COMP 4801 FINAL YEAR PROJECT

Efficiency and Robustness in Federated Learning
Interim Report

Supervisor	Professor C. L. Wang
Name	Yue Yin, BEng(CS), 3035448512
	Ruijie Li, BEng(CE), 3035446992

Contents

1 Introduction

Carrying powerful sensors and being used at high frequency, digital devices like phones and tablets are becoming the primary computing devices for most people nowadays. Such devices have access to an unprecedented amount of private user data, which can be very valuable to machine learning applications. However, because of the privacy concern and high cost of network communication, traditional distributed machine learning scheme becomes infeasible, as it requires uploading user data to some centralized location like a data-center [1].

To utilize such sensitive data for machine learning without compromising user privacy, federated learning (FL) [1] has been proposed as a solution that trains models without direct access to data. Federated learning brings about new opportunities as well as new challenges, in terms of the efficiency and robustness of the scheme.

2 Background

2.1 Federated Learning

2.1.1 Motivations

End devices with powerful sensors have access to huge amount of user data valuable for machine learning applications. However, such data are usually privacy sensitive and should not be uploaded to the data center for model training. Federated learning aims to utilize such data for model fitting without compromising user privacy, by decoupling model training with data access.

2.1.2 Working mechanisms

The architecture of a federated learning system is similar to the parameter-server model [2], originally proposed for the distributed machine learning problem, where a server orchestrates clients. One round of training in federated learning involves four steps:

1. Server selects a set of clients. Selected clients download the model from the server.
2. Clients compute updates to the model using local data.
3. Clients upload their own updates to the server.
4. Server aggregates clients' updates into an update to the global model.

This completes one synchronization cycle between the client and the server. In this paper, we call this synchronization cycle the *clock*.

The overall goal of federated learning is to solve the optimization problem

$$\min_{\omega} F_i(\omega_i) = \frac{1}{n_i} \sum_{i=1}^{n_i} f(x_j^i; \omega_i),$$

where x_j^i is the j-th data point in the i-th worker node. At clock t, ω_t can be calculated by local gradient descent:

$$\omega_t := \omega_{t-1} - \eta_{t-1} \nabla F(\omega_{t-1}).$$

where η_{t-1} is the learning rate.

2.1.3 Differences from distributed machine learning

A key difference between federated learning and distributed machine learning is that user data remains on the client devices, without being uploaded to the data-center. The FL server aggregates clients' updates to improve the global model.

Several other properties distinguish federated learning from distributed optimizations:

- **Non-IID** In distributed machine learning, the global dataset is partitioned and shuffled to be independent and identically distributed (IID). For federated learning, each training dataset is collected from the usage of each particular device and any particular user's data might not be representative of the global dataset [1,3], so the data is neither independent nor identically distributed (Non-IID).
- **Unbalanced Distribution** In distributed machine learning, the global dataset is partitioned into uniformly-sized blocks. In FL, the amount of data on each end device varies significantly [1].
- **Resource Heterogeneity** In distributed machine learning, machines are usually in similar conditions. For FL, there is significant variance in the system characteristics of end devices in computational power, training participation rate, etc [3]. It is typical to have stragglers (slower clients) in the FL setting.
- **Limited Communication** In distributed machine learning, machines are normally connected with high-bandwidth, low-latency cable networks. For FL, end devices are normally connected by wireless networks, so communication can be unstable, slow, and expensive [1].

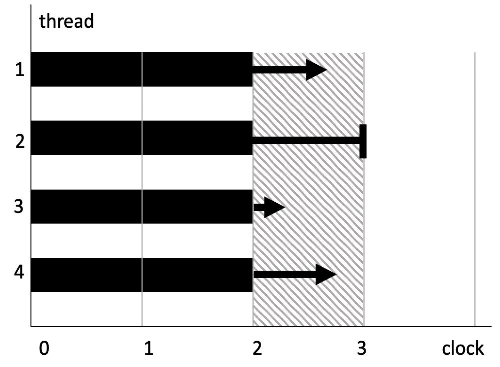
2.2 Synchronization schemes

Three major synchronization schemes have been proposed in the field of distributed machine learning. Existing federated learning systems are usually also covered by these schemes:

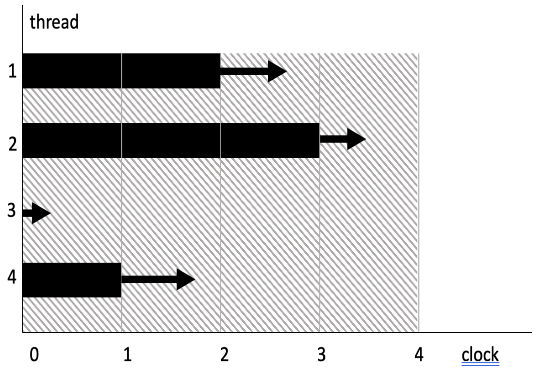
BSP: Bulk Synchronous Parallel. Systems like FedAvg [1] and FexProx [3] synchronizes clents under the BSP protocol, where one client cannot continue to the next clock until the PS receives updates from all clients in the same round. In the presence of stragglers, the overall training time is determined by the slowest client.

ASP: Asynchronous Parallel. Systems like [4,5] use an ASP protocol for synchronization. The protocol imposes no constraints on clients' clocks and clients can proceed without any waiting. ASP systems are often faster than BSP systems [4], due to the looser synchronization constraints. However, with significant stragglers, ASP cannot guarantee convergence [5].

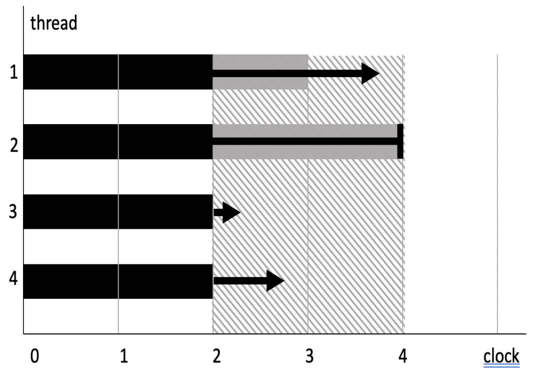
SSP: Stale Synchronous Parallel. Systems like DSSP [6] and FlexPS [7] adopt the SSP strategy [8], which bounds the difference between the fastest client's clock and the slowest client's clock with a *staleness bound*, denoted by s. In other words, the fastest worker can exceed the slowest one by at most the staleness bound.



(a) BSP



(b) ASP



(c) SSP

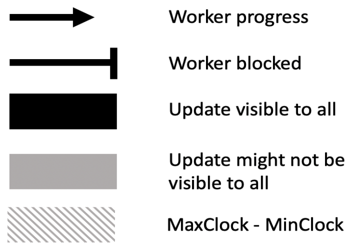


Figure 1: BSP, ASP, SSP

SSP can be viewed as the general model: when $s = 0$, SSP becomes BSP; when $s = \infty$, SSP becomes ASP. The three schemes are visualized in 1. In this project, we use SSP as the synchronization protocol for federated learning.

To differentiate federated learning schemes from distributed machine learning schemes adopting similar protocols, we use the notation BSP-FL, SSP-FL, and ASP-FL in our discussion.

3 Objectives

In federated learning, the presence of stragglers makes highly synchronous schemes (like BSP-FL) inefficient, because the overall training time is determined by the slowest worker [9]. On the other hand, the nature of non-IID datasets causes the difference between the the local optimums and the global optimum, making insufficiently synchronous schemes (like ASP-FL) suffering the problem of incorrect convergence [5].

As an intermediate solution between BSP and ASP, SSP is faster than BSP while achieving an accuracy higher than ASP [6]. However, SSP-FL has no guarantee in either its **efficiency in training time** or its **robustness against incorrect convergence**, both of which depending on the value of the staleness bound, s. In SSP, the value of s is fixed throughout the training process, and it is usually hard for the user to specify a single value for s. Finding a suitable s typically involve manually searching in a range of integer values through trails [6].

In short, by combining BSP and ASP, SSP can potentially avoid ASP's disadvantage of being prune to incorrect convergence and BSP's inefficiency due to stragglers. However, considering the difficulty of finding a good value of s for SSP-FL, the problem of efficiency and robustness remains to be solved.

Efficiency and Robustness. The objective of this project is to develop a federated learning synchronization scheme that achieves efficiency (short training time) and robustness (against incorrect convergence) at the same time.

4 Methodology

4.1 Staleness Bound

SSP combines ASP and BSP through staleness bound s, the parameter that bounds the gap between the fastest client(s) and slowest worker(s) [8]. In distributed machine learning, the parameter is fundamental for the performance of SSP, as it affects both the training time and the model accuracy. As s increases, the total training time decreases and the model converges more quickly [10], but the accuracy drops [11].

In federated learning, the impact of staleness bound on model accuracy and training time have not been thoroughly examined yet. We expect similar relationships between s, accuracy and training time. Our preliminary evaluation result in 5.2.4 is consistent with our expectation.

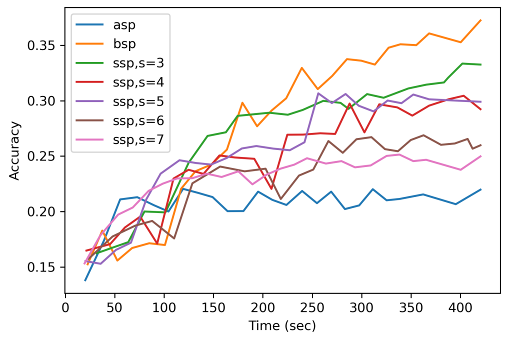
Based on results from distributed machine learning and our preliminary result of SSP-FL in 5.2.4, we expect the following relationship between s, model accuracy and training time:

When s increases, the training time decreases, the model converges faster, but the accuracy ceiling is lower.

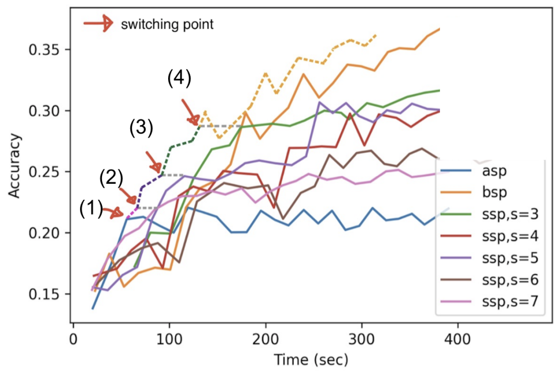
We plan to verify this expectation more thoroughly through experiment. The testing plan is detailed in 6.1.

4.2 Adaptive strategy

If the expectation is true, the relationship can be visualized as in 2a (this is actually obtained from our preliminary experiment result). The observation is that SSP-FL with smaller s reach lower accuracy quickly, while SSP-FL with larger s reach higher accuracy, but taking longer to converge.



(a) SSP, different s



(b) Adaptive s

In an adaptive strategy, SSP-FL can vary s throughout the training process. Intuitively, at the beginning of the training process, s should be large to quickly converge. Once the current value of s reaches its accuracy ceiling, the strategy should decrease s, allowing SSP-FL to converge to a higher accuracy, at the cost of longer training time.

This idea is illustrated in 2b, where time points to reduce s are marked (the "switching point" in the figure). Ideally, if the strategy can detect the switching points, the performance of the adaptive strategy (the dashed line) outperforms any strategy with a fixed s (including BSP-FL and ASP-FL).

5 Current Progress

5.1 FL Framework

Building an efficient and robust federated learning system is one of the top priorities of current stage. Based on Flower, a FL framework which is unaware of heterogeneous client conditions and can scale to a large number of clients [12], we built a customized FL framework, suitable for implementing and evaluating our strategies.

5.1.1 Adoption and Refinement of Flower

Flower is an open-source framework with well-written documentations. The focus of Flower's design is to help make the transition from existing machine learning to federated learning simple, so Flower is largely based on common machine learning libraries such as PyTorch and Tensorflow [13]. Such implementation grants a clear structure and room for customization, especially for the strategy of synchronization.

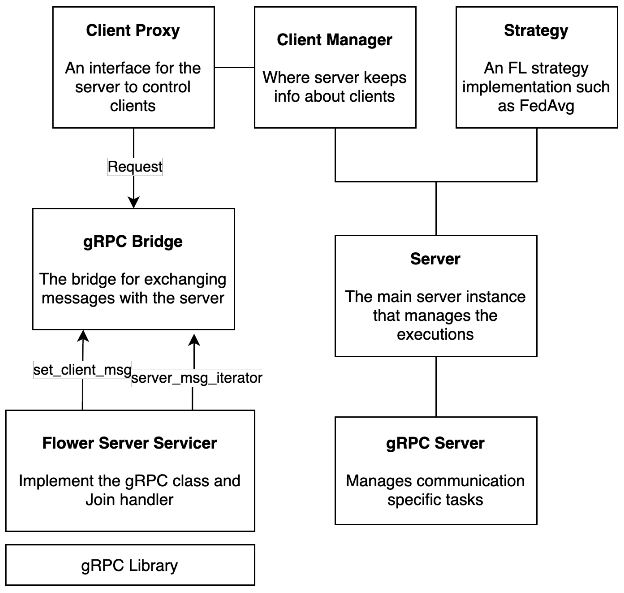

Client Proxy
An interface for the server to control clients
Client Manager
Where server keeps info about clients
Strategy
An FL strategy implementation such as FedAvg
Request
gRPC Bridge
The bridge for exchanging messages with the server
Server
The main server instance that manages the executions
set_client_msg
server_msg_iterator
Flower Server Servicer
Implement the gRPC class and Join handler
gRPC Server
Manages communication specific tasks
gRPC Library

Figure 3: The structure of Flower Framework

Flower is originally designed to execute BSP-FL (like FedAvg [1]). Flower server uses `FitIns`, a Fit Instruction to tell the clients to start training and also encloses the global model held by the server. Flower clients use `FitRes`, a Fit Result to contain the client's trained model based on its local dataset. In each round, Flower distributes `FitIns` to each registered client, and wait for all of their `FitRes` before proceeding to the next round of training.

However, as argued previously, BSP-FL systems like FedAvg [1] are inefficient in the presence of stragglers. As a result, we adopt and implement SSP-FL to address the problem. Instead of using a single thread to collect all `FitRes` at server, we create one thread per client to handle the message exchange of that specific client, enabling the FL server to serve each client concurrently.

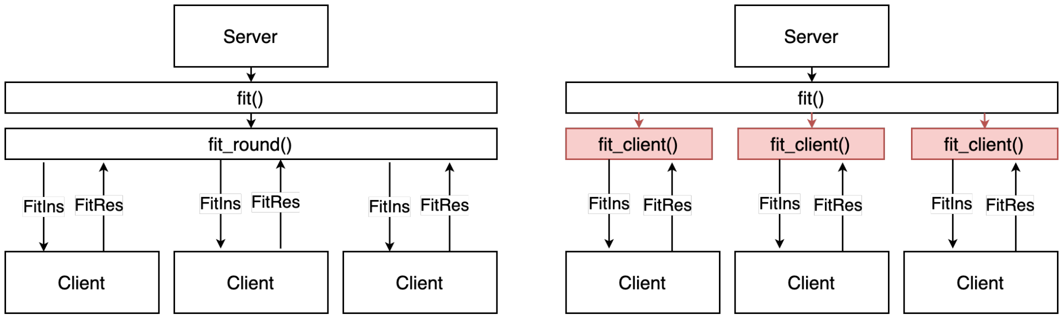


Figure 4: Original training structure (*Left*) and modified training structure (*Right*)

5.1.2 Simulating Resource Heterogeneity

To evaluate an FL algorithm empirically, it is crucial to simulate a sufficient number of heterogeneous clients. We used the combination of Docker [14] and Kubernetes [15]. Docker containers provide an easy interface to simulate clients with different resources, and Kubernetes helps to manage containerized applications, which, in our case, are programs running SGD to compute local updates. We created the dockerfile according to our implementation and packed the running environment inside the docker.

5.2 Stale Synchronous Parallel Implementation

5.2.1 SSP in FL

SSP-FL has the following characteristics [8]:

- Each client should train in an asynchronous manner. Once a client returns a training result, the server should immediately update its global model accordingly;
- If a client's cached model is not s-old, the client should continue training using its cached model;
- If a client's cached model is more than or equal to s-old, the client should request the server to obtain the most updated global model;
- If the fastest client exceeds the slowest client by s clocks, the fastest client will be blocked until the difference becomes smaller than s.

The global parameter s is the *staleness bound* of the SSP model. A *clock* is a training round of the client. When the client receives a global model from the server, it will update the cached global model and its clock.

5.2.2 Design and Architecture

Taking the above mentioned characteristics into consideration, we observe that two more messages are necessary to implement SSP on top of Flower. We introduced `ReadIns` and `ReadRes` messages, namely Read Instruction and Read Result. Read Instruction will be sent to the server from a client if it finds out its cached clock is smaller than its current clock by the staleness bound s.

The complete workflow of our design is shown as follows:

1. The server launches and begins to wait for incoming connections.
2. All clients launch, connect to the server and are registered in the client manager.
3. The server initializes an initial global model.
4. The server enters `fit` function, creating threads running `fit_client` for all clients.
5. `fit_client` sends a `FitIns` to all registered clients, informing them to start training and also tells them their current clock.
6. After receiving a `FitIns`, the client first checks if its cache is stale. The client does that by checking if the clock in `FitIns` is bigger than its the clock of its cached model by more than s. If yes, they will send a `ReadIns` to request the server's global model. In `ReadIns` the client also adds a parameter called `min_clock`, which is the expected minimum *global clock*. The global clock is the smallest clock across all clients. If the global clock does not reach `min_clock`, it means that this client needs to be blocked until the global clock increases to `min_clock`. `min_clock` is calculated by `current_clock`$-s$. If the cache is fresh, the client will proceed to training in step 8. In the first round, clients notice that they do not hold any local model, so they consider their cache stale and send a `ReadIns`.
7. The server receives `ReadIns` and `min_clock` of that client. The server then checks if the `min_clock` is ahead of global clock by more than s. If yes, this `fit_client` will be blocked and no further instructions will be given to the client. If no, it sends back its current global model by a `ReadRes` and also invokes the client's `read` to read the response.
8. All the clients now ensure that they have a fresh model to work on before training. They start training and return the new weights in `FitRes`.

9. A `fit_client` thread receives the `FitRes`, it updates the global model with the response. It then checks the clocks of all the clients and updates the global clock. During this update, if the server finds out that a blocked client's `min_clock` has been reached, it will be unblocked. Then the `fit_client`, which is in a loop, fires another `FitIns` with an incremented clock. The client handles the instruction by repeating step 6.

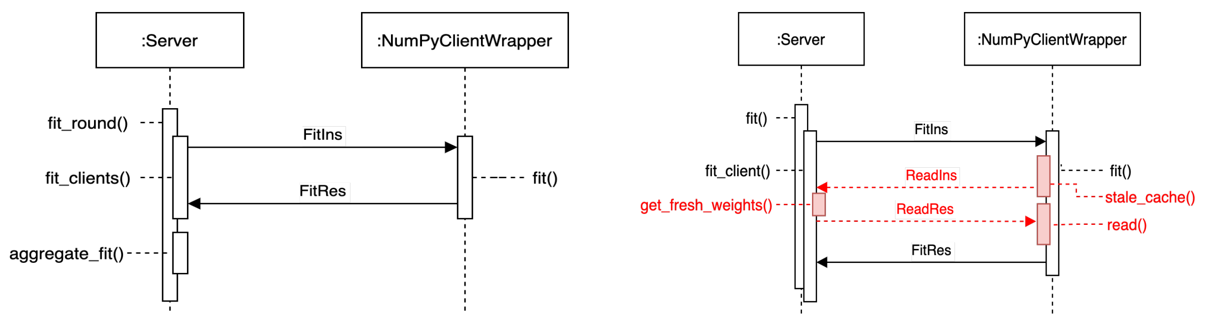
10. After all clients have finished their rounds, i.e. all `fit_client` threads have completed their jobs, the server evaluates the model by aggregating the loss from their respective test data. Then the server sends out shutdown signals to clients and exit.

Major differences between our implementation and the original Flower implementation:

- **The client is aware of its current round.** In the original Flower implementation, to simplify client structure, the client is only responsible for training and unaware of the whole process. In our implementation, the client needs this information to decide the staleness of its cached model.

- **FitIns does not carry the global model.** Flower implements BSP-FL, so the client gets a copy of the global model for every training round, carried in `FitIns`. In our implementation, clients only fetch the global model when its cache is stale, so it makes no sense to attach the global model with `FitIns`. Weights are passed in `ReadRes` only.

- **The server keeps a global clock to track the global model progress.** In Flower's original implementation, all clients are at the same round due to the single thread implementation of the server. In our case, a global clock is introduced to save the current progress of the global model, as the client needs to cache this clock to calculate the difference between local and global progress.



(a) Original Flower implementation (b) Our implementation

Figure 5: Comparisons of implementations

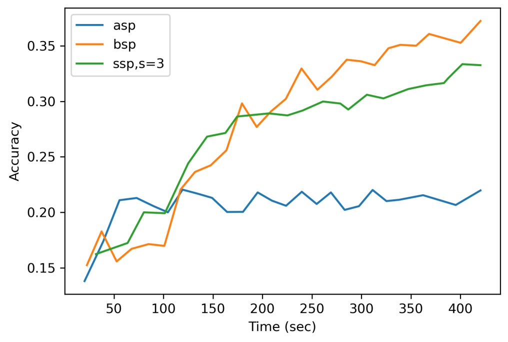
5.2.3 Testing Plan

To obtain the preliminary performance of our SSP FL implementation based on Flower, we designed a simple testing plan to simulate FL conditions.

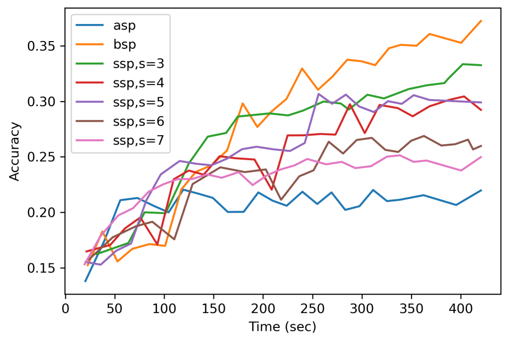
- **Nodes** For this simple test, we use a single 6-core machine to run the processes of the server and clients.
- **Clients and Training Rounds** We open 3 clients for training and each one trains 25 rounds.
- **Datasets** We prepare CIFAR-10 [16] in a non-IID way so that the clients are given random parts of the dataset.
- **Resource Heterogeneity** We implement a random delay from 0 to 10 seconds (both inclusive) during each training round to simulate possible heterogeneity in computational resources and network conditions.
- **What to Collect** We are interested in the time spent and the accuracy of the global model after each training round. We also collect the sum of the computing time of all three clients.
- **Removing Outliers** We obtain the results for 20 times and get the median to remove possible outliers caused by different resource allocation on the node.

5.2.4 Results and Analysis

The plots of the results are shown in Figure 6 and 7.



(a) ASP, BSP, and SSP(s=3)



(b) All SSP tests

Figure 6: SSP-FL training accuracy

Although the test requires a more comprehensive design to stand against more scrutiny, we can still observe certain patterns from the preliminary results.

From Figure 6a, where we compare a representative SSP ($s = 3$) against BSP ($s = 0$) and ASP ($s = \infty$), we notice the following observations:

- ASP-FL starts out with a faster convergence speed, but ends up with a low accuracy;

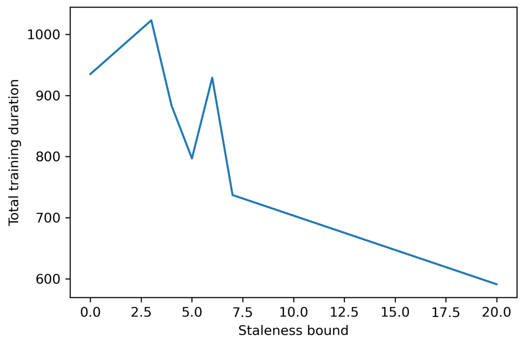


Figure 7: Total training time of different s

- BSP-FL slowly converges, but it can reach a higher accuracy ceiling;
- SSP-FL is in the middle, with a convergence speed faster than BSP-FL but slower than ASP-FL, and an eventual accuracy higher than ASP-FL but lower than BSP-FL.

In 6b we can further observe that different s values also have impact on the convergence speed as well as the accuracy ceiling. Generally a bigger s (closer to ASP) will result in a faster convergence speed and a lower accuracy ceiling. On the other hand, decreasing s (closer to BSP) will generate a higher accuracy in the end, but may suffer from a slow start.

In Figure 7, we can find a negative correlation between the value of s and total training time. With a bigger s, the situation of waiting for stragglers is less likely to occur, so the training time can be saved for a fast worker.

These findings identify with the challenges mentioned in 3, and we can see these challenges still persist in FL settings with non-IID datasets and the difference between the the local optimums and the global optimum. With difficulty in finding a suitable s, a new strategy is required to get a better chance achieving better accuracy with an efficient training speed.

6 Future Works

6.1 More Comprehensive Testing

Our current testing plan is not comprehensive enough and we need more solid test results to prove our expectation about s, on which the adaptive SSP-FL strategy is based. We plan to improve in the following aspects.

- **Node** We will use Kubernetes to launch the framework in containers on more nodes.

- **Clients** At least 16 clients will be used in our refined test, as client numbers are large in real-life FL.
- **Datasets** Instead of using CIFAR-10, we will use more renowned bench-marking datasets specially designed for FL, such as LEAF to better simulate real situations [17].
- **Resource Heterogeneity** We will simulate different bandwidth conditions using Kubernetes and still use random delays to simulate differences in computational resources.

6.2 Adaptive SSP-FL

One core problem related to the adaptive SSP-FL strategy we introduced in 4.2 is how to detect switching points effectively. One simple strategy is to monitor the accuracy changes in the training process, which is similar to detecting model convergence. Also, [18] monitors the product signs of consecutive gradient updates for their adaptive strategy. These techniques are candidates for our strategy, and their effectiveness remain to be evaluated.

References

[1] H. Brendan McMahan, Eider Moore, Daniel Ramage, Seth Hampson, and Blaise Agüera y Arcas. Communication-efficient learning of deep networks from decentralized data, 2017.

[2] Mu Li. Scaling distributed machine learning with the parameter server. In *Proceedings of the 2014 International Conference on Big Data Science and Computing*, BigDataScience '14, New York, NY, USA, 2014. Association for Computing Machinery.

[3] Tian Li, Anit Kumar Sahu, Manzil Zaheer, Maziar Sanjabi, Ameet Talwalkar, and Virginia Smith. Federated optimization in heterogeneous networks, 2020.

[4] Cong Xie, Sanmi Koyejo, and Indranil Gupta. Asynchronous federated optimization, 2019.

[5] Ming Chen, Bingcheng Mao, and Tianyi Ma. Efficient and robust asynchronous federated learning with stragglers. In *Submitted to International Conference on Learning Representations*, 2019.

[6] Xing Zhao, Aijun An, Junfeng Liu, and Bao Xin Chen. Dynamic stale synchronous parallel distributed training for deep learning, 2019.

[7] Yuzhen Huang, Tatiana Jin, Yidi Wu, Zhenkun Cai, Xiao Yan, Fan Yang, Jinfeng Li, Yuying Guo, and James Cheng. Flexps: Flexible parallelism control in parameter server architecture. *Proceedings of the VLDB Endowment*, 11(5), 2018.

[8] James Cipar, Qirong Ho, Jin Kyu Kim, Seunghak Lee, Gregory R Ganger, Garth Gibson, Kimberly Keeton, and Eric Xing. Solving the straggler problem with bounded staleness. In *Presented as part of the 14th Workshop on Hot Topics in Operating Systems*, 2013.

[9] Jiawei Jiang, Bin Cui, Ce Zhang, and Lele Yu. Heterogeneity-aware distributed parameter servers. In *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD '17, page 463–478, New York, NY, USA, 2017. Association for Computing Machinery.

[10] Qirong Ho, James Cipar, Henggang Cui, Seunghak Lee, Jin Kyu Kim, Phillip B Gibbons, Garth A Gibson, Greg Ganger, and Eric P Xing. More effective distributed ml via a stale synchronous parallel parameter server. *Advances in neural information processing systems*, 26:1223–1231, 2013.

[11] Hanpeng Hu, Dan Wang, and Chuan Wu. Distributed machine learning through heterogeneous edge systems, 2019.

[12] Daniel J. Beutel, Taner Topal, Akhil Mathur, Xinchi Qiu, Titouan Parcollet, and Nicholas D. Lane. Flower: A friendly federated learning research framework, 2020.

[13] Daniel J Beutel, Taner Topal, Akhil Mathur, Xinchi Qiu, Titouan Parcollet, and Nicholas D Lane. Flower: A friendly federated learning research framework. *arXiv preprint arXiv:2007.14390*, 2020.

[14] Dirk Merkel. Docker: lightweight linux containers for consistent development and deployment. *Linux journal*, 2014(239):2, 2014.

[15] D. Bernstein. Containers and cloud: From lxc to docker to kubernetes. *IEEE Cloud Computing*, 1(3):81–84, 2014.

[16] Alex Krizhevsky. Learning multiple layers of features from tiny images. Technical report, 2009.

[17] Sebastian Caldas, Sai Meher Karthik Duddu, Peter Wu, Tian Li, Jakub Konečný, H. Brendan McMahan, Virginia Smith, and Ameet Talwalkar. Leaf: A benchmark for federated settings, 2019.

[18] Serge Kas Hanna, Rawad Bitar, Parimal Parag, Venkat Dasari, and Salim El Rouayheb. Adaptive distributed stochastic gradient descent for minimizing delay in the presence of stragglers, 2020.