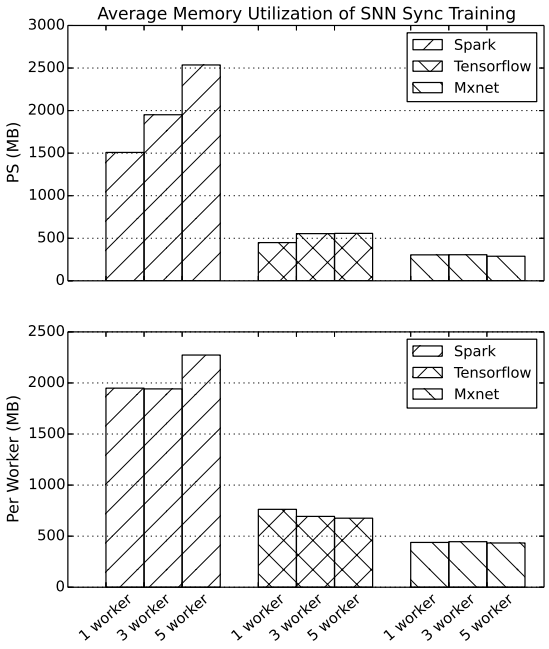

Figure 14: Average Memory Utilization with Different Cluster Size

[3] http://ganglia.sourceforge.net/.

[4] http://mxnet.io/architecture/overview.html/.

[5] http://www.cl.cam.ac.uk/research/srg/netos/camsas/blog/2015-07-08-timely-pagerank-part1.html.

[6] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, et al. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *arXiv preprint arXiv:1603.04467*, 2016.

[7] T. Chen, M. Li, Y. Li, M. Lin, N. Wang, M. Wang, T. Xiao, B. Xu, C. Zhang, and Z. Zhang. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. *arXiv preprint arXiv:1512.01274*, 2015.

[8] D. E. Culler. Dataflow architectures. Technical report, DTIC Document, 1986.

[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*, pages 1223–1231, 2012.

[10] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[11] A. Halevy, P. Norvig, and F. Pereira. The unreasonable effectiveness of data. *IEEE Intelligent Systems*, 24(2):8–12, 2009.

[12] 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*, pages 1223–1231, 2013.

[13] 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*, volume 41, pages 59–72. ACM, 2007.

[14] 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 *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 583–598, 2014.

[15] 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*, pages 439–455. ACM, 2013.

[16] D. G. Murray, M. Schwarzkopf, C. Smowton, S. Smith, A. Madhavapeddy, and S. Hand. Ciel: a universal execution engine for distributed data-flow computing. In *Proc. 8th ACM/USENIX Symposium on Networked Systems Design and Implementation*, pages 113–126, 2011.

[17] K. Ousterhout, R. Rasti, S. Ratnasamy, S. Shenker, B.-G. Chun, and V. ICSI. Making sense of performance in data analytics frameworks. In *NSDI*, volume 15, pages 293–307, 2015.

[18] L. G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, 1990.

[19] E. P. Xing, Q. Ho, W. Dai, J. K. Kim, J. Wei, S. Lee, X. Zheng, P. Xie, A. Kumar, and Y. Yu. Petuum: a new platform for distributed machine learning on big data. *Big Data, IEEE Transactions on*, 1(2):49–67, 2015.

[20] 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*, pages 2–2. USENIX Association, 2012.

[21] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster computing with working sets. *HotCloud*, 10:10–10, 2010.