Distributed Machine Learning Using Apache ZooKeeper

Rishikesh Ingale, Han Li, Yixiang Wang
University of California San Diego

1 Abstract

In this paper, we compare the efficiency of two different distributed machine learning systems, the Parameter Server and Horovod. Both systems boast achieving the best convergence speed despite using different machines (i.e. different CPU RAM and network cards, and different datasets). We implemented a form of each system using ZooKeeper and measured their performance using the same machines and learning task. We describe these two systems, how we implemented them using ZooKeeper, and the experiments performed as well as their results. The experiments involved solving a simple classification problem between two classes of the MNIST dataset.

2 Introduction

Today's machine learning model has grown so much, in terms of the number of parameters, that one single machine might not be able handle such a large training process. Moreover, the data might be collected on different machines. Therefore, if one tries to combine all the pieces of data together, the size of the total data might be hundreds of petabytes–a quantity that a single machine's file system might not be able to handle.

Training across multiple servers solves this problem; however, with parallelization comes an expectation to be faster than training on one machine. We examined two distributed system architectures on which machine learning can be executed and determine which is the best option for a specific learning model. Both of these systems claim that they achieve the best convergence speed, but the results were obtained using different hardware. We implemented a form of each of these systems on the same hardware and coordinate processes using Apache ZooKeeper [3]. Like any distributed machine learning system, both systems have workers which carry out the training process, so we made sure to give each system the same goal of replicating the same model across their workers. This has real-world applications: more often than not, predictive models are offered as services (MLaaS) and it is useful to replicate those models across different servers so that one is not overflowing with requests from all around the world and there is no single point of failure. The purpose of this paper is to answer the following questions:

1. Which of the two systems is better suited for a simple classification problem?

2. Does ZooKeeper aid our efforts, or introduce more latency?

3 Overview

In this section, we provide some background knowledge on the coordination service integral to our software artifact, the algorithm that will run on multiple processors, and the two systems that we have decided to implement.

3.1 Apache Zookeeper

Zookeeper is a service that can coordinate different processes in a distributed application. It incorporates elements from distributed locks, message broadcasting, and shared registers to form a replicated, centralized service.

Distributed applications have the need for coordination in many ways. Configuration is one of those ways as all machines or processes may need to run on the same settings. Group leader election and group membership also play a big role in many distributed systems in order to change view and keep the system running. Some form of a lock service is used in almost all distributed applications to keep at least some level of consistency. Many need to implement all three of these coordination primitives, and some need even more. Zookeeper differentiated itself by exposing an API that can be used by developers to implement their own primitives for their own need. Ideally, developers only need this one coordination service and personalize it to serve multiple purposes.

The consistency model Zookeeper implements is called A-linearizability. It guarantees two things: firstly, all writes from a single user are in order and secondly, all the operations are serviced in some total order. It is a relaxed version of release consistency as read operations from a single user are not guaranteed to be serviced in order. As it turns out, this relaxation is much more efficient and do not have any major consequences as distributed applications generally do not need to read in a strict order. Note that complete release consistency can also be achieved by Zookeeper with some implementation tricks.

The basic layout of an application using ZooKeeper will contain at least two layers. The first layer contains clients (traditionally known as servers in a distributed service), where data is distributed and replicated for durability. The second layer contains the ZooKeeper servers, which maintain a tree structure inspired by file systems that consists of multiple wait-free data objects called znodes. Generally metadata will be stored in these objects for coordination. Just like a file system, Zookeeper has a notion of directories, and files within a directory cannot be duplicated. The broadcasting of a write to the ZooKeeper servers is called Zab, which stands for ZooKeeper's atomic broadcast. Clients send their operations to the leader ZooKeeper server, which finds a quorum and uses Zab to execute the operation on all servers. Servers will ultimately perform all the operations in the given order.

One thing to note is that ZooKeeper, because of its atomic broadcast, can be a slow service. Therefore, we intend to carry out the experiments using only one ZooKeeper server.

3.2 Training

A common problem in computer vision is training a model to differentiate between two classes, otherwise known as a binary classification problem. For our experiments, we used the MNIST dataset [1], the classes of which are the digits zero to nine, where each class contains multiple 28-by-28 images (vectors of length 784) of the handwritten digit. For the sake of simplicity, we will be differentiating between the digits zero and one by leveraging a logistic regression model. Training this type of model requires performing the following update step for some number of times:

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

where θ represents the parameter vector, h_θ represents the activation unit (the function that does the classification given the parameter vector and training examples), and $(x^{(i)}, y^{(i)})$ is the ith image/label pair in a list of m pairs. Should training be done on a single worker, the worker simply does the above for however many iterations specified over the entire training set. However, if the training set was split across n workers, then each worker would end up with a different set of parameters after the first iteration. Thus, after each iteration, the sum of n subtrahend vectors should be computed and sent to each worker to subtract from its parameter vector before the next iteration of updates begin.

3.3 Parameter Server

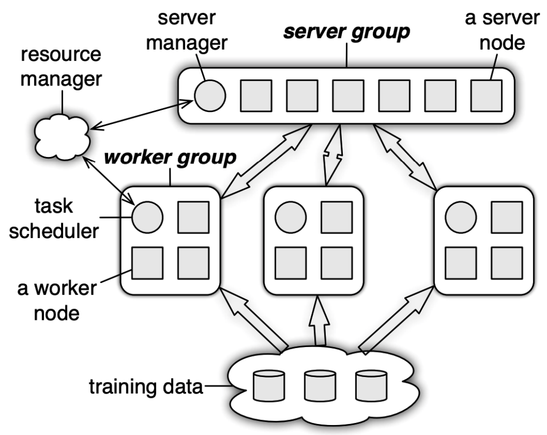

Figure 1: Architecture of the Parameter Server

The Parameter Server [2] is a system which divides nodes into servers and workers. Server nodes are in charge of maintaining the global model parameters, which includes serving clients as well as workers after each iteration of training. Each server node maintains a subset of the parameters. Server nodes also communicate with each other to replicate updated parameters for reliability and scaling, since each server node may join or leave at will. The server manager maintains a mapping of parameters to nodes and is responsible for knowing if a server node is live or not. Training data is split amongst the worker nodes, so naturally the training workload is divided just as equally. The task scheduler is responsible for assigning a range of data and keeping track of the training progress. If a worker node exits without finishing its current task, the task scheduler is responsible for finding another node to repeat this task. Thus, the Parameter Server is a fault-tolerant distributed system.

This system also allows for various consistency models for people to choose from. Usually, to start training iteration i, machine learning model needs to apply the gradient calculated in iteration $i - 1$ beforehand. This slows down the training progress if one worker has completed iteration $i - 1$ and waits for other workers to finish that same iteration. The Parameter Server supports another consistency model called "bounded delay", in which a task will be blocked until all the previous tasks initiated η time ago have all finished. If $\eta = 0$, the system essentially achieves sequential consistency. If $\eta = \infty$ then there are no dependencies between tasks.

3.4 Horovod

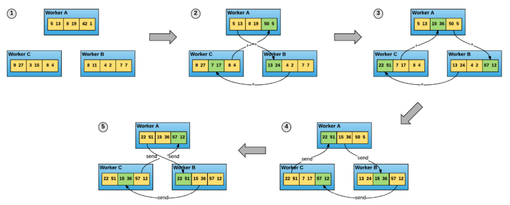

Figure 2: Ring-allreduce Algorithm

Horovod [4] is a system that only has worker nodes, designed for inter-GPU communication so that clients can utilize multiple GPUs to increase training speed. Similarly, training data is split amongst the nodes and the workload is divided just as equally as in the Parameter Server system. The motivation behind Horovod is that it is more scalable than the Parameter Server. When dealing with a high number of workers, network and computation ability of the server group in the Parameter Server system could become a bottleneck, especially with a low server-to-worker ratio. Moreover, sending millions of parameters in deep learning (multiple GB of data) all at once would be inefficient and time-consuming. As a result, the nodes utilize the Ring-allreduce algorithm. Consider nodes A, B, and C. A passes its gradient computation to B, so B now has the gradients from A and itself. B then passes the sum to C, which now has the gradient vector with respect to all the training data. C then sends this vector to A and B, and all nodes are fully updated. This is just a process of only one group of parameters. For other groups of parameters, it would also follow the same process. Note that in Horovod, the passing activity of all the groups of parameters are happening simultaneously, which means no idle time for all the workers. The total number of updates or passing gradients between workers is $O(N^2)$ where N represents the number of workers in the Horovod system. This system is scalable, but nothing regarding its fault tolerance is mentioned.

4 Implementation

Our implementation relies on Apache ZooKeeper, which maintains the shared hierarchical namespace that our processes will use for coordination. It guarantees reliability and its transactions to be ordered. Given our goals and the questions that we want to answer, we are going to assume no node failures will occur. Furthermore, we will be using one ZooKeeper server for both implementations, so that we avoid most of the latency caused by Zab. We also verified that the training yielded the same parameters across workers.

4.1 Parameter Server

We have written two programs: one for server nodes and one for worker nodes. In our simulation, the server group will only have one node which will act as its own manager as well as the task scheduler for the worker nodes. ZooKeeper is the resource manager as well as the primary mode of communication between the server and worker nodes. The server node takes in as arguments the number of workers, the number of training iterations, and the address/port pair of the ZooKeeper server. The worker node takes in as arguments a worker ID, the number of training iterations, its data file, and the address/port pair of the ZooKeeper server.

The training process is kicked off with the server node creating one znode per worker and one znode for itself ("/m"). A worker is able to identify its znode by appending its worker ID to "/w". At the beginning of each iteration, the server node sets watches on the worker znodes using ZooKeeper's asynchronous API and creates a "/start[iteration number]" znode. Each worker blocks until it confirms the start znode's existence, after which it sets a watch on the server node's znode (also using ZooKeeper's asynchronous API), compute its gradients, and writes to its own znode. Upon receiving an update notification, the server node adds the gradient of the znode to its accumulated gradient (which has an initial value of the zero vector).

Once the server node has received the gradients from all worker nodes, it writes the accumulated gradient to its znode, which sends an update notification to the workers. Each worker then reads the accumulated gradient, performs the parameter update, and creates an "/ack[worker ID]" znode. The server node blocks until it confirms the existence of all acknowledgements, after which it creates an "/end[iteration number]" znode. Each worker blocks until it confirms the end znode's existence, after which it proceeds to the next iteration.

Over the course of the entire training process, the server and worker nodes delete any znodes that no longer serve any purpose, namely the znodes that have an iteration number or a worker ID appended to them (start, end, ack).

4.2 Horovod

In our Horovod design, we only needed to write one worker class that would synchronize the gradients between each other. Since we built our programs to use the hierarchical namespace of ZooKeeper, the entire gradient vector is accumulated when passed along to subsequent workers during the update phase, whereas in the original Horovod system, since workers needed to communicate through the network, the gradients were divided in parts before being passed sequentially.

Initially, each worker will create a znode with the path "/w[worker ID]" and listen to the previous znode in the order (otherwise known as the ring). This means worker2 listens to znode1 and worker1 listens to the znode with the largest ID, if znode IDs start at 1. Then, each worker will first start a training process by executing a gradient computation script that does the training and outputs the gradients with respect to the worker's portion of the data to a file. Once a worker finds that its child process has finished, it reads the gradient from the resultant file and stores the gradient to its znode, notifying the next worker. If the next worker is still waiting for its own gradient computation to finish executing, it would still read the gradient from the previous worker and store the accumulated gradient to its znode once the child finishes. If the next worker's gradient computation finishes and it has been waiting for gradient of previous worker, it would immediately write the accumulated gradient to its znode, which triggers an alert to the worker listening to that znode. This is made possible using ZooKeeper's asynchronous API. As an example, worker2 will first read the gradient vector stored in znode1, add it to its own gradient vector, and write to znode2, triggering an alert to the worker listening to znode2. The next znode, znode3, will do the same, triggering an alert to the next znode, and this process continues until the last worker.

At the last worker, the result of adding its gradient yields the gradient with respect to all of the data. It will first store this final gradient to its own znode, which worker1 is alerted of since it had set a watch, and then updates the parameter vector for its next gradient computation process to read and continue training. The rest of workers will read the final gradient from their previous worker's znode and update their parameter vector for their next gradient computation process to read and continue training.

Once each znode has successfully updated its copy of the parameter vector, all of the workers will start the next iteration of the training process. To do this, each worker will set a znode of path "/start[worker ID]epoch[current iteration]". Each worker will check the existence for such a path from all the worker IDs and start the next iteration of the training process upon finding all the paths. This entire process is repeated for all iterations.

5 Evaluation

For both systems, we ran the logistic regression algorithm for 25 iterations on 1, 3, 5, and 10 workers. We timed various aspects of each system for each worker and iteration, averaged them across iterations and workers (in that order), and examined their relationship with the number of workers. The experiments were run on an Amazon EC2 c4.4xlarge instance (16 virtual CPUs and 30 GiB of memory) with an Amazon Linux 2 AMI (x86_64). We parallelized the components of each system using GNU Parallel [5].

5.1 Parameter Server

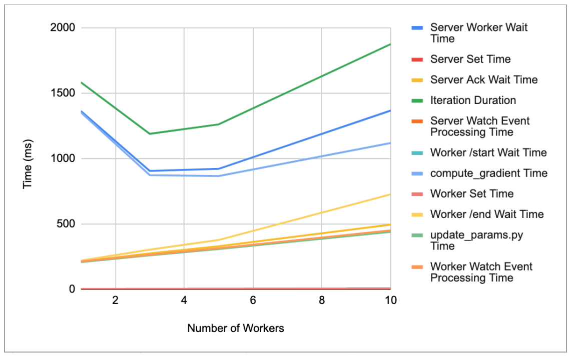

Figure 3: Parameter Server Results Graph

From Table 1, it is immediately noticeable that using ZooKeeper to build a distributed Parameter Server system is slow since all of the measured times are in the order of at least milliseconds. However, ZooKeeper is not directly responsible for the most of the figures in the table, since setting data in the ZooKeeper server only takes at most 10 milliseconds.

One can notice a direct correlation between the "Server Worker Wait Time" and "compute_gradient.py Time" rows as well as one between the "Server Ack Wait Time" and "update_param.py Time" and "Worker Watch Event

	1 Worker	3 Workers	5 Workers	10 Workers	Average
Server Worker Wait Time (ms)	1364	906	922	1368	1140
Server Set Time (ms)	3	3	3	4	3.25
Server Ack Wait Time (ms)	213	276	330	495	328.5
Iteration Duration (s)	2	1	1	2	1.5
Server Total Time (s)	40	30	32	47	37.25
Server Watch Event Processing Time (ms)	2	2	2	4	2.5
Worker /start Wait Time (ms)	4	4	5	8	5.25
compute_gradient.py Time (ms)	1354	874	867	1120	1053.75
Worker Set Time (ms)	3	4	5	8	5
Worker /end Wait Time (ms)	220	304	378	727	407.25
Worker Total Time (s)	40	30	32	47	37.25
update_params.py Time (ms)	209	260	309	440	304.5
Worker Watch Event Processing Time (ms)	214	265	315	452	311.5

Table 1: Parameter Server Results Table

Processing Time" rows, which can also be confirmed by Figure 3. Therefore, it can be inferred that the root cause of the majority of the time cost is due to child process creation, execution, and destruction. Another thing to note is the training time versus the number of workers. It seems that there is an optimal number of workers for training this machine learning model (3 workers). Adding more workers after that point does not make the training faster. The best explanation for this is that the time saved by reducing the amount of data on each worker is less than the time spent by creating, executing, and destroying additional child processes. On average, child process creation, execution, and destruction account for 91% of the training process which equates to an average of 33.7 seconds.

It is very unlikely that the extra latency is caused by gradient calculation as the table shows a general decrease. On the other hand, the table shows a general increase in the parameter update time, the cause of which is unknown since the time to update parameters does not depend on the number of workers. The time spent waiting on the accumulated gradient does depend on the number of workers, since the server accumulation of the workers' gradients is synchronous (see "Worker Watch Event Processing Time" in Table 1).

5.2 Horovod

The first thing to note is that adding more workers didn't necessarily improve the overall training time. Horovod with only one worker spends more time running Python programs than Horovod with more workers. This meets our expectation since each Python program needs to deal with less amount of data with more workers. Also, the

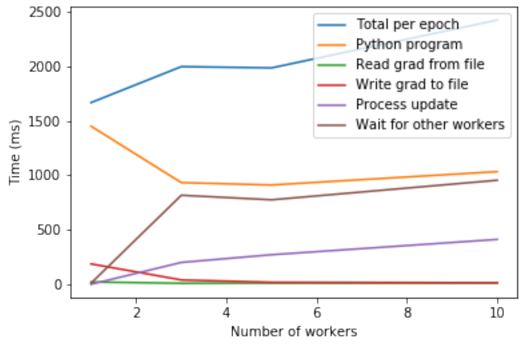

Figure 4: Horovod Results Graph

time that each worker uses to process updates increases with the number of workers due to the fact that there are more znodes created in the ZooKeeper server, and so the time to deal with read/write requests increases. The "Processing Update Time" row in Table 2 includes the time to read gradients from a znode and to write gradients to another znode.

Another thing to note is that the total time of running the training process on Horovod is almost comparable with the time for executing all of the Python programs. We implemented the Horovod workers to use Apache Zookeeper and decided to pass all the gradients together at once to subsequent workers. So, while two workers are working on passing the gradients, all the other workers just wait and have nothing to do. Note that in the original Horovod design, the gradient vector is split into parts amongst the workers. Each worker is responsible for the initiation of passing its own part of the gradient. In other

	1 Worker	3 Workers	5 Workers	10 Workers	Average
Average Time Per Epoch (ms)	1667.47	1997.57	1985.79	2425.2	2019.00
Python Computing Time (ms)	1450.27	932.53	910.96	1033.46	1081.81
Reading Gradient From File Time (ms)	20.75	8.67	11.41	12.48	13.33
Write Gradient To File (ms)	186.52	39.31	17.70	13.35	64.22
Processing Update Time* (ms)	0.0	200.17	270.79	411.32	220.57
Idle Time (ms)	9.93	816.89	774.93	954.55	639.07

Table 2: Horovod Results Table

	Average Total Time (s)
train.py	2.5
Parameter Server	37.5
Horovod	50.5

Table 3: System Comparison Table

words, in the original Horovod design, all the workers must have something to do at any time, whether it is sending gradients to the next worker or receiving gradients. In the original Horovod design, each worker needs to do $O(N)$ updates, where N represents the total number of workers, while in our simplified design, each worker just has $O(1)$ updates. However, our design, which was expected to be more efficient because of less updates, yields a worse performance since there is too much idle time for each worker. Moreover, since the time to write and read gradients is negligible when compared to the waiting time, our Horovod design with less updates doesn't show too much improvement.

5.3 Systems Comparison

Our first question was which system was better suited for a simple classification problem. From the results obtained, the Parameter Server outperformed Horovod when considering the total time for the training process. The most probable cause of this is because reads within a single iteration do not have to follow a set order like in Horovod. In the Parameter Server, the server node read the gradients of the workers in FIFO order, and the worker nodes also read the accumulated gradient in FIFO order (guaranteed by ZooKeeper). In Horovod, the order was determined by the worker IDs, so the wait times were not parallel. Were the Parameter Server system configured with multiple server nodes, additional latency would be introduced due to an increased number of znodes and API calls. It is unclear whether that would be enough to make the system slower than Horovod.

Our second question was whether ZooKeeper would decrease training time. Running a simple Python script to train a logistic regression model on all of the data on the same EC2 instance took 2.5 seconds. Compared to the results obtained from both distributed systems implemented, this is just short of twenty times faster than either system. As mentioned in the introduction of this paper, the main reason to look towards distributed machine learning is the amount of data. In our case, the data was able to fit on a single machine. Therefore, there was no need to separate computing gradients and updating parameters into two separate scripts and invoke child processes. There was also no need to deal with network latency or consistency issues.

6 Future Work

Due to the fact that the data sets machine learning models need to handle today are growing exponentially, training on a single machine is not desirable as doing so cannot benefit the fast growing cloud computing industry. However, due to the limited time we have, we only explored the surface of distributed machine learning. If we had the opportunity, we would like to expand this work further.

6.1 No Idle Time Per Worker

As described in the Horovod section, our implemented Horovod worker passes all the parameter gradients at one time, which results in too much idle time per Horovod worker. It would be interesting to see whether all workers passing gradients partially like the original Horovod design would achieve better performance since it takes negligible time to read from and write to znodes; currently our implemented Horovod worker spends too much time waiting for other workers.

6.2 Bounded Delay in Parameter Server

Bounded delay is another consistency model that is supported by the Parameter Server in the original paper. To reiterate, there is a variable called η. Tasks should only be started once all the tasks initiated η iterations ago have all finished. $\eta = 0$ means the system achieves sequential consistency, while $\eta = \infty$ means no dependency at all. In our implementation, we just follow the case where $\eta = 0$, which results in a lot of waiting time per epoch. However, if we increase this value, some workers may start their tasks for iteration i before other workers finish iteration $i - 1$. We might sacrifice some model performance but we predict less waiting time per iteration. Moreover, it is also interesting to find an optimal η value by which we sacrifice a little model performance but gain a lot in saving the overall training time.

6.3 Multiple ZooKeeper Servers

In our current setup, we only utilized one ZooKeeper server. A single ZooKeeper server is enough to demonstrate the possibility of performing distributed machine learning and the communication overhead which the ZooKeeper service introduced. However, this approach does not utilize the full power of the ZooKeeper service as znode data is not replicated. So, it is very possible that this single ZooKeeper server can be the bottle neck of a larger setup of our experiments as it would need to handle all of the operations sent from different clients. Not replicating also means that if this single ZooKeeper server shuts down, training could be stopped prematurely. In the situation where each iteration of training could take a long time and consume lots of computation power, no duplication/fault tolerance is far from ideal.

6.4 Multiple Server Nodes

For the Parameter Server experiments, we used a single server to communicate with all workers as a demonstration of how this should work. Because of this setup, this single server can be under a lot of network pressure if the number of workers is too high, so alleviating this pressure may lead to better performance in that scenario. It also may not, as mentioned earlier, due to the additional latency caused by creating more znodes and making more API calls. Multiple servers can also be responsible for different but overlapping parts of the accumulated gradient, thereby becoming more fault tolerant, like the Parameter Server architecture described in the paper.

6.5 Other Coordination Services

Zookeeper is a very popular coordination service because of how customizable it is. Developers can simply utilize the API it provided to make a coordination service of their own need. Although this feature is very user-friendly, ZooKeeper may not be the best service for our needs because of its atomic broadcasting. In order to achieve even better performance, we would like to explore other coordination services as well to see which one works the best. A highly specialized in-house solution may be the ultimate goal, but the skills and time needed to achieve that is beyond our reach.

6.6 Support for Neural Networks

Neural networks are very popular in the scope of machine learning today because of all the advances and improvements on accuracy made in fields such as computer vision and natural language processing. Moreover, in our current implementation, our python program just trains a simple logistic regression which takes negligible time to compute predictions and gradients. However, it takes much longer time for a neural network to make predictions and perform back-propagation. Thus, with a neural network, we might see both Parameter Server and Horovod beats the single-core python program as dividing the dataset significantly reduce the time of Python programs.

However, this comes with its own challenges. Whereas a logistic regression model has one activation unit, a neural network has multiple layers of multiple activation units. So, the parameters are not organized into one vector, but rather multiple matrices. One possible way to handle this in ZooKeeper is to have one znode per matrix under each worker znode. A matrix with m rows and n columns can be serialized as a vector in which the first two elements are m and n, and the rest are the elements listed row by row. With this approach, multiple nodes can effectively communicate with each other but serializing and deserializing a large model will introduce more latency.

7 Conclusion

To reiterate, both systems performed worse than running a single script that takes in all of the training data. The majority of the time cost for both systems was child process overhead. One might reason that if the majority of the time cost is caused by this, the Java parent process could do the training itself. However, this would eliminate the benefit of vectorization and would make training slower, especially for large models.

On the topic of comparing both distributed systems, the Parameter Server has the edge in terms of training time because of more parallelization made possible by a central node. Were there more server nodes, the centralized aspect would still exist, but there would be more latency due to replication and more reads and writes.

One of the lessons to be taken away from this project is the realization that the ideal number of processes for a training job is a function of the amount of data. There can be a scenario in which a large dataset can fit onto a single machine, but using no more than three nodes to do distributed training is quicker than running one training script. If the dataset is just a few data points too large for one machine, then it is possible that distributed training among two nodes can be quicker than distributed training among three. Ultimately, there is a point where the time spent creating, executing, and destroying child processes is greater than the time saved by parallelizing training.

Another lesson to be taken away from this project is the importance of parallelizing processes as much as possible when seeking to speed up a certain process. This can be achieved by eliminating the need to rely on a strict order for reads.

To summarize, distributed machine learning is ideal only for large datasets and must be as asynchronous as possible to guarantee faster training.

References

[1] LeCun, Y., Cortes, C., and Burges, C. Mnist handwritten digit database. vol. 2.

[2] Mu Li, David G. Andersen, J. W. P. A. J. S. A. A. V. J. J. L. E. J. S. B.-Y. S. Scaling distributed machine learning with the parameter server. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication* (2014), OSDI '14.

[3] Patrick Hunt, Mahadev Konar, F. P. J. B. R. Zookeeper: Wait-free coordination for internet-scale systems.

[4] Sergeev, A., and Balso, M. D. Horovod: fast and easy distributed deep learning in tensorflow.

[5] Tange, O. Gnu parallel 20210522 ('gaza'), May 2021. GNU Parallel is a general parallelizer to run multiple serial command line programs in parallel without changing them.