Implementing and Benchmarking a Fault-Tolerant Parameter Server for Distributed Machine Learning Applications

CS244B Final Project

Anusri Pampari
anusri@stanford.edu

Aman Patel
patelas@stanford.edu

June 3, 2022

Introduction

Recent state-of-the-art machine learning algorithms have revolutionized our ability to model and extract insights from data in a wide variety of disparate fields, from language to vision to biology. One of the strongest trends in current AI research involves a shift towards larger models with billions of variables ([1], [2]) trained on petabyte-scale datasets ([8], [6]), thus necessitating the need for distributed training systems. One such system, called the Parameter Server (PS), has seen widespread adoption in both academia and industry ([5], [3]) and has undergone multiple generations of development customized towards various machine learning and deep learning applications. The first generation of these systems, a "data parallel" setting, involves a single central server and multiple worker nodes. In this setting, training data is shared among workers, while the central server stores a global set of model weights, which each worker draws upon. Specifically, at each iteration, all workers fetch the most recent weights from the server, compute gradients with a local batch of data, and push the gradients back to the server so the global weights can be updated. This system can iterate through petabyte-scale training data far more quickly than is possible with a single-core training system.

However, globally sharing all model weights can impose several challenges. First, requiring a single server to access all parameters and update all gradients per iteration necessitates a network bandwidth proportional to the number of model parameters. Second, as only one server is utilized, the process of fetching gradients and updating weights is slow and can be parallelized. Hence, a second generation of these systems have focused on "model parallel" settings, in which model weights are distributed across multiple parameter servers. This reduces the network bandwidth and computational load per server, thus improving overall speed.

In this study, we provide and explore an implementation of a model-parallel distributed training system in Python, using the Ray library([7]). It is important to note that parameter server implementations are often tailored to the machine learning task at hand, so we focus specifically on a linear or logistic regression setting. First, we will demonstrate the efficacy of our implementation of consistent hashing, which we use to distribute weights across servers [4]. Next, we will use a logistic regression model and the MNIST dataset to benchmark a variety of system properties, including runtime and accuracy, in addition to comparing our system to a traditional data-parallel setting. Finally, we add fault tolerance to our system by ensuring training can continue when a server goes down, and we explore the effect this extension conveys on performance.

1 System and Implementation

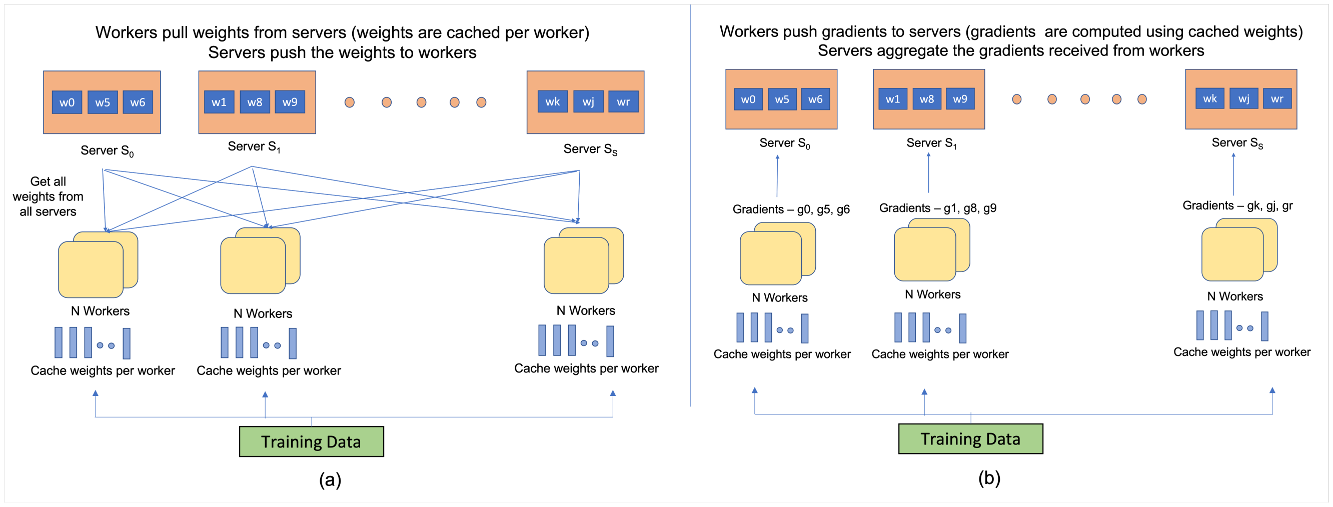

Workers pull weights from servers (weights are cached per worker)
Servers push the weights to workers
w0 w5 w6
w1 w8 w9
wk wj wr
Server S0
Server S1
Server SS
Get all weights from all servers
N Workers
Cache weights per worker
Training Data
(a)
Workers push gradients to servers (gradients are computed using cached weights)
Servers aggregate the gradients received from workers
Gradients – g0, g5, g6
Gradients – g1, g8, g9
Gradients – gk, gj, gr
(b)

Figure 1: A schematic of the distributed training system. (a) At pre-specified iterations, each worker pulls and stores the latest weights from all servers. (b) At every iteration, workers compute gradients based on local training data and push them to the relevant server. The servers then use these gradients to perform weight updates.

Algorithm 1 Synchronous Parameter Server for Linear/Logistic Regression

```
Input: Server IDs $\{S_0, S_1 ... S_S\}$, Number of workers per server $N$, Number of weights $W$, Training iterations $I$, Checkpoint iteration interval $C$
  Randomly initialize $W$ weights $\{W_0, W_1 ... W_W\}$
  Use consistency hashing to map $\{W_0, W_1 ... W_W\}$ to $\{S_0, S_1 ... S_S\}$ and get $D_{ws}$ dictionary
  Distribute $\{W_0, W_1 ... W_W\}$ to $\{S_0, S_1 ... S_S\}$ based on $D_{ws}$
  while t $< I$ do
      if $t\%C == 0$ then
          Workers pull $\{W_0^t, W_1^t ... W_W^t\}$ from all servers based on $D_{ws}$
          $\{W_0^t, W_1^t ... W_W^t\}$ are saved as a checkpoint
      end if
      Workers compute and push gradients $\{G_0^t, G_1^t ... G_W^t\}$ to respective servers based on $D_{ws}$
      Servers use gradients $\{G_0^t, G_1^t ... G_W^t\}$ to update $\{W_0^t, W_1^t ... W_W^t\}$ based on $D_{ws}$.
  end while
```

We implement a synchronous parameter server (also described in Algorithm 1) for linear or logistic regression with the following system components:

Servers: We can have either a single central server that stores a global set of model weights (total size $\|W\|$), or multiple servers with the $\|W\|$ weights distributed across them using a weights-to-server mapping D_{ws} obtained from consistent hashing. See Figure 1 and Figure 2a for more details. As will be described below, servers have two main functions: to push weights to workers, and to pull and aggregate gradients from them.

Workers: Every server S_i is associated with N workers, where the workers jointly learn the set of weights W_{si} that S_i stores. Each worker stores its own local copy of the model, with all weights frozen except those in W_{si}. In each iteration, every worker independently samples a batch of the training data and uses its local copy of the weights to determine what changes should be made to W_{si}, assuming the other weights in the model are constant, to minimize the model's loss on the data. These changes (or gradient updates) are then pushed to S_i and aggregated to obtain a new set of master weight values for W_{si}. This kind of gradient update, called coordinate gradient descent, is theoretically well studied in machine learning [10]. At the start of training and after every C iterations, all workers' local models are updated with the latest weights from each server, thus synchronizing training across the system.

In the case of a single server setting, the most expensive step in Algorithm 1 involves communicating and updating the subgradients G^t, both of which scale linearly with the number of parameters. By increasing the number of servers, and distributing the weights evenly across these servers, we can speed up this step of the algorithm and therefore improve the efficiency of the system. In Section 1.1, we show how we can distribute weights evenly and achieve load-balancing by using consistent hashing. In Section 1.2, we use this mapping within our parameter server, and we show improvements in runtime between multiple-server and single-server settings. We acknowledge that when weights are distributed across multiple servers, failure of one server can block the progress of the system. Therefore, in Section 1.3, we introduce a fault tolerance algorithm to account for server failure by re-assigning the weights of the failed server using consistent hashing.

1.1 Consistency Hashing Performance

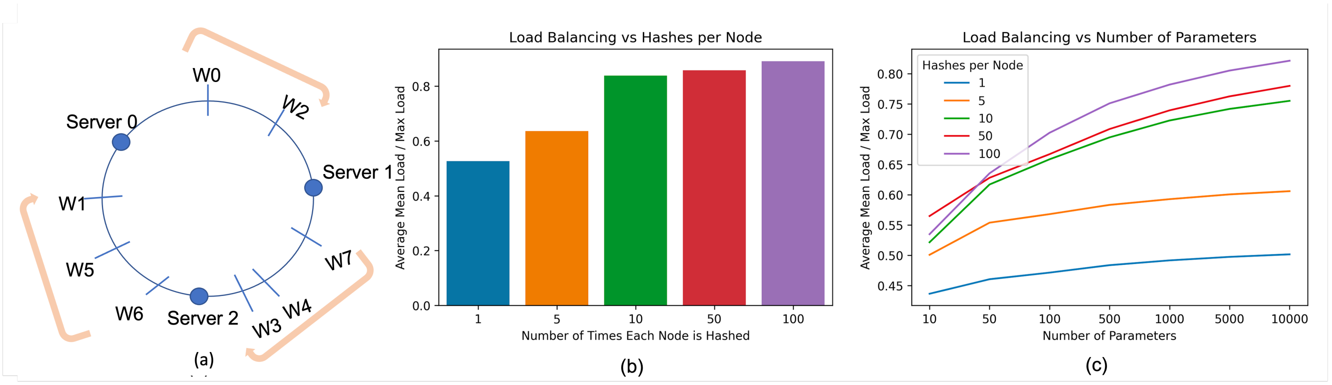


Figure 2: Characterization of our consistent hashing implementation. a) A schematic of the consistent hashing process. b) Relationship between load balancing ability and the number of times each node is hashed. c) Relationship between load balancing activity and the number of keys (parameters) in the hash ring.

Before applying our consistent hashing implementation to model training, we verified its behavior and efficacy in load balancing. Suboptimal performance in this regard could lead to a lopsided assignment of weights to servers, potentially greatly hampering training efficacy.
To set up the problem, assume we have n nodes and k keys, where $k >> n$, and we want to assign each key to a node. In our distributed. system, nodes would correspond to servers and keys to model parameters. As shown in Figure 2a, we achieve this by hashing each node and key to a position (from $0 - 2\pi$ radians) on a circle. Finally, we assign each key to the nearest node with position greater than the key's position. In essence, each node "covers" the area between its position and the previous node's position. In this manner, each node has an expected value of k/n nodes assigned to it.
However, the distribution of keys per node may be rather lopsided in practice. For example, if two nodes hash nearby to each other, then one of them will cover a very small portion of the circle. One solution is to hash each node multiple times to different points on the circle. In this manner, we eliminate the randomness of hashing each node only once and likely produce a more even distribution between nodes.
To ensure this hypothesis holds for our implementation, we tested a consistent hashing system with 5 nodes and 785 keys while varying the number of times each node was hashed. Figure 2b indicates how the load balancing ability (averaged over 20 runs with different sets of keys) varies with the number of

hashes per node.
The results clearly indicate that in our implementation, a large number of hashes per node is necessary to obtain effective load balancing. This is expected behavior, and we have used it to inform the settings of our training system.
In a similar manner to the previous concern, a sufficiently small number of keys could also skew the load balancing. We therefore conducted similar experiments, except this time varying the number of keys in our hash ring. The results are shown in Figure 2c. As expected, an increase in the number of keys also increases the load balancing efficiency. It is informative to note that the number of parameters in our test model (785) would place us at an efficiency of approximately 0.8 with 100 hashes per node.
Overall, these experiments allow us to profile the behavior of our consistent hashing implementation and to demonstrate its proper functionality. They also result in more informed parameter choices for our distributed training system.

1.2 Benchmarking System Performance

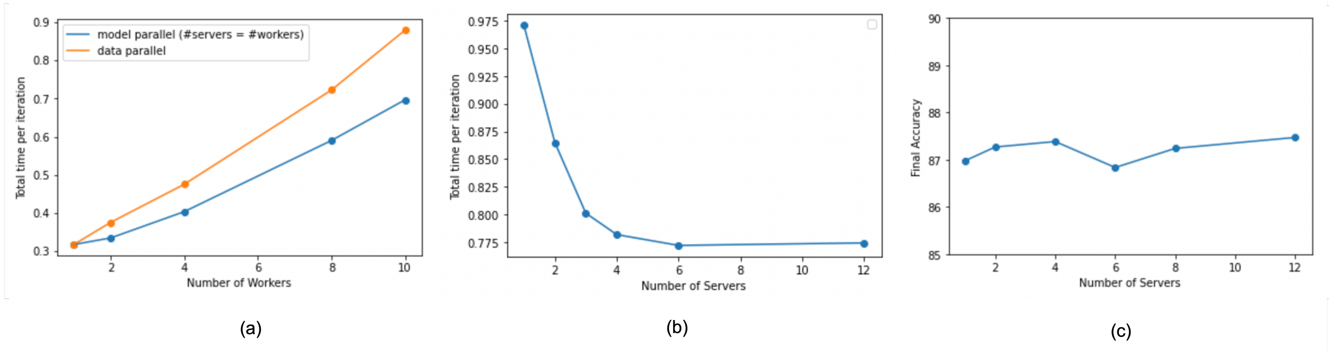


Figure 3: Performance of model-parallel parameter server (a) Model-parallel is more efficient than data-parallel at various worker counts (b) Increasing the number of servers improves efficiency (c) Increasing the number of servers does not produce an appreciable decrease in overall accuracy

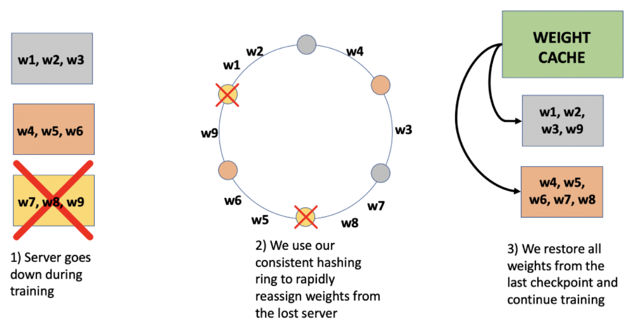
In this section, we carefully characterize the performance of using multiple servers with distributed weights (model parallel), and we compare the efficiency of this system to that of using a single server (data parallel). To achieve a relatively equal distribution of weights in the multiple server setting, we use the consistency hashing technique with 100 hashes per server as described in section 1.1. For benchmarking, we use the MNIST dataset, which consists of images of digits from 0-9, and we train a logistic regression model designed to classify these images as even or odd. The model has a total of 785 parameters. All the experiments in this section are averaged for 3 different seed runs. For simplicity, we checkpoint weights at every iteration.

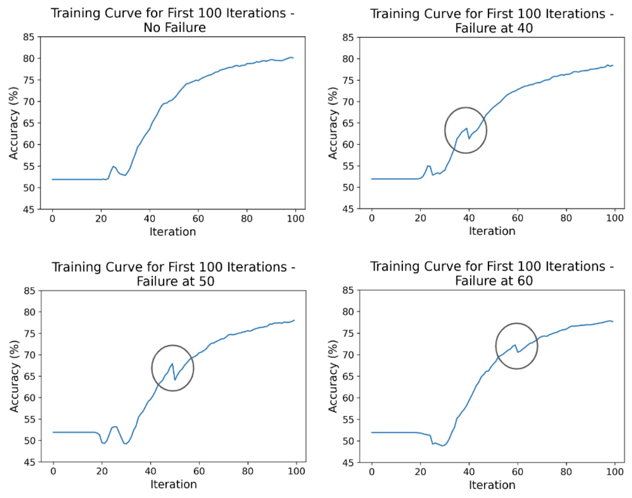
First, we compare the efficiencies of the multiple and single-server settings while varying the total number of workers in the system. Specifically, if we have N total workers, then the multiple-server setting will have N servers, each with one worker per server, while the single-server setting will of course have all N workers coordinating with the central server. In Figure 3a, we show that the total time per iteration is consistently more efficient with multiple servers as compared to one server and that the efficiency gap widens as the number of workers increases. This result matches expectations, because with an increasing number of workers in the data-parallel setting, a single server is now responsible for collecting and aggregating all gradients from all workers. This serializes both the communication of workers with the servers (with multiple workers accessing the same server) and the weight update step within the server. With multiple servers, this bottleneck is overcome because every worker is now responsible for pushing only a subset of the total gradients to its constituent master server. Furthermore, the individual servers are responsible for gradient updates of their own weights. Thus, these processes can run more efficiently in parallel.

Having established the superior efficiency of the multiple-server setting, we now further characterize it. First, we focus on the effect of server count on runtime. Specifically, given a constant total of 12 workers in the system, we vary the number of servers and track how the time per iteration changes as a result. Note that in this case, more servers would mean fewer workers per server. We run these experiments for 500 iterations. In Figure 3b, we show that as the number of servers increases, the time per iteration decreases and then eventually plateaus. Once again, this result reflects expected behavior - as we increase the number of servers, we reduce the runtime of the gradient computation step until it no longer becomes the major bottleneck.

Next, we probe the effect of server count on model accuracy. To ensure uniform training convergence, we keep the number of workers per server (one) as a uniform value. As shown in figure 3c, varying the server count does not produce an appreciable decrease in accuracy, which is a reassuring result for the reliability of modules trained in a distributed manner. However, we acknowledge the task-dependent nature of these results, which is a known drawback of parallel coordinate descent-based systems. In the worst case, if features (and therefore weights and gradients) are highly correlated, randomly separating them across servers may lead to slow convergence or even divergence. Some previous work combats this issue by pre-computing feature correlations and grouping them accordingly across servers [9]. We suggest this method as a future extension of our work.

1.3 Fault Tolerance


w1, w2, w3
w4, w5, w6
w7, w8, w9
1) Server goes down during training
w1
w2
w4
w3
w9
w6
w5
w8
w7
2) We use our consistent hashing ring to rapidly reassign weights from the lost server
WEIGHT CACHE
w1, w2, w3, w9
w4, w5, w6, w7, w8
3) We restore all weights from the last checkpoint and continue training



(a)

(b)

Figure 4: A procedure to allow robustness to server shutdowns. a) A schematic of our approach. b) Training curves with server failure at 40, 50, and 60 iterations compared with no failure

An important property of distributed systems in practice is fault tolerance - if one of the many components of the system fails, the system should be able to rebound and continue execution. In this section, we introduce a fault tolerant extension to our training system. Specifically, we focus on robustness to server failures, which otherwise would crash the entire training process.

The algorithm is described in Figure 4a. When a server goes down, we utilize our consistent hashing ring to locate the relevant weights and reassign them to new servers. As the most recent universally available version of these weights are those stored at the last checkpoint, we restore all weights from this checkpoint for uniformity, and we resume training.

To ensure the efficacy of this algorithm, we simulated server shutdowns at known iterations in the training process, and we observed training behavior as a result. The results are displayed in Figure 4b, in which we compared simulated failures at iterations 40, 50, and 60 to the training curve without any failures. In each case, performance drops slightly at the failure iteration, which is expected since weights were restored from the last checkpoint, but training rebounds quickly and continues as normal. All plots were the result of averaging over three runs of training with five servers, one worker per server, and checkpointing weights every five iterations. Through our algorithm, a server shutdown did not prove catastrophic, but rather only caused a momentary decrease in performance. Thus, we conclude this procedure can indeed provide robustness to server failures in distributed training systems similar to ours.

2 Discussion

In this study, we present a parameter server system for distributed training of machine learning models. First, we describe the details of our algorithm and approach, and we characterize the consistent hashing framework essential to the procedure. Next, we benchmark our system against a data parallel setting and show improvements in total runtime. We also show that increasing the system's server count does not cause a drop in accuracy, and, with a constant total worker count, causes a decrease in runtime. Finally, we describe a fault tolerant extension to our system, which allows for training to recover quickly after a server crashes.

Of course, several opportunities for extensions and improvements exist. Our model is quite simple from a machine learning standpoint, and most notably, there are no dependencies between model weights, as would be the case in deeper neural networks. Accounting for these dependencies would require more complex algorithms than those presented here, but for parameter servers to form an essential part of the modern machine learning toolkit, this functionality is essential. Despite this limitation, our framework can be easily extended to a variety of other machine learning applications, including topic modeling approaches like Latent Dirichlet Analysis (LDA)[5]. Furthermore, our system is currently synchronous - that is, all workers are always at the same point in training, and model weights are globally synchronized at set intervals. If we switch to an asynchronous setting, thus allowing workers to operate independently without global control and locking, it is possible that our system will operate more efficiently. Thus, characterizing the tradeoffs between synchronous and asynchronous settings represents another key area of future work. Finally, introducing other forms of fault tolerance, particularly against worker shutdown, would make our system more complete.

In conclusion, we demonstrate the utility of our parameter server system for distributed training, and we present several avenues of future exploration.

3 Reproducibility

Code for this project can be found at https://github.com/amanpatel101/CS244BParameterServer. All the notebooks and data to reproduce the plots are also provided with the github repo.

References

[1] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever,

and Dario Amodei. Language models are few-shot learners. *CoRR*, abs/2005.14165, 2020. URL `https://arxiv.org/abs/2005.14165`.

[2] Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, Parker Schuh, Kensen Shi, Sasha Tsvyashchenko, Joshua Maynez, Abhishek Rao, Parker Barnes, Yi Tay, Noam Shazeer, Vinodkumar Prabhakaran, Emily Reif, Nan Du, Ben Hutchinson, Reiner Pope, James Bradbury, Jacob Austin, Michael Isard, Guy Gur-Ari, Pengcheng Yin, Toju Duke, Anselm Levskaya, Sanjay Ghemawat, Sunipa Dev, Henryk Michalewski, Xavier Garcia, Vedant Misra, Kevin Robinson, Liam Fedus, Denny Zhou, Daphne Ippolito, David Luan, Hyeontaek Lim, Barret Zoph, Alexander Spiridonov, Ryan Sepassi, David Dohan, Shivani Agrawal, Mark Omernick, Andrew M. Dai, Thanumalayan Sankaranarayana Pillai, Marie Pellat, Aitor Lewkowycz, Erica Moreira, Rewon Child, Oleksandr Polozov, Katherine Lee, Zongwei Zhou, Xuezhi Wang, Brennan Saeta, Mark Diaz, Orhan Firat, Michele Catasta, Jason Wei, Kathy Meier-Hellstern, Douglas Eck, Jeff Dean, Slav Petrov, and Noah Fiedel. Palm: Scaling language modeling with pathways, 2022. URL `https://arxiv.org/abs/2204.02311`.

[3] Rong Gu, Shiqing Fan, Qiu Hu, Chunfeng Yuan, and Yihua Huang. Parallelizing machine learning optimization algorithms on distributed data-parallel platforms with parameter server. *2018 IEEE 24th International Conference on Parallel and Distributed Systems (ICPADS)*, 2018. doi: 10.1109/padsw.2018.8644533.

[4] David Karger, Eric Lehman, Tom Leighton, Rina Panigrahy, Matthew Levine, and Daniel Lewin. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing - STOC '97*, 1997. doi: 10.1145/258533.258660.

[5] Mu Li. Scaling distributed machine learning with the parameter server. *Proceedings of the 2014 International Conference on Big Data Science and Computing - BigDataScience '14*, 2014. doi: 10.1145/2640087.2644155.

[6] Chao Liu, Fan Guo, and Christos Faloutsos. Bbm: bayesian browsing model from petabyte-scale data. *Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining - KDD '09*, 2009. doi: 10.1145/1557019.1557081.

[7] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I. Jordan, and Ion Stoica. Ray: A distributed framework for emerging ai applications, 2018.

[8] Sree Hari Krishnan Parthasarathi, Nitin Sivakrishnan, Pranav Ladkat, and Nikko Strom. Realizing petabyte scale acoustic modeling. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, 9(2):422–432, jun 2019. doi: 10.1109/jetcas.2019.2912353. URL `https://doi.org/10.1109%2Fjetcas.2019.2912353`.

[9] Chad Scherrer, Ambuj Tewari, Mahantesh Halappanavar, and David Haglin. Feature clustering for accelerating parallel coordinate descent, 2012.

[10] Stephen J. Wright. Coordinate descent algorithms. *Mathematical Programming*, 151(1):3–34, 2015. doi: 10.1007/s10107-015-0892-3.