# Comparative Study of Distributed Training Approaches

Karan Dabas
*Arizona State University*
Tempe, Arizona, USA
kdabas1@asu.edu

Mahidhar Reddy Dwarampudi
*Arizona State University*
Tempe, Arizona, USA
mdwaram1@gmail.com

Siddharth Kale
*Arizona State University*
Tempe, Arizona, USA
skale9@asu.edu

***Abstract*—An exponential growth of users on internet and vast space of cyber-physical systems where technology interact with our environment on daily basis while coming up with strategies to perform certain tasks such as object recognition for autonomous driverless cars or amazon go where people can pick and go and the in-store setup takes care of detecting items you picked and bill you on it. These are some of the places where industry make use of deep learning to train big bulky models to achieve certain objectives. Training such networks require large amount of hardware resources and time. In this comparative study, we will be analyzing various approaches to distributed training over different experimental setups such as training on a single GPU, and using different types of parallelisms such as Data, Model and Pipeline for comparison. The key contribution will be to evaluate such techniques over parameters like average GPU utilization(on the standalone and multi-GPU system), time taken to reach a target accuracy, average training time per epoch, overall training time among others. Specifically, we want to look into differences between (1) standalone training and data parallelism with multi-GPU setup, (2) Gpipe and model parallelism approach for the case of the large model architecture. We will investigate their overall usefulness among different settings and uncover the advantages and disadvantages of each backed with our evaluation metrics. The use case of this project is to serve as a guide to finding the best choice for each experiment that makes the most of the hardware resources provided with least training time.**

***Index Terms*—Deep Learning, Distributed Systems, Stochastic Gradient Descent, Communication, Parallel Optimization Algorithms, Gpipe, model parallelism, pipline parallelism**

## I. Introduction

Over the last decade, we have seen a tidal wave of advancements in hardware and various deep learning architectures catered to different tasks in the industry such as CNN, RNN, LSTM, R-CNN and many more. These techniques have achieved state-of-the-art results in various domain-specific tasks and are highly data-intensive. As we increase the amount of the data or the size of these models the overall computation spaces increases and thus expose us to various problems like high training time, costly hardware support, partial utilization of given resources among different training approaches. People in industry and academics have started adopting techniques that can help them train these bulky models on a multi-GPU systems which are costly and many a times are not even utilized to its full potential. We as a team wants to divert out attention towards uncovering the differences among distributed training techniques: data parallelism, model parallelism and pipeline parallelism at the level of utilization which matters the most when someone is trying to pick an approach for a task.

## II. Problem Formulation

As discussed in the previous section, the current techniques of DNN are highly data intensive. There is huge data influx, and everyone in the community is moving towards adopting a Big Data solution. A solution that can handle large data-set for training, for a very large model, and provide results that are acceptable according to industry standards. Achieving all this within a given amount of time is very hard, and keeping the cost down is a problem for many. Currently that are various parallel training approaches as discussed in the section III. Each one has their unique traits and could be beneficial in different kinds of scenarios.

Here we have a scenario at hand where a data scientist is presented with a huge model with the parameters ranging in about few thousands. Handling these many parameters for any single device, even if that device is a GPU is hard. Moreover to train such a huge model and draw inference from it, the scientist is given a huge data-set. The scientist has formulated two different kinds of models for this specific task. Imagine the first model is small but requires a lot of data samples, and the other model is too huge to be fit into one device's RAM.

Keeping in mind the above requirements, we want the user to be able to make a decision on what kind of parallelism approach they might want to try. We are comparing various parallelism techniques and evaluating the inferences in terms of training time and costs. The cost here could be hardware cost or even actual monetary cost.

To ease the burden of the user, we are focusing on creating a thorough report about different kinds of models that we are using for different kinds of tasks combined with different kinds of data sets. We are using different approaches to train these models. We focus on drawing inference about the training time, cost, model performance, time required to reach a certain accuracy, average time per epoch, usage level of the machines. With this information, we believe a user will be able to make a decision about what kind of parallel technique to incorporate, or if the technique will actually be beneficial rather than using a traditional setup.

## III. Literature Review

Before I jump into different distributed training approaches lets get a quick recap of the what is stochastic gradient descent, how it works, showcase its inherit nature and underlying challenges faced while trying to parallelize it.

### A. *Communicating gradients differently*

There exists different methods at making training distributed and will first start with defining what synchronous and asynchronous updates means and later dive into explaining different approaches proposed by researchers in past and how they fall into either of these two buckets.

Let's first define a parameter server(PM), it is used to keep record of the global model parameters and is connected to one or many worker nodes/machines. In a multiple worker node setup the each worker may either keep track of partial/parts of the global parameters or write in parallel asynchronously. The workers can request to push/pull parameters from the server. The only drawback is that of bandwidth bottleneck on parameter-server side. The read and writes to and from parameter servers can be done synchronously or asynchronously.

In the synchronous gradient update, the PM only updates the entire model after all the workers have sent their gradients. Such approach limits the speedup to the slowest worker as fast workers will have to wait for slower workers. An important characteristic of the synchronous update is that the PM gets the most latest gradients of all the workers and it does not hamper the overall model convergence. But the idea of making fast worker wait and say idle can lead to **straggler problem** that slows the training time.

With asynchronous updates we overcome the straggler problem. The fast workers do not wait for slow workers. A worker may send its local gradients to the PM while others are still calculates their gradients. The preeminent challenge with asynchronous updates is that of **data staleness**, due to the fact that fast worker may end up using stale gradients from PM which can imperil the model convergence. Additionally, the fastest worker may update its local parameters by its own sub-dataset, which can cause the local worker model to deviate from the global model. Such issues can be constrained by limiting the staleness in general. Next I will discuss various distributed training approaches and show you where they fall-in and how did they tackle the issues associated with staleness or straggler based on type of communication.

### B. *Parallel Training Approaches*

*1) Data Parallelism:* According to paper [6], to achieve Data Parallelism. We have a series of workers that we make use of for the computation. Each of these workers maintains a local copy of the model we are working on. Each local copy has its own sets of weights. The inputs for the models are then partitioned and send to different workers in the network. During the training phase, all the workers periodically synchronize the weights with all other workers(see Fig-1). Most commonly used techniques for weight synchronization are parameter averaging, update based approaches and synchronous, asynchronous methods. Interestingly this technique scales up well with the amount of data but has high communication cost. Data parallelism can be thought of as a distributed training approach, as the training examples are distributed among the workers. [7]

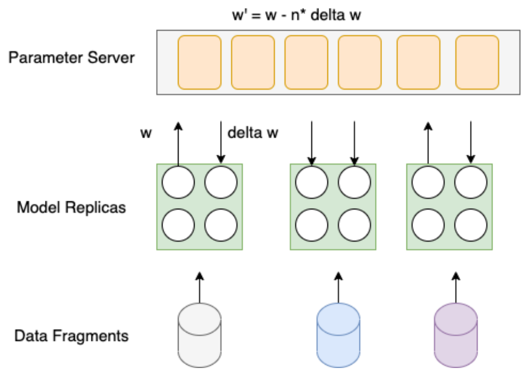

Fig. 1. Data flow architecture with data parallelism

*2) Model Parallelism:* In model parallelism, we have a set of workers. Each worker in the set attains a partial subset of a model that we are training. The model we are training is divided into subsets, and each of the partitioned subset is sent to all available workers. Each worker is responsible to update the parameters of their assigned subset of the model.Unlike Data Parallelism, here each workers in the set of all available workers get the same inputs. And the inferences are drawn up to the same inputs albeit for different parts of a model.

Model parallelism is mainly used for very large models, where training the model on a same compute machine, be it CPU or GPU would not be possible. It could be due to the fact that the model is so large that the machine is unable to hold all the data of the model(parameters), at once. But, at the same time, using model parallelism, there are few issues as discussed in the paper [6]. First one is under utilization of the resources. And the second is that the burden on properly partitioning the model for different GPUs fall on the shoulder of a programmer.

*3) Pipeline Parallelism:* Pipeline Parallelism: Pipeline Parallelism is an intelligent combination of Model Parallelism and Data Parallelism. Model Parallelism has a huge Drawback, in most cases, when one GPU is running computations for a few layers all the other GPUs are Idle, waiting for the outputs from that GPU. To utilize most of the GPUs during this time, Pipeline Parallelism leverages over Data Parallelism, it split the batch of data to smaller batched and splits them over multiple GPUs using Data Parallelism, so The GPUs in the beginning won't be idle instead would be processing smaller batches of Data. It does not guarantee 100% utilization of GPUs, but it is still better than Model Parallelism.

Some of the common features of Gpipe and PipeDream are that they have data parallelism. The Data is divided into multiple batches for training. Similarly, both have Model Parallelism as well, the model is distributed over multiple GPUs. So essentially batches of data flow through various

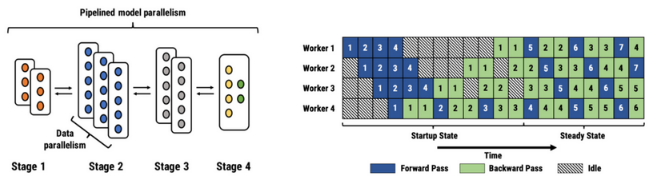

Figure 2: In the figure on the left, we show an example pipeline-parallel assignment with eight GPUs split across four stages. Only activations and gradients at stage boundaries are communicated. Stages 1, 2, and 3 have replicated stages to ensure a load-balanced pipeline. The figure on the right shows another pipeline with 4 workers, showing start-up and steady states. In this example, the backward pass takes twice as long as the forward pass.

Fig. 2. Asynchronous updates, where blue nodes represent workers and green nodes are servers. **referenced from [1]

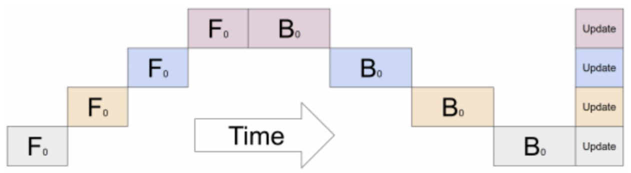

Fig. 3. Model Parallelism: F0 is one batch of data flowing through multiple GPUs

GPUs containing different layers of a model. Both Gpipe and Pipedream have pipeline level parallelism, but slightly different implementations.

The difference between the two frameworks is how PipeLine Parallelism is implemented. In GPipe, it's exactly implemented as the definition on the top, Batches are divided into micro batches and these batches flow through the GPUs, this will keep the GPUs busy for most of the time, very efficient. The parameters flow back through all the GPUs and updated along with the batches using distributed stochastic gradient. So, it has a mixture of intra batch and inter batch parallelisms.

PipeDream is slightly different: PipeDream takes it slightly to the next level. PipeDream basically injects the batches or mini batches of data into the Pipeline and keeps them in steady state and makes sure that all workers are busy by not only shifting the data around but also changing the function of the GPU from forward to backward. It also uses far less data transfer between each GPU as it will only send the activation functions and gradients. The below image shows the working of pipe dream, and you'll see that more workers are kept busy when compared to Gpipe.

Most Systems that we discussed in the class like Hogwild, DistBelief, Petumm and Horovod, only used one type of parallelism, either data parallelism or Model Parallelism. Neither used both. Even if they used both, the implementation was not as efficient and, in most cases, either other GPUs must wait or there were too many network calls, which increased the chance for failure. PipeDream and Gpipe has efficient pipeline level parallelism, which is not efficiently implemented in any of the other systems for distributed learning.

Most of the above mentions framework rely mostly on higher band width for data processing, none of them have advanced pipeline level parallelism to divide the batches to smaller micro batches to save band width or only share activations and gradient boundaries instead of all the weights.

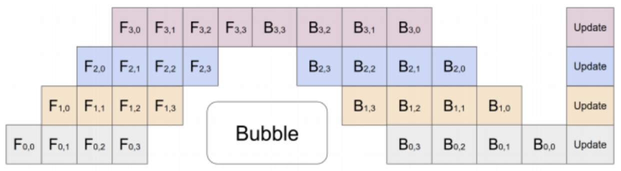
$F_{3,0}$ $F_{3,1}$ $F_{3,2}$ $F_{3,3}$ $B_{3,3}$ $B_{3,2}$ $B_{3,1}$ $B_{3,0}$
$F_{2,0}$ $F_{2,1}$ $F_{2,2}$ $F_{2,3}$ $B_{2,3}$ $B_{2,2}$ $B_{2,1}$ $B_{2,0}$
$F_{1,0}$ $F_{1,1}$ $F_{1,2}$ $F_{1,3}$ $B_{1,3}$ $B_{1,2}$ $B_{1,1}$ $B_{1,0}$
$F_{0,0}$ $F_{0,1}$ $F_{0,2}$ $F_{0,3}$ $B_{0,3}$ $B_{0,2}$ $B_{0,1}$ $B_{0,0}$
Bubble
Update

Fig. 4. Pipeline Parallelism: F0 is divided into multiple smaller batches F(0,0), F(0,1), F(0,2), F(0,3)

## C. DistBelief: Downpour SGD

In [3], the researchers at google proposed a large scale distributed network called DistBelief with two different algorithms, I will focus on the Downpour SGD technique (see Fig-1). The idea is quite intuitive, it is a variant of stochastic gradient descent with asynchronous update methodology. This technique all partition the data into fragments and create multiple copies of the model on each worker. The training is done with each worker with their own data fragment in parallel. The workers communicate updates through a centralized parameter server that stores the recent state of all the parameters for the model. Each worker updates a component of the complete parameter server. This is different that hogwild in many ways such as, each worker has its own replica of the entire model instead of having a part of it and the communication of parameter updates are made to parameter servers but here each worker responsible for updating the subset of parameters that fall under it thus non-overlapping. For example if their are 20 workers then each worker is tasked to apply updates to 1/20th of the model parameters. The biggest concern here is that of communication overhead but the authors of this technique also proposed a solution by limiting the number of update requests from parameter servers only to every n steps and send update gradient values after every n' steps where n and n' may or may not be equal to each other. The model replicas/ workers have complete freedom to send or receive updates independently from other workers. The only downside of downpour SGD is that its still require high bandwidth as the updates are at high frequency for synchronization with parameter server. As found out in the mentioned paper, this strategy works astonishingly well with noncovex models as well.

## D. Scale and Heterogeneity aware Asynchronous distributed training algorithm

Lastly I would like to cover a more recent approach from paper [5], the system proposed here is called Scale and Heterogeneity aware Asynchronous distributed training algorithm (SHAT). SHAT takes into account two main concerns (1) the scale (amount of workers) of distributed training, (2) the diversity among individual workers to reduce the local differences among them. The main idea here is that instead of workers replacing its local weights with that of global parameters, the workers take into account the scale of distributed training and the updates must reflect it. This is done by creating a degree of staleness for each worker. Such inclusion helps on two folds: (1) the worker models are combined more effectively and (2) the workers catch up to the learning direction of the global model more quickly. In comparison to other systems above that primary focuses on asynchronous communication strategy with a simple update technique for workers from PS, this approach uses the scale and difference among workers into account to boost-up the convergence.

## IV. Experimental Setup

For the evaluation of the proposed solution to the problem we have at hand, we explored various cloud service vendors which could provide us with a GPU cluster. This section will be discussing which cloud vendors we tried out and why we went ahead with a particular one. We have selected Paperspace to do our evaluations on. The other vendors and the problems we faced with them are discussed below.

### *A. Google Colab*

Google Colab is a great tool provided by Google Research for people to use. You can use Upto 32 GB of GPU memory which is made instantly available. It provides an interactive notebook to work with and is connects with your Google Drive where you could upload all of the project data. It's simple interface and easier to use services made it our first choice. But there was a problem that we needed 2 GPUs. The problem with Google Colab is that if your increase the GPU memory it allocates a single bigger GPU not two of those. As we needed multiple GPUs we couldn't really use Google Colab.

### *B. AWS*

Our second choice was AWS. It was due to the fact that it is the largest cloud provider in the world, moreover, all of our team were familiar with it. We wanted to acquire g3.8xlarge EC2 instance that would have provided us with 2 GPU. Also the cost of running this instance was fair at a rate of §1.076 per hour. But to get this large of an instance one has to make an request to AWS for additional resources allocation. We sent out that request to AWS but they denied our request due to the ongoing GPU and other chipsets shortage.

### *C. Google Cloud Services*

Google Cloud Services is the second best available in the market. While it is rate is a bit higher, due to us not getting our GPU allocation from AWS, we tried to use Google Cloud. But the problem occurred that it required one to have a significant amount of history to request a GPU cluster allocation.

### *D. Agave Cluster*

ASU provides a very good service to the students. Students can request for access to the Agave cluster which is a distributed computing platform. It has various different GPUs which can be requested by submitting a job. The job has to take into account the time required for the task to finish. Also one has to specify the GPUs required. Furthermore, Agave has various preinstalled modules that one can make use of. Agave also provided a interactive mode which is very helpful if making of GUI kind software. But to install any more modules, sudo access is needed. Which is not provided to the students by the administrator. For our project we required to install new modules which was not possible for us.

### *E. PaperSpace*

We went ahead with PaperSpace due to several reasons. One of the biggest reason was we were able to get 2 GPUs for our training and evaluations part. Moreover, it has a very easy to use interface and there is not a very steep learning curve for using this particular Cloud Service. As for our setup we have used 2 RTX 4000 GPUs. Each GPU has memory of 8 GB. For the libraries we have made use of PyTorch 1.11.0, Python 3.8 and CUDA 10.0.

## V. Proposed Solution

We have chosen two machine learning training tasks to solve the given problem. The following gives a brief description of both the tasks:

### *A. Text Detection*

- We are implementing EAST (Efficient and Accurate Scene Text Detector) [13] for text detection.
- We would be separating the layers of this models over multiple GPUs for Model and PipeLine Parallelism.
- The data is a combination of Text detection tasks ICDAR 2013 and ICDAR 2015.
- The would give us enough data for the data parallelism.

### *B. Object Detection*

- We would be using transfer learning, without freezing the initial layers so that the gradients would pass to the initial layers for object detection.
- We are using Imagenet dataset [16] for training. The dataset is big enough for data parallelism.

Now we implement these tasks over 5 different training environments to get two sets of comparable results:

- Training on a single GPU, without any parallelism using multiple batch sizes.
- Training using Data Parallelism using multiple GPUs (same as GPU as the above environment)
- Training using Model Parallelism using multiple GPUs (same as GPU as the above environment), any one of DistBelief, HogWild [2] and SHAT will be used.
- Training using PipeLine Prallelism using multiple GPUs (same as GPU as the above environment), using Gpipe [9].

Finally, we compare the results in the following manner:

- We compare the first 2 environments, single GPU and Data Parallelism. We are comparing these two environments, as model lies in the same GPU and not distributed over multiple GPUs.
- We compare the second 2 Model and Pipeline level parallelism. These 2 environments are comparable because

the model is distributed over multiple GPUs, unlike the above environment.

## VI. Data Description

We have decided go with three different dataset to do the testing part. The Datasets being CIFAR-10, ImageNet - 64x64, and ICDAR - 2015. This is due to the fact that we wanted to draw inferences from a larger section of test cases which is very well covered by the three datasets we have chosen. Furthermore, testing on various datasets for which different kinds of models needed solidifies the trends across a broader set of use-cases. The description and our reason for selecting the data is described thoroughly in this section.

### A. CIFAR-10 Dataset

CIFAR-10 [15] dataset contains 60000 images of various elements. The total number of classes is 10. Each image belonging to a particular class has a size of 32x32. Every single image is RGB. Our reason for choosing this dataset was to have a dataset which is large enough to test Data parallelism on, and meanwhile light and simple enough to establish a benchmark on. This dataset is perfect for that as the size of the images are not too small while they are simple and small enough to test on a model that is not too complex. Also, 60000 was enough to test the Data Parallelism approach and still get satisfactory results to compare other results with. Moreover, this dataset allowed us to have complex models running which can actually draw some load on the GPUs.

### B. ImageNet Dataset

Imagenet is a vast dataset of images containing over 1 million images. The version we have specifically used is the ImageNet - 64x64. It contains 1331167 image samples, this includes both our training and testing samples. We choose this dataset it has a wide range of images, according to [16] there are over 80000+ nouns. This gives us a very large dataset to work with which is good for our distributed training system. We went with this one for one more reason being that the 64x64 provided a more complex data compared to the CIFAR-10 32x32 images. This was necessary as we had to test with a more rigorous data which might generate a little more load on the GPUs. Moreover, the total size on disk for this dataset unpacked is about 12.5 GB. which is a lot of data given the size of each image is just 64x64.

### C. ICDAR Dataset

ICDAR(International Conference on Document Analysis and Recognition) is a conference held each year, it is a premier conference in the Document Analysis field. We choose to use their 2015 challenge dataset. This dataset was particular for Incidental Scene Detection. We choose the specific dataset of text segmentation. This is due to the fact that we wanted to test a model which is able to do text recognition. Our idea behind this was that it is complete different kind of problem given the previous two datasets were focusing on image recognition. Moreover, using this dataset allowed us to test certain models like EAST, which has a lot of interconnected components. This is further discussed in the results section.

## VII. Results

Initially we trained tried testing Gpipe by running it on a small CNN model, with CIFAR-10 dataset. Our results were pretty standard. From Figure 5 we can see that as the batch size increases the accuracy decreases.

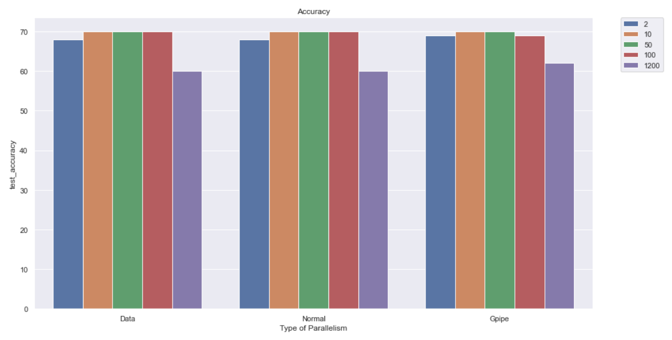

Fig. 5. Final Accuracy after same number of epochs on different models for a small CNN

Similarly, the time taken for each epoch decreases as batch size increases from Figure 6.

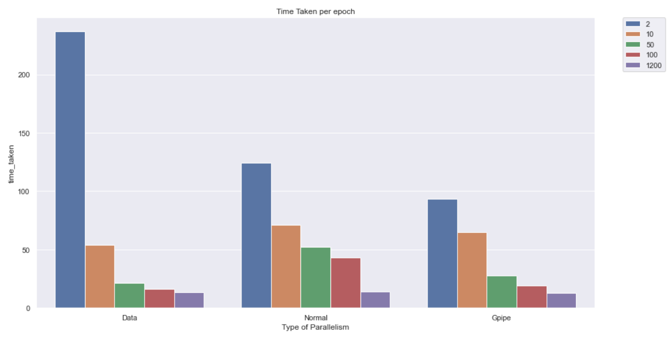

Fig. 6. Total training time for same number of epochs for the CNN

This is when we realized that neither our model nor our data was Data Intensive, so we have decided to more to larger models and bigger datasets. The first model we chose was ResNet-101. For ResNet-101, we used the Imagenet dataset. Here are are our results.

First we measure average taken to epochs. Lower is better. Data Parallel took the lowest time and Model Parallelism takes the highest time. Gpipe [Time] takes a longer time than Data Parallelism.

For average time taken per epoch: Data Parallel $>$ Gpipe[Time] $>$ Gpipe[140,230] $>$ No Parallelism $>$ Gpipe Size $>$ Model Parallelism. Data Parallel beats Gpipe[Time] by nearly 36%.

Next we compared, maximum and average GPU utilization. We found out that Data Parallelism and Gpipe[Time], which had very close max time utilization, but when we looked at average utilization, Gpipe[Time] is significantly lower than Data Parallelism Figure 8.

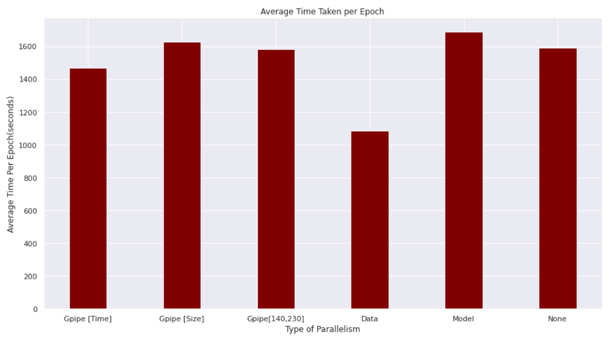

Fig. 7. Resnet Average time taken per epoch

For Max GPU Utilization (higher is better): Gpipe[Time] > Data Parallelism > Gpipe[140,230] > Gpipe[Size] > Model Parallelism > No Parallelism.

For Average GPU Utilization (higher is better): Data Parallelism > Gpipe[Time] > Gpipe[140,230] > Gpipe[Size] > No Parallelism > Model Parallelism

We should also keep in mind that No parallelism has 100% max and average GPU utilization, which is maximum for a single GPU, for all of the others it is more than 100% except model parallelism.

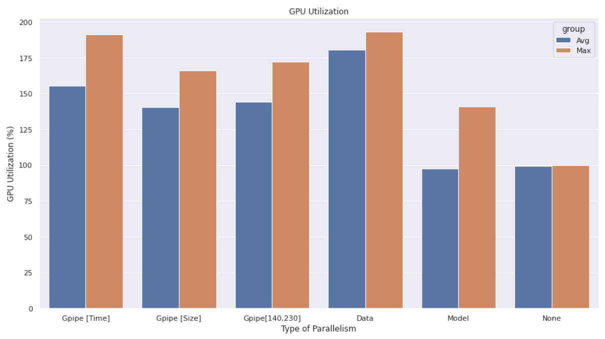

Fig. 8. Resnet Average and Max GPU Utilization

We then compared average GPU Memory usage. Which also follows a similar trend, Gpipe beating all of the other by atleast 10%.

For Max GPU Memory Utilization (Figure 9, Higher is better): Data Parallelism > Gpipe[Time] > Gpipe[140,230] > Gpipe[Size] > No Parallelism > Model Parallelism

In throughput, predictably Data Parallelism performed the best almost 27% greater than the next best Gpipe[Time]. We made 2 important observations Figure 10 (higher is better), Data has highest throughout while training as, the data is divided between 2 different GPUs.

For High GPU Utilization while training: Data Parallelism > Gpipe[Time] > Gpipe[140,230] > No Parallelism > Gpipe[Size] > Model Parallelism

Second observation was, when there was no back-propagation, in validation steps all the scenarios tried to push as many samples as possible, as steps increased,where

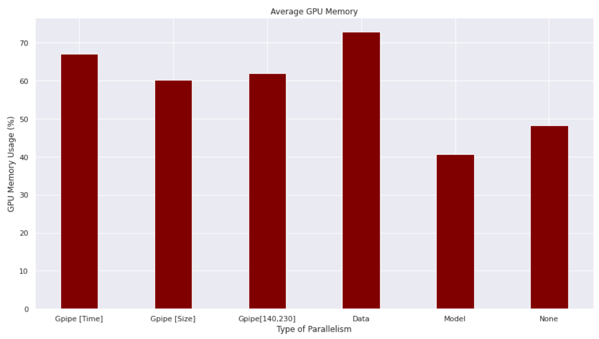

Fig. 9. Resnet Average GPU Memory Usage

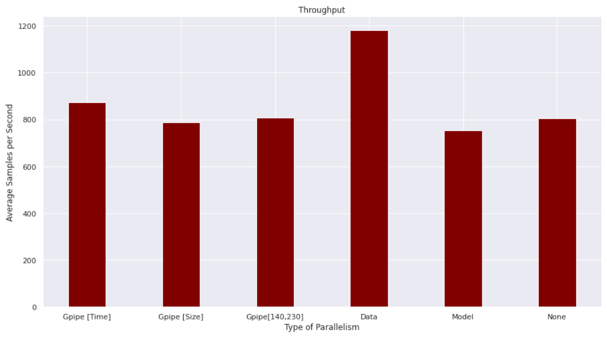

Fig. 10. Throughput/ Samples per second while training in Resnet

Gpipe[Time] performed the best. Figure 11 shows how they vary over time.

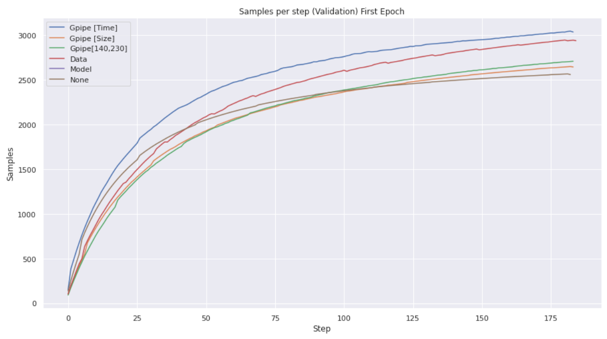

Fig. 11. Samples processed per step while Validation in Resnet

Then we finally looked at the final accuracies after 5 epochs in all the scenarios, this is where Data Parallelism did not beat most of the other frameworks.

Gpipe[Time] and Gpipe[Size] performed the best in scenario.

Figure 12 shows that, Gpipe[140,230] > Gpipe[Time] ≈ Gpipe[Size] > No Parallelism > Data Parallelism > Model Parallelism

When we looked into the accuracy per epoch graph, we figured out why the Data Parallelism didn't perform well.

Figure 13 and Figure 14, shows how the accuracy increases and loss decreases. We see that in all of the scenarios the accuracy consistently accuracy consistently decreases. Weirdly

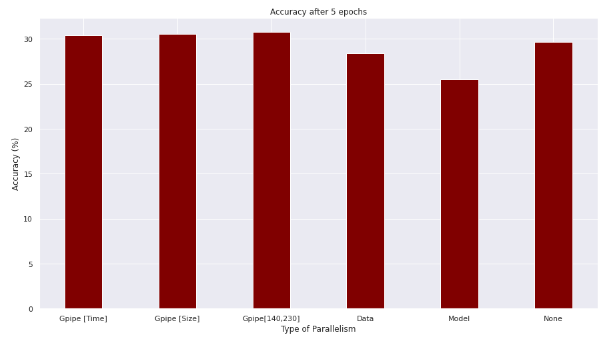

Fig. 12. Resnet Accuracy after 5 epochs

enough our Gpipe[140,230] performed the best in this scenario, but we cannot always guess the correct layer split.

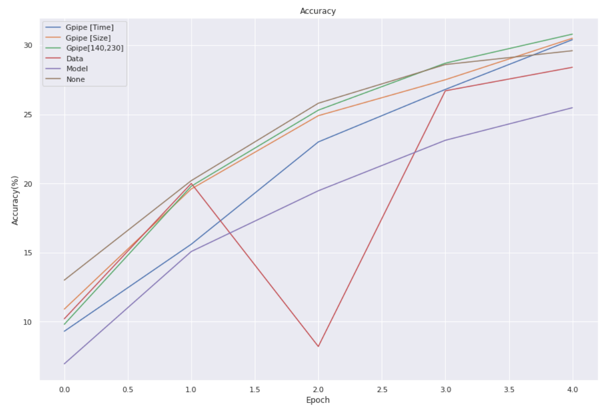

Fig. 13. Resnet Final Accuracy

So, it is better go with Gpipe[Time] or Gpipe[Size], as both of them performed consistently well. If we consider time and memory, Gpipe[Time] has the best performance. If we are aiming for higher throughput and GPU Utilization, then Data Parallelism is the better.

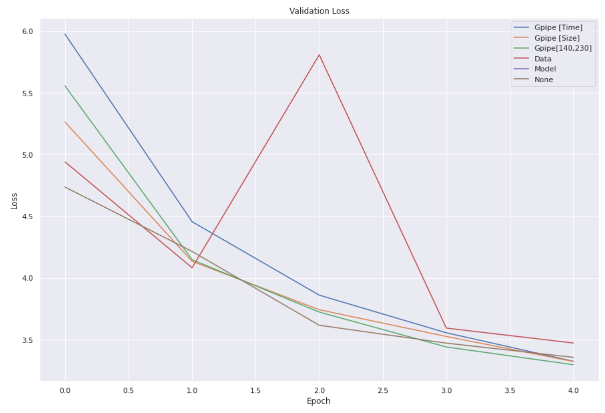

Fig. 14. Validation Loss in Resnet

Next we chose EAST Network, which has many control and data dependencies, which might cause problems when the model is divided into multiple GPUs, as the some layers might be waiting for the previous layer's outputs. Hence, we chose this model to find a good parallelism to choose when training such a model.

For no parallelism, the batch size id 5, as the memory in a single GPU was not sufficient we reduced it from 15. In all of the other scenarios the batch size was 15. As there were too many inter dependencies in the network we carefully observed the network and separated it into 3 parts, the first part was put into the first GPU, the second 2 parts are put into the second GPU.

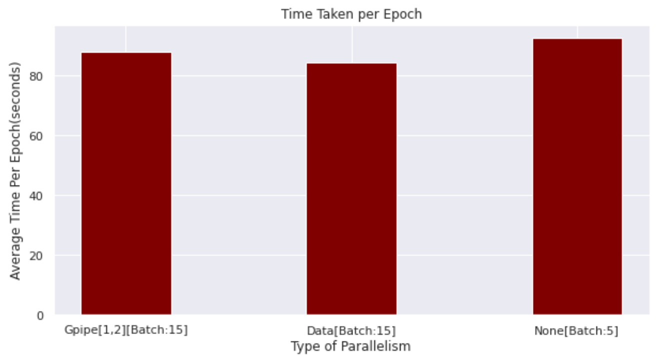

Fig. 15. Time Taken per epoch EAST Network

From Figure 15 and 16, we can see that the training times were similar in all the scenarios.

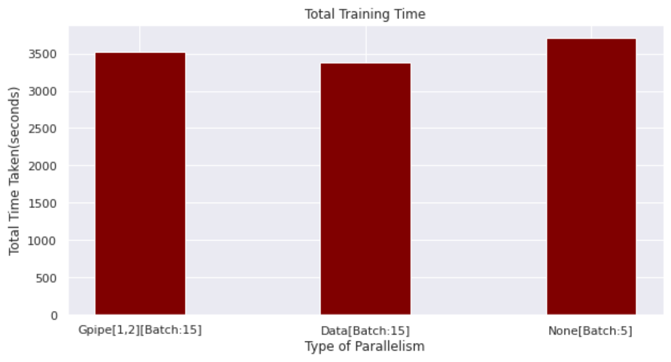

Fig. 16. Total Training time for 5 Epochs EAST Network

East has too many data dependencies and control dependencies, which will cause a lot of waiting, leading to idle GPUs this is the reason why in all the 3 scenarios the training time is similar.

But, when we look at GPU Memory Utilization in Figure 17, we can clearly see that Data Parallel and No Parallel scenarios are at the same level, while GPipe split doesn't perform very well. The memory utilization gains on the Data Parallel system are minute when compared with standalone training.

In 18 we can see that, when we compared the max usage of GPUs in each of the scenarios, Data Parallel and Gpipe are much higher than Standalone, but in case of average GPU Utilization, they don't do so well, especially Gpipe doesn't perform better than Standalone.

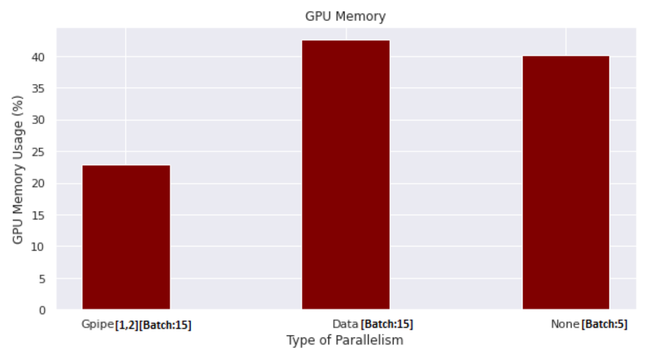

Fig. 17. GPU Memory Utilization for EAST Network

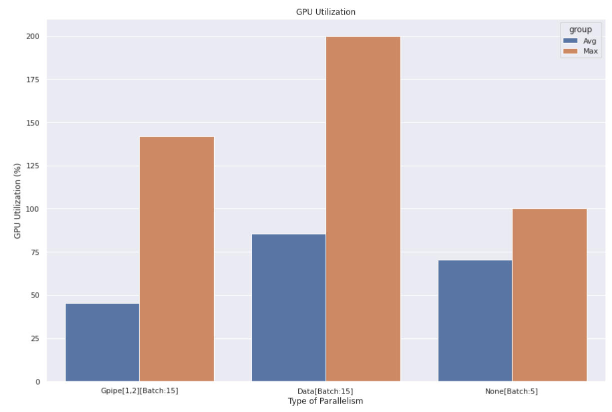

Fig. 18. GPU Utilization for EAST Network

So finally all of this watered down to final accuracy of the models after training for same number of epochs, As EAST has three different types of accuracy, we did not compare accuracy, instead we compared the final loss in the in prediction, Figure 19. Here we saw that, the loss for Data Parallelism and Standalone training with a smaller batch size is very similar. From this we can conclude that for complex models, it is better to use standalone training with a smaller batch size for training, as average GPU Utilization and Time taken to train are similar.

## VIII. Conclusion and Future Work

In conclusion, we can say that for the following scenarios, the corresponding type of parallelism is better for faster training or better GPU Utilization

1) Fastest/ Highest Accuracy [Flattened Model]: Gpipe Time > Gpipe Size.
2) Highest GPU Utilization: Data Parallelism
3) Too many control and data dependencies: Standalone (with lower batch size), Data parallel (larger dataset)
4) Larger Model with control and data dependencies: Gpipe (Component Level Split)

This paper while discusses many parallel distributed training techniques, there is only one technique evaluated for pipeline parallelism. The technique is Gpipe. For future work one can focus on implementing PipeDream and comparing the

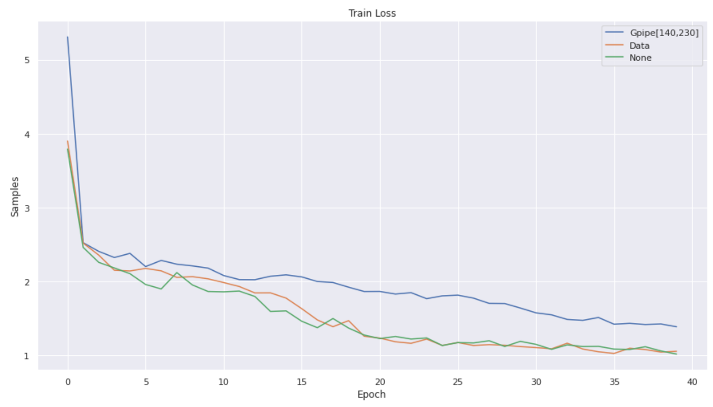

Fig. 19. Train Loss for EAST Network

results with Gpipe. Specially how much does using PipeDream changed the GPU utilization and the throughput for certain scenarios.

Another improvement made to the evaluations would be adding more models to test upon. A lot of different models will solidify trends and will eventually be beneficial in picking up for what dataset and what model a certain technique should be used. One could use Bayesian Networks, or LSTMs, or any other complex models that could drive the GPU utilization to a certain level to have a suitable comparisons of the models.

## References

[1] Shuo Ouyang, Dezun Dong, Yemao Xu, Liquan Xiao, Communication optimization strategies for distributed deep neural network training: A survey, Journal of Parallel and Distributed Computing, Volume 149, 2021, Pages 52-65, ISSN 0743-7315, https://doi.org/10.1016/j.jpdc.2020.11.005. (https://www.sciencedirect.com/science/article/pii/S0743731520304068)

[2] Recht, Benjamin et al. "Hogwild: A Lock-Free Approach to Parallelizing Stochastic Gradient Descent." NIPS (2011).

[3] Dean, Jeffrey, et al. "Large scale distributed deep networks." Advances in neural information processing systems 25 (2012).

[4] Ho, Qirong, et al. "More effective distributed ml via a stale synchronous parallel parameter server." Advances in neural information processing systems 26 (2013).

[5] Ko, Yunyong, and Sang-Wook Kim. "SHAT: A Novel Asynchronous Training Algorithm That Provides Fast Model Convergence in Distributed Deep Learning." Applied Sciences 12.1 (2022): 292

[6] Narayanan, Deepak, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R. Devanur, Gregory R. Ganger, Phillip B. Gibbons, and Matei Zaharia. "PipeDream: generalized pipeline parallelism for DNN training." In Proceedings of the 27th ACM Symposium on Operating Systems Principles, pp. 1-15. 2019.

[7] Shallue, Christopher J., et al. "Measuring the effects of data parallelism on neural network training." arXiv preprint arXiv:1811.03600 (2018).

[8] https://pytorch.org/docs/stable/pipeline.html

[9] Huang, Yanping, et al. "Gpipe: Efficient training of giant neural networks using pipeline parallelism." Advances in neural information processing systems 32 (2019).

[10] https://ai.googleblog.com/2019/03/introducing-gpipe-open-source-library.html

[11] https://www.microsoft.com/en-us/research/blog/pipedream-a-more-effective-way-to-train-deep-neural-networks-using-pipeline-parallelism/

[12] https://www.linkedin.com/pulse/gpipe-pipedream-two-new-frameworks-scaling-training-deep-rodriguez/

[13] Zhou, Xinyu, et al. "East: an efficient and accurate scene text detector." Proceedings of the IEEE conference on Computer Vision and Pattern Recognition. 2017.

[14] Lin, Tsung-Yi, et al. ”Microsoft coco: Common objects in context.” European conference on computer vision. Springer, Cham, 2014.
[15] https://www.cs.toronto.edu/ kriz/cifar.html
[16] https://www.image-net.org/