Parameter Server for Distributed Machine Learning

Hongyi Sun[1], Xinyu Kang[1], Boxiang Wang[1], Haocheng Yang[1], and Ting Cheng[1]

[1]Harvard University

May 8, 2023

Abstract

Distributed machine learning has become increasingly important in recent years as the size of datasets and the complexity of machine learning models continue to grow. One of the key challenges in distributed machine learning is efficiently managing the communication and synchronization of model parameters across multiple nodes. The parameter server architecture is a popular approach that addresses this challenge by centralizing the storage and management of model parameters on a dedicated server, which can be accessed and updated by worker nodes synchronously. In this project, we explore the parameter server architecture for distributed machine learning, with a focus on its implementation and performance in the context of a convolutional neural network. Specifically, we design and implement a parameter server using Ray and evaluate its scalability and efficiency under various scenarios. Our results indicate that the use of parameter server architecture can address the two major challenges of traditional machine learning: the problem of training data being too large to fit into a single machine's memory, and the limitation of computation power on a single machine. Our experiments show that the parameter server architecture can effectively distribute the workload among multiple machines, reducing the computation time required for large-scale datasets and complex models. This approach can provide useful guidance for designing and optimizing distributed machine learning systems.

1 Background

Machine learning has achieved remarkable success in many fields, such as image classification tasks, natural language processing, Ad click prediction, and speech recognition. By employing a set of powerful models, machine learning algorithms can extract intricate patterns from vast amounts of data. Obtaining a model with such exceptional computational capabilities requires a period to adjust the model's parameters and establish a representation of the input values from the provided dataset.

Non-distributed machine learning algorithm trains the machine learning model on a single machine. When using the non-distributed machine learning algorithm in the training of models, it has two main problems: the incompatibility of model size, dataset size, and memory and limited computing power on a single machine. When training the machine learning model on a single machine, it has to load the entire model and dataset into the machine's memory for processing. As machine learning models have become increasingly complex and the size of datasets has grown exponentially over the past few years, it has become infeasible to put everything on a single machine. As shown in Figure 1, even back 10 years ago, the training data size used for Ad click prediction already hit 700 TB.

The computation power of non-distributed machine learning is also limited by the capabilities of a single machine. As the size of the dataset and the complexity of the machine learning model increases, the training will be more time-consuming on a single machine. It is even possible to stop the development

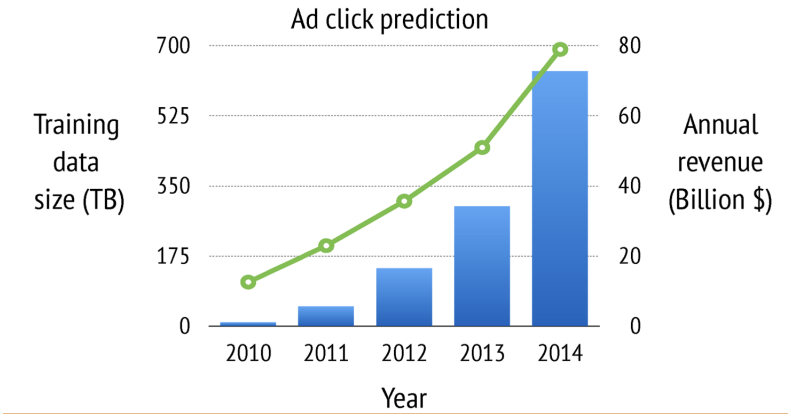


Figure 1: The training data size trend for Ad click prediction using machine learning

and deployment of advanced models on a single machine.

To address these challenges, distributed machine learning methods have been developed in the past few years. It divides the training of a machine learning model into smaller parts and assigns them to multiple machines. This approach not only mitigates the limitations of memory capacity and processing power but also has a significantly faster training time.

There are several distributed machine learning methods: Data Parallelism, Model Parallelism[1], Federated Learning[2], and Parameter Server[3]. Each distributed machine learning method has its own advantages and special requirements.

Data Parallelism is a method to divide the dataset into small parts and each machine is assigned a small portion of the data set. Each machine has a copy of the model and trains it independently with its assigned data subset.

Model Parallelism is a method to divide the model is separated into multiple parts and each machine has one part of the model. Each machine works on a portion of the model and shares the intermediate results with each other. In some cases, people will combine Data Parallelism and Model Parallelism together.

Federated Learning is a method that multiple machines train their local models using their own data, without sharing the data with a central server. Every machine periodically shares its updates with the central server, which aggregates these updates and sends the improved global model back to the clients.

In our work, we have specifically focused on the **Parameter Server** architecture. Our study involved implementing this architecture and comparing its performance with traditional non-distributed machine learning techniques. Additionally, we have taken into account fault tolerance considerations in our design.

2 Parameter Server Architecture

The parameter server algorithm has the following main components: central server, workers, and task scheduler. The central server aggregates updates of gradients from all the workers and applies the update to the model and gets a new set of parameters. It will pass the current set of parameters to workers.

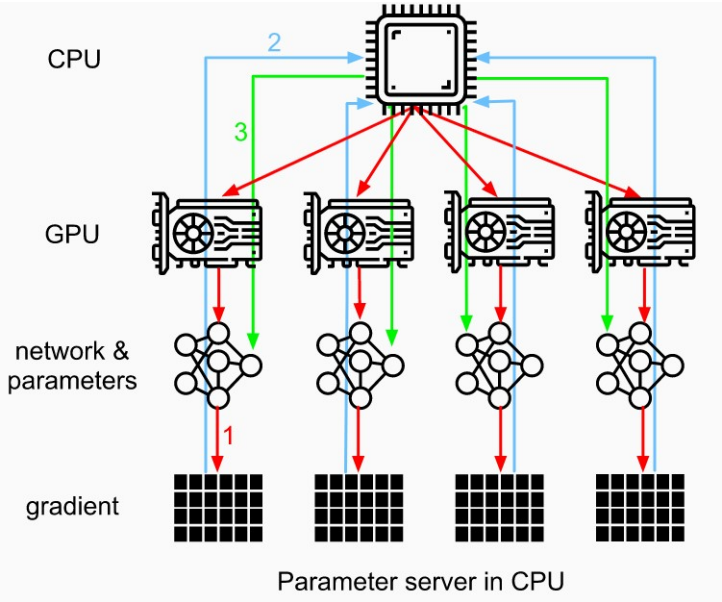


Figure 2: The Distributed Training Procedure

Workers are responsible for the assigned computing task for training the machine learning model. Each worker gets parameters from the server and sends the updated parameters back to the server. The task scheduler is responsible for coordinating the training task and ensuring workers are computing efficiently. It assigns tasks to workers to balance the workload across workers. It also handles the fault tolerance in this system. If any worker crashes, it will close it and rebalance the workload. The task scheduler will check the data flow between workers and the server and if workers have access to the most up-to-date model parameters.

Figure 2 illustrates how the distributed training occurs in our parameter server architecture. There are four main training steps:

1. Call task scheduler to assign servers and workers.
2. A batch of data is read on each machine, split across multiple GPUs, and transferred to GPU memory. Then predictions and gradients are computed on each GPU separately.
3. The gradients from all local GPUs are sent to the parameter server (CPU here).
4. Server will work on the gradients and update the model and save the model to the disk.
5. Server will then broadcast new parameters to each worker and repeat.

3 Algorithm and Implementation Details

We present the following algorithm for our parameter server in Algorithm 1 where $\nabla f_i(w_{t-1})$ represents the gradient of the loss function $f_i(w)$ with respect to the model parameters w, computed using worker i's local training data and the current working set of the model w_{t-1}. The learning rate η controls the step size of the gradient descent update. The algorithm runs for a fixed number of iterations, and the goal is to achieve the desired level of accuracy.

Algorithm 1 Distributed Machine Learning using Parameter Server Architecture under task scheduler

Require: N number of training workers, L number of training iterations

procedure DISTRIBUTEDMACHINELEARNING
  Initialize server model w_0
  Partition training data into training workers T_1, T_2, ..., T_N
  for $t = 1, 2, \ldots L$ **do**
    for each worker $i \in 1 \ldots N$ **do**
      Pull working set of model w_{t-1} from server
      Compute local gradient $g_{t,i} = \nabla f_i(w_{t-1})$ using worker i's working set and training data
      Push gradient $g_{t,i}$ to server
    end for
    Aggregate gradients on server: $g_t = \frac{1}{N} \sum_{i=1}^{N} g_{t,i}$
    Update server model: $w_t = w_{t-1} - \eta g_t$, where η is the learning rate
  end for
end procedure

For communication between the server and worker, we employed Ray's remote method. In the Ray distributed computing framework, remote functions are defined using the `@ray.remote` decorator, which tells Ray to execute the function in a separate process or on a separate machine. When a remote function is called, Ray serializes the function arguments, sends them to a worker node, and executes the function on that node. The result is then serialized and sent back to the calling process.

In our algorithm, The global model is pulled by workers at the beginning of one iteration. This stage is implemented using the `update_weights()` method defined on the worker's class. Workers will use its assigned local dataset and the pulled working set of the global model to compute the gradients. They then ship these gradients back to the server by calling the server's `add_weight()` method. The server is responsible for gathering the gradients, updating the model, and synchronizing all workers to enter the next iteration by broadcasting the updated model to them.

Data Partition

Data partitioning is an essential aspect of our parameter server implementation. It plays a crucial role in distributing the workload across multiple workers, facilitating parallel processing and scalability. Prior to our training, we divide the training dataset into shards using a predefined partitioning scheme. This scheme involves randomly sampling a sub-dataset for each worker. By doing so, we ensure an even distribution of data among the workers, mitigating the risk of any individual worker becoming a performance bottleneck. Furthermore, we guarantee that the assigned data partitions are stored locally by the respective workers, effectively reducing network communication overhead during each training iteration.

Parameter Partition

To better distribute the system, besides partitioning the training data, we can also partition the model parameters among servers using a hashing technique called consistent hashing[4]. The general idea is that all model parameters can be represented as key-value pairs, which are then hashed into a hash table. The servers utilize a partitioning strategy similar to that of a conventional distributed hash table. Both the keys and server node IDs are inserted into a hash ring, as illustrated in Figure 3. Each server is responsible for managing a specific key range, starting from its insertion point and extending up to

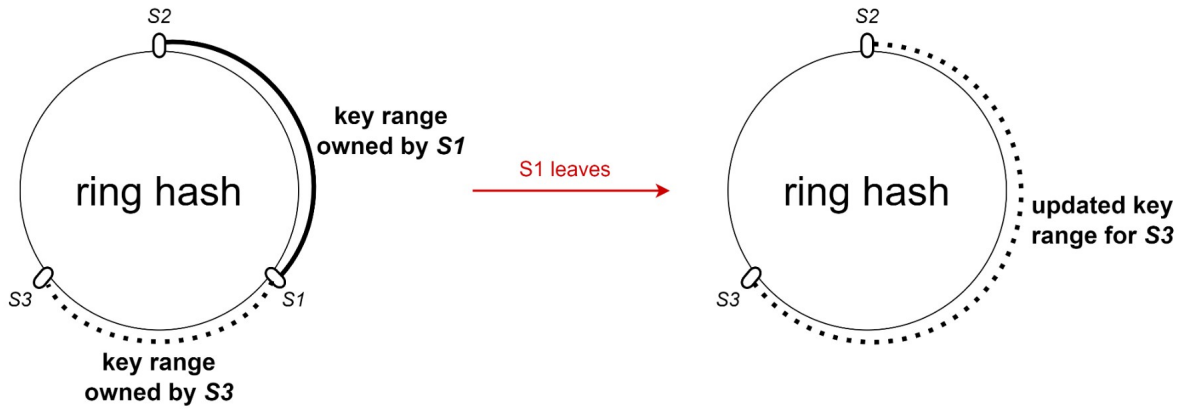


Figure 3: Server Layout and Key Range Mapping in Ring Hash

the next insertion point by another server in the counter-clockwise direction. One of the advantages of this hash ring-based approach is its flexibility in accommodating server additions or removals without requiring extensive rebalancing of key ranges among the remaining servers. When a server leaves the system, its workload is automatically taken over by a random server in the system. In our illustration in Figure 3, when server 1 leaves, the system randomly decides that server 3 will take over server 1's work. This random assignment ensures that the departure of a server does not disproportionately burden the other servers, maintaining a balanced distribution of key ranges among the active servers.

Worker Fault Tolerance

The fault tolerance for workers can be understood as follows: once a worker from a pool of workers is disconnected, the training will not be terminated, and other workers can take the load of the disconnected worker and carry on. In Ray, fortunately, we can utilize the `ActionPool` API, which allows us to create a pool of Worker actors and automatically respawn any dead workers. To initialize workers, instead of doing $[w1, w2, w3, ..., w_n]$, we call $ActorPool([w1, w2, w3, ..., w_n])$. And instead of calling `.remote()` function one by one on each actor, we call `.remote()` on a pool of actors to distribute the work.

Server Fault Tolerance

We have implemented server fault tolerance in our parameter server implementation, as illustrated in Figure 4. Instead of relying on a single server, we utilize multiple servers.

Each server node stores a replica of k counterclockwise neighbor key ranges relative to the one it owns. Let's consider a scenario where we have three servers in the system and $k = 2$. In this case, server 1 stores replicas of the key ranges belonging to both server 2 and server 3 as illustrated in 4. Similarly, server 2 and server 3 also hold replicas of the key ranges owned by the other two servers.

Worker and Server Persistence

One of the significant advantages of this approach is that it enables the system to retain its progress even if the training process is interrupted. Instead of starting the training from scratch, the system can resume training from the saved checkpoints. This feature enhances the persistence of the system, ensuring that the progress made during training is not lost in case of disruptions or failures. In machine learning

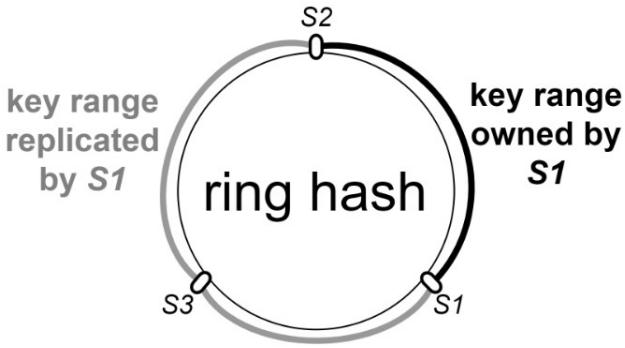


Figure 4: Server Fault Tolerance

training especially in PyTorch, doing this is relatively easy as all we have to do is to save the weights (checkpoints) of the trained models, and load it to the new model again on startup.

Unit Tests Implementation

We included unit tests for each component in our implementation as well. In order to run the unit tests, users might need to use PyPI to install all required Python packages and use 'python test.py' to run all unit tests.

4 Experiment

Dataset

We used MNIST[5] for our experiments. MNIST is a dataset of handwritten digits that is commonly used in the field of machine learning as a benchmark for image classification algorithms. The dataset consists of $60,000$ training images and $10,000$ testing images, each of which is a grayscale image of size 28×28 pixels representing a single digit from 0 to 9. MNIST is a suitable dataset for our parameter server experimental training because it is relatively small and easy to work with while still large enough to be experimented with in distributed machine learning system like our parameter server.

Model

To ensure that our parameter server architecture can be used for a wider range of machine learning tasks, we perform our experiments with a simple model for image classification and object detection.

Training Details

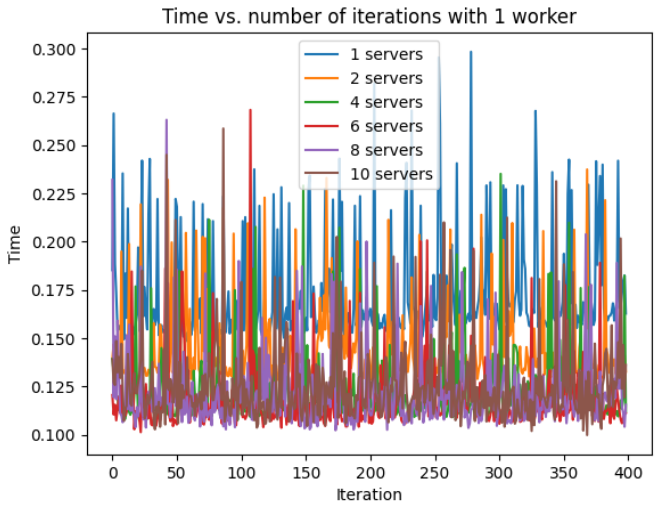
We maintained consistent settings across all experiments. Specifically, we employed a batch size of 128 and a learning rate of 0.1.

To control the aggregation and update process, we imposed a maximum step limit of 400 on each server. Additionally, we ensured that each worker trained its assigned subset of the model for 400 epochs. It is worth noting that our experiments were conducted on a local computer featuring an Intel Core i7-11800H Processor. Due to device limitations, GPU support was unavailable.

Evaluation

In our parameter server implementation, we assess its performance in terms of both training time and accuracy. To analyze the impact of the number of workers on runtime and accuracy, we employ sketching and present the results.

(a) Accuracy vs. number of iterations with 1 worker



(b) Time vs. number of iterations with 1 worker

(c) Accuracy vs. number of iterations with 1 server

(d) Time vs. number of iterations with 1 server

Figure 5: Experiment results

Results

We did two types of experiments as shown in Figure 5:

1. We have one fixed worker, and tuned the number of servers. We can observe that the training accuracy increases and training time decreases as the number of servers increases. The reason for the runtime decrease might be that with a larger number of servers, the workload can be distributed

across multiple nodes. Each server is responsible for processing a portion of the model parameters. This parallelism enables faster computation and therefore a decrease in training time. Moreover, the diminishing returns effect is observed for the reduction in running time with an increase in the number of servers. As more servers are added to the system, their impact on reducing the overall running time becomes less significant. Thus, we can conclude that our system has scalability limits, beyond which adding more servers does not result in proportional improvements. Besides, the increase in the training accuracy also indicates that having more servers can lead to improved convergence of the training process. With multiple servers working simultaneously, the model can benefit from diverse perspectives of the parameter space, which can help the model escape local optima and converge to better solutions, resulting in improved training accuracy.

2. We have one fixed server and tuned the number of workers. When examining the training accuracy, we observe that it consistently improves as the number of workers increases up to a threshold of less than 8 workers. Once the threshold is surpassed, it is important to note that even though the top accuracy achieved remains higher than when using fewer workers, there are notable fluctuations compared to the case of a smaller number of workers. One potential reason for this behavior could be that when we have 8 or more workers, they demand a greater amount of CPU resources than our current device can provide since each worker requires CPU resources to perform computations, handle data transfers, and communicate with other workers. Another reason is that the cost of communication plays a more important role with respect to the training time, and the stability of the communication between workers influences the result. When considering the training time, we find that it tends to increase on average as the number of workers increases. This can be attributed to two main factors: communication overhead among workers and the potential for load-balancing issues.

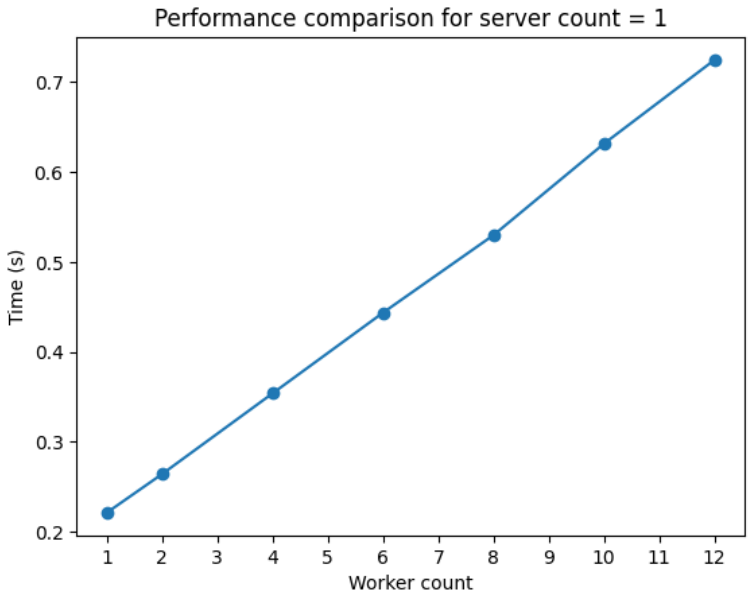


Figure 6: Time usage comparison with server number fixed

,

Besides, according to Figure 6, we found that the time used increases almost proportionally to the worker count. This result might be related to the fact that communication time usage is greater than training

time usage in this specific implementation. When facing larger models like Transformer-based models (GPT-3[6], ViT[7]), it might yield a different trend.

5 Future Work

We can further overlap the computation (forward and backward propagation) with communication to gain better speedup. Currently, the communication methods employed are synchronous ray functions to perform tasks such as sending, gathering, and broadcasting, which stops the worker from doing meaningful local computations while waiting for parameter updates from the server. We can change to asynchronous functions and utilize the worker's idle time without waiting for the communication to complete, which can then further maximize the utilization of resources.

6 Summary and Discussion

In this paper, we introduced a parameter server framework that can be used to solve distributed machine learning problems. We also added fault tolerance to the framework to ensure that the system can continue to work even if workers or servers are terminated during the training process, as long as there is at least one worker and one server available. To demonstrate the efficiency of our framework, we conducted experiments on a simple deep learning model. The results show that our framework can effectively handle distributed machine learning problems with high accuracy and low latency. Admittedly, the model used in our project is relatively simple compared to real-world machine learning models. Notably, there are currently no dependencies between model weights, which are quite common for deeper neural networks. Despite the limitation, we have already shown in our experiment that our framework could be easily extended to other machine learning tasks. Overall, our framework provides a practical solution for large-scale distributed machine learning problems that require fault tolerance and efficient training and scalability.

References

[1] Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Mark Mao, Andrew Senior, Paul Tucker, Ke Yang, and Andrew Ng. Large scale distributed deep networks. In *Advances in neural information processing systems*, pages 1223–1231, 2012.

[2] H. Brendan McMahan, Eider Moore, Daniel Ramage, Seth Hampson, and Blaise Aguera y Arcas. Communication-efficient learning of deep networks from decentralized data. *arXiv preprint arXiv: 1602.05629*, 2017.

[3] Mu Li, David G Andersen, Jun Woo Park, Alexander J Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene Shekita, Baoping Su, James Wilson, et al. Scaling distributed machine learning with the parameter server. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 583–598, 2014.

[4] I. Stoica, R. Morris, D. Liben-Nowell, D.R. Karger, M.F. Kaashoek, F. Dabek, and H. Balakrishnan. Chord: a scalable peer-to-peer lookup protocol for internet applications. *IEEE/ACM Transactions on Networking*, 11(1):17–32, 2003.

[5] Li Deng. The mnist database of handwritten digit images for machine learning research. *IEEE Signal Processing Magazine*, 29(6):141–142, 2012.

[6] Tom B Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *arXiv preprint arXiv:2005.14165*, 2020.

[7] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, et al. An image is worth 16x16 words: Transformers for image recognition at scale. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 3156–3164, 2021.