Asynchronous Training of Deep Learning Models/LLMs

Sarthak Joshi[1]
Supervised by Jens Domke[2]

1 Introduction

In recent years, deep learning and, especially, Large Language Models (LLMs) have emerged as a major breakthrough in various domains. From acting as highly sophisticated chatbots nearly indistinguishable from human speech, to navigating obstacles in real time in self-driving cars, these large-scale models are quickly becoming a part of everyday life. However, to properly function as a productive resource, these models need to be trained with vast quantities of data. While many large datasets are publicly available, training using these is still a massive constraint due to the sheer amount of time they take to be trained on, even with multiple GPUs. Therefore, any method that can reduce the time taken to train a model on these datasets without sacrificing accuracy is desirable. One of the major bottlenecks when training a model at scale is the need to synchronize at every training step, which can add massive overheads, especially with a larger number of batches and the latency involved in multi-node GPU setups.

In this work, we have implemented a Python module that enables asynchronous training of models on large datasets partitioned across GPUs using PyTorch [2]. Our module seamlessly integrates into the training pipeline with minimal modifications to the training code. Through this module, we have attempted a variety of asynchronous training approaches and compared them to synchronous training. Unlike prior works that rely on gradients as the basis for computing all the weight values, even when the gradients can be stale due to asynchronous training, we treat gradients as simple intermediate values to train the weights for each batch, but instead rely on merging the model weights themselves to obtain a good final solution.

2 Background

Training a deep learning model or LLM generally consists of preprocessing a dataset so it can be used by a model and then feeding inputs from that pre-processed dataset into a training loop. A single iteration through this training loop is called an epoch, and it represents one pass through the entire training data across all the processes. Each epoch consists of multiple training steps. In each of these training steps, a batch of inputs from the dataset is used to train the model. The loss computed for each of these inputs is averaged for the batch. Each step consists of a sequence of operations. The first is the forward pass, where a batch of inputs is passed into the model. They pass through a series of layers that the model is composed of and generate an output. The outputs that are generated are then compared against the labels or the expected outputs to compute loss as a measure of the difference between them. Then this loss is fed backwards into the model and is used to compute gradients for each weight, which is a measure of how much that weight needs to be modified by. Based on these gradients, the weights are updated, and that concludes one training step. After this, a new batch of inputs is fed to the model in the next step, and eventually, these weights get to a state where the outputs generated fit well with the expected outputs of the dataset. Larger models like LLMs are able to fit a much larger amount of data, and so they are trained with very large datasets, which makes them give good outputs for a large variety of queries.

However, large datasets require the model to be trained on a much higher number of batches, and this can easily become infeasible for a single GPU. Parallelism is needed at this stage to efficiently train the model in a reasonable amount of time. The standard approach used to train models at scale is the synchronous training approach. Instead of training the entire dataset to a model on a single GPU, the dataset is partitioned across multiple GPUs. Each GPU holds its own copy of the model, and all these models are initialized with the same value. The GPUs are only fed inputs from their locally assigned partitions and generate different outputs based on those inputs. These outputs are used to compute losses that would differ across the GPUs, and these losses are then passed backwards to generate different gradients for the same weight in different GPUs. However, if these gradients were to be applied immediately, the weights of models across the GPUs would diverge into different states. We would essentially obtain different models that are trained to fit their local partitions, but not the entire dataset. Therefore, the gradients computed for all the weights are averaged before applying them. This ensures that all the weights are updated to the same values across all the GPUs, and this parity is maintained at each training step. The gradients computed for a model represent the amount that the model weights need to change to generate the correct outputs for the inputs of the current training step. By averaging the gradients, we essentially combine the impact of each of the inputs across the GPUs into a single model update. This is essentially equivalent to increasing the batch size of the inputs, but as the batch is partitioned across the GPUs, we don't need to bring a large chunk of data into the individual GPU memories.

This data-parallel synchronous training using multiple GPUs is generally conducted using the DistributedDataParallel (DDP) module [1] of the PyTorch library. This module seamlessly integrates itself into the training pipeline by simply wrapping around the model during initialization. It defines functions that hook into the post backward pass phase of the training step. These functions average the gradients before they are used to update the model. Averaging gradients involves performing an Allreduce function across the GPUs, which may even be on different nodes. This allreduce needs to be performed at every training step and involves averaging the same number of elements as the total number of trainable weights in the model (one gradient value for each trainable weight). Therefore, the communication overhead scales up with both the number of GPUs and the model size. To mitigate this overhead, DDP attempts to parallelize computations with communications. It divides the trainable weights into buckets and averages all the gradients corresponding to weights in a bucket immediately after they are all computed, even while the backward pass is still ongoing to compute the other gradients. However, despite this optimization, communication still incurs a heavy cost in parallel training.

We instead propose an asynchronous approach to training the models, which removes the need for communication at each step altogether. There have been prior works that have attempted to train a model asynchronously [4] [8] [11], but they generally rely on maintaining a main model that is updated with the gradients (or an average of the currently available gradients) generated from local models, which are updated

[1] *Department of Computational and Data Sciences, Indian Institute of Science*
[2] *Supercomputing Performance Research Team, RIKEN Center for Computational Science.*

```
model = Model()
optimizer = Optimizer()
train_loader = DataLoader(dataset)

model = model.to(device)

for epoch in range(num_epochs):
  for (input, label) in train_loader:
    input = input.to(device)
    label = label.to(device)
    optimizer.grad_zero()
    output = model(input)
    loss = LossFunction(output,label)
    loss.backward()
    optimizer.step()
```

(a) Baseline PyTorch Training Loop

```
init_process_group(backend='nccl')

model = Model()
optimizer = Optimizer()
train_loader = DataLoader(dataset)

model = DDP(model).to(device)

for epoch in range(num_epochs):
  for (input, label) in train_loader:
    input = input.to(device)
    label = label.to(device)
    optimizer.grad_zero()
    output = model(input)
    loss = LossFunction(output,label)
    loss.backward()
    optimizer.step()

destroy_process_group()
```

(b) Synchronous Training with DDP

```
init_process_group(backend='nccl')

model = Model()
optimizer = Optimizer()
train_loader = DataLoader(dataset)

model,train_loader = DDAP(model,optimizer,
                          steps_per_epoch,
                          num_epochs,
                          train_loader).to(device)

for epoch in range(num_epochs):
  for (input, label) in train_loader:
    input = input.to(device)
    label = label.to(device)
    optimizer.grad_zero()
    output = model(input)
    loss = LossFunction(output, label)
    loss.backward()
    optimizer.step()

destroy_process_group()
```

(c) Asynchronous Training with DDAP

Fig. 1: Comparing code changes when using DDP and DDAP

asynchronously on other GPUs. Periodically, the local models are overridden by the weights of the main model to keep them updated. As these methods are still reliant on gradients, they often suffer from the problem of stale gradients, where the gradients used to update the main model are older, resulting in inefficient weight updates and slower convergence of the model. These works have to come up with other mechanisms to mitigate the impact of these stale gradients.

In our implementation, we do not rely on averaging gradients but instead merge the weights. Therefore, we do not deal with stale gradients. There are prior works that introduce model merging techniques [5] [10] [12], which we have used in our asynchronous training mechanism, but these techniques are largely designed with the idea of fine-tuning a pre-trained model on different datasets or using different hyperparameters and then merging them to get a better model. However, we argue that, on principle, if these methods can merge models trained on different datasets and retain the knowledge gained from them, they should also be capable of merging models that have been obtained by training the same initial model using different partitions of the same dataset.

3 Implementation

We have implemented a Python module called DistributedDataAsynchronousParallel (DDAP). Much like DDP, this module can be wrapped around any Pytorch model with minimal code changes. It needs some additional inputs, such as the optimizer, the number of training steps per epoch, the number of epochs, and optionally, the Dataloader corresponding to the training dataset, which can all be trivially provided. Figure 1 shows how any generic PyTorch training code can be easily converted to use DDAP just like DDP. The module automatically performs any communications or other operations needed for asynchronous training in the background without any modifications from the user's end. When the Dataloader corresponding to the training dataset is not provided, the module hooks into the post optimizer step phase of the training step to perform any communications if needed. Wrapping around the dataloader corresponding to the training dataset is useful when training using mixed precision with a GradScaler. It allows us to ensure that the weights are merged at the end of the iteration rather than before the scaler update, which we found to be slightly better performing. Overall, DDAP is a pretty minimalistic wrapper that has a negligible impact on the overall execution. As we perform all our communications at the end of the training step, after the weights have been updated, we do not provide the same computation-communication overlap that DDP does, as it communicates during the backward pass. However, that is more than compensated for by the extremely low frequency of communications in our setup. Using this module wrapper as the basis to perform operations in the background, we have attempted a variety of methods to train a model asynchronously.

3.1 Simple Averaging

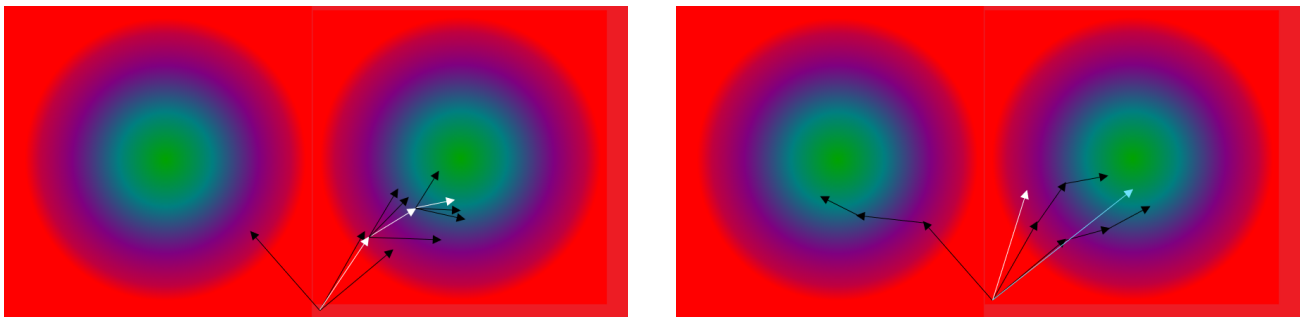
The simplest solution can sometimes give the best results with minimal extra work. For this, we implemented a mechanism to average the weights of all the diverged models in the different GPUs at the end of an epoch. This is essentially equivalent to the operation done in Federated Learning, but instead of performing a weighted average based on the number of batches at different edge devices, we are performing a simple average on a set of GPUs in the same location, since the batches are evenly assigned to all the GPUs. Since it is equivalent to federated learning, we know that this will eventually converge, but the question is whether the convergence occurs faster than the synchronous case.

3.2 Model Shifting

One of the reasons for simple averaging not to converge quickly could be when the models in different GPUs overfit to their locally assigned data. When averaged, these model weights would arrive at some middle point of all the diverged models, which could possibly be good for none of the data partitions. One approach we attempted to address this was shifting the models into different GPUs (a ring shift) for a few epochs before averaging the weights. The models, being stored in the GPU memory, can be easily transferred using GPU-GPU communication links. When the model is moved to a different GPU, it will now be trained using the dataset partition assigned to that GPU. Therefore, the model can be trained on a different chunk of data without needing to shuffle the dataset, which can be a more expensive operation for larger datasets. When multiple models have been trained on multiple dataset partitions with a large enough overlap, they may produce results closer to the correct solution when averaged. However, this approach results in a single merged model not emerging until multiple epochs, which could cause slower convergence if the model takes a low number of epochs to converge in the other cases. Therefore, it needs to be paired with averaging or merging at least every other epoch to be feasible.

3.3 TIES Merge

Another issue with the model shifting approach can be seen when looking at a model as a multi-dimensional tensor of a very large number of dimensions, with each trainable weight being represented by a dimension. In such a multi-dimensional space, the tensor moves towards some solution represented by a point in that space that generates low loss. However, when



(a) Gradients average at each step in synchronous training

(b) Merging models after some iterations using average (white) and TIES merge (blue)

Fig. 2: Model updates in synchronous vs. asynchronous training when there are two equally good solutions

we are in a space with hundreds of millions of dimensions (for large models like LLMs), many equally good solutions can exist very far apart from each other as isolated low-loss valleys that are equally deep. When the models train while being synchronized at each step, they would compute different gradients that would represent taking a small step towards different valleys, but then the gradients would be averaged, and the step taken would bring the model closer to one of many low-loss valleys (solutions). Similar to gravity, this model is now more likely to get closer to that solution in the next step, as the closest solution would generally pull the weights towards it more. However, when training asynchronously, the models actually take multiple steps towards completely different low-loss valleys. They get further pulled into the valleys they stepped towards as they keep diverging. This is depicted in Figure 2. This can even persist when they are given inputs from a different partition, as the low-loss valleys here represent equally good solutions for low loss on the entire dataset, which can exist. Therefore, simply changing the order in which a model is trained using different partitions of the training dataset is enough to completely change the solution the model moves towards. This kind of divergence would result in a bad average outcome even when all the models have been trained on all the dataset partitions, just in a different order. The outcome would be bad even when each of these individual models is relatively good and produces low losses for the entire dataset. In scenarios like these, a solution can be to pick a group of models rather than averaging everything.

The essence of this idea is incorporated in the TIES (TrIm, Elect Sign and Merge) model merging method [12]. This model merging approach was initially defined to merge multiple models generated by fine-tuning a pre-trained model on different datasets. It also looks at a model as a multi-dimensional tensor and first computes the offset of the model from the initial state for all the models being merged. This offset is then summed up and used to decide the direction of the final computed offset. Along each dimension, only the tensors that match the direction of the sum of offsets along that dimension are used to compute the final offset. Therefore, for the dimensions in which all the offsets are along the same direction, this will be the same as averaging. On the other hand, for the dimensions where the offsets diverge into different directions, only the tensors that match the direction of the sum of offsets will be used for averaging. Once the final offset is calculated in this manner, it is added to the initial state of the model to generate a merged model that is the same across all the GPUs. A simplified example of this method in two dimensions is depicted in Figure 2, where we can see that TIES merge results in the final model ending up in a region of lower loss as it only uses two out of three of the diverged models to compute its offset for the dimension represented by the x-axis. This method also involves trimming the low magnitude offsets by setting them to zero, but we found a slightly better performance by not performing the trim. This is likely because, when training models from scratch, small updates to the weights can have a much greater impact as compared to when fine-tuning a pre-trained model. Another factor to consider when merging models is the optimizer state. Optimizers like Adam maintain one or more "momentum" metrics. These prevent new gradients from wildly swinging the direction of the training. If we simply merge the models, the local optimizers may still have a divergence in their momentum metrics, which would cause the models to immediately diverge in the next training step. To mitigate this, we also merge these momentum metrics of the optimizer using TIES merge.

3.4 Other Approaches

Along with these, we also experimented with a variety of other approaches involving various combinations of the above. We also experimented with pairwise model shifting instead of ring shift, model shifting based on the models with the greatest amount of difference from each other, electing direction in TIES merge based on consensus instead of the average of offsets, and electing direction in TIES merge based on the accumulated gradient values. We also experimented with training the model synchronously for a limited portion of the training at the beginning, the end, or both, or in smaller segments across the executions. In general, all of these approaches gave comparable or worse results than the prior three, even at smaller scales.

4 Experimental Results

We have conducted a series of experiments measuring the accuracy of a model against the time taken to train. All cases have been trained for 6 epochs using 7 GPUs in a single node. All of these experiments use RTX A6000 GPUs except for the last case, which trains on 3% of the C4 dataset, where A100 GPUs are used. We have used the BERT base model and the AdamW optimizer for all our experiments, and also used mixed-precision training for faster training times in both synchronous and asynchronous cases. We first present the results when fine-tuning a pre-trained BERT model.

Figure 3 shows the results of training the model on the Stanford Sentiment Treebank (SST2) dataset [6] from the General Language Understanding Evaluation (GLUE) benchmark [7]. This dataset classifies the sentiment of a given sentence as positive or negative. In Figure 3, the x-axis represents the

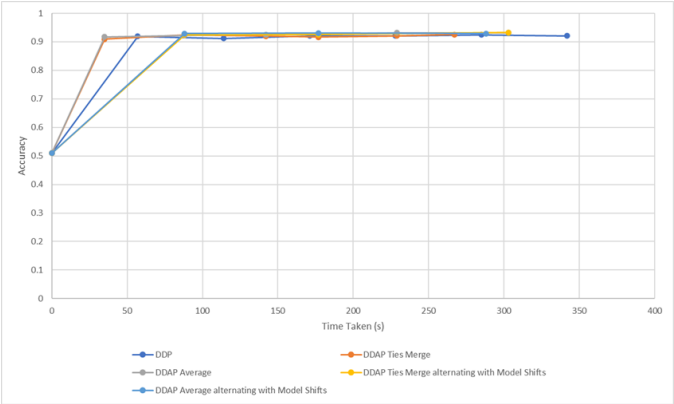


Fig. 3: Comparing DDP and DDAP for the SST2 dataset

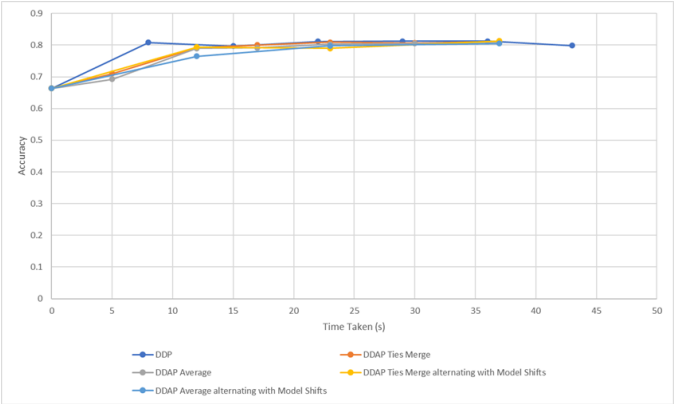


Fig. 4: Comparing DDP and DDAP for the COLA dataset

time taken and the y-axis represents the accuracy achieved. The vertices in the line represent the points at which an epoch of training is finished. Initially, the pre-trained model yields an accuracy of around 50% without any fine-tuning (at epoch 0). We can see that the DDP case takes the longest time to finish the 6 epochs for all of the results. However, what truly matters in runs like these is the point where the model reaches a convergence point, beyond which further training does not give meaningful improvements. Here, we can see that the convergence is achieved in a single epoch, but as the DDAP executions have a lower time taken for one epoch of training, they achieve the convergence accuracy sooner. Using model shifting, alternatively with model merging every epoch, also achieves the same accuracy eventually, but as the models are not merged until the second epoch, we present the accuracy values considering only the accuracies at the 2nd, 4th, and 6th epochs when the models are merged, and the same across all GPUs. Due to this delay in merging, this case falls behind simply merging at the end of the epoch and synchronous training.

Figure 4 presents the same results for fine-tuning on the Corpus of Linguistic Acceptability (COLA) dataset [9] from the same GLUE benchmark. This dataset classifies English sentences into being grammatically correct or not. This is a relatively small dataset and thus, only takes a few seconds to train using multiple GPUs. Here, we observe a scenario where the DDP case cleanly outperforms all the others by again converging in a single epoch, while the asynchronous approaches take 2 epochs, making them fall behind even though they have a slightly faster per-epoch runtime. As this dataset is relatively small, using synchronous training is better here, as there is a lower chance of learning common patterns from different partitions when training asynchronously, which leads to a greater divergence quickly.

The true value of saving training time is realized when training on a large dataset. To properly judge the feasibility of asynchronous training, we have used the Colossal Clean Crawled Corpus (C4) dataset [3]. This is a massive 305 GB dataset. C4 is a large repository of web crawl data containing large chunks of text scraped from many websites. We have trained BERT on it using Masked Language Modelling (MLM). In this approach, some of the words in the sentences get replaced by MASK tokens randomly, and the label is set to the original word. The training objective is for the model to be able to identify which out of the over 30000 tokens in its vocabulary would best fill in the blank represented by this mask. Due to the size of this dataset, it is not feasible to train

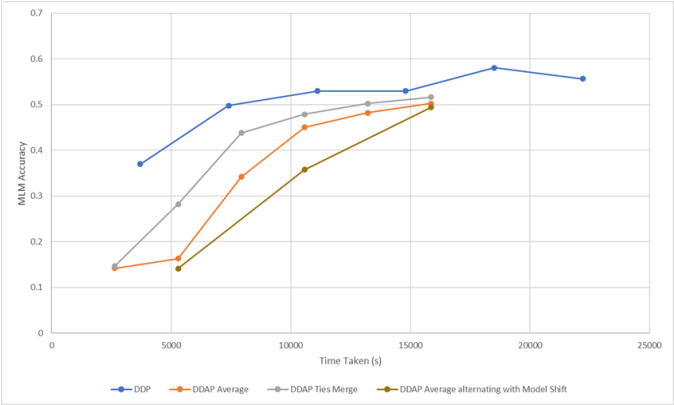


Fig. 5: Comparing DDP and DDAP with baseline approaches

it with only 7 GPUs, so we have instead used a 0.5% partition of the dataset for training. This is still a pretty large dataset, which takes over an hour per epoch for the synchronous case to train on. Figure 5 depicts the comparisons of DDP and DDAP with the baseline approaches. Here, we observe that synchronous training completely outperforms asynchronous training, achieving a higher convergent MLM accuracy and also reaching it faster, even with a noticeably greater time per epoch. We also observe that TIES merging neatly outperforms simple averaging here. On the other hand, alternating model shifting with averaging gives the worst results here, as it consistently falls behind the other cases.

From this, we can identify that the models are diverging too much due to much higher training steps per epoch (16289 steps per epoch), which leads to slower convergence with asynchronous training. Therefore, one way to resolve this is to more frequently merge the models. Figure 6 shows the results when the models are merged at every 10%, 1%, and 0.1% completion thresholds of the training epochs. Here, we find that for larger datasets like these with lots of training steps per epoch, we can merge pretty aggressively with a very low overhead as compared to the synchronous case. Even when merging at every 0.1% completion threshold, it is still one synchronization every 16 training steps, which only adds a small amount of overhead. We can observe that when merging at 1% or lower completion thresholds, the MLM accuracy obtained becomes pretty comparable (sometimes even higher) to the synchronous case. Furthermore, due to the time per epoch being lower when training asynchronously, the asynchronous cases achieve those accuracy values in a lower time interval, leading to faster convergence. Therefore, by training a model asynchronously,

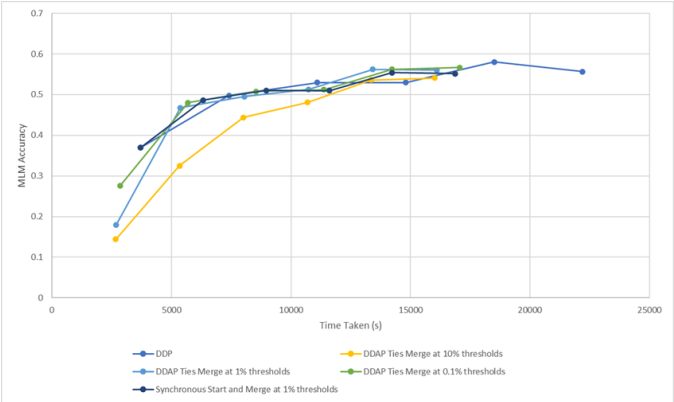


Fig. 6: Comparing DDP and DDAP with more aggressive synchronization

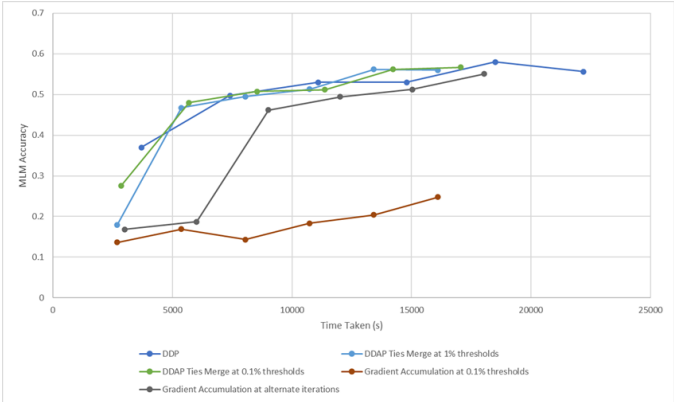


Fig. 7: Comparing DDAP and DDP with Gradient Accumulation

one can obtain similar or even higher accuracy values than the synchronous case in a lower amount of time. The primary point where the synchronous case holds an advantage is for the very first epoch, where it achieves a higher accuracy than the rest. However, for this larger dataset, a single epoch is not enough to reach convergence accuracy. However, if it is still a point of concern, we can simply perform a synchronous first epoch followed by asynchronous epochs. In Figure 6, performing a synchronous start for the first epoch followed by TIES merge at 1% completion thresholds for the following epochs leads to the total time taken being comparable to merging at 0.1% completion threshold, which is still significantly less than the synchronous case. This boosts the accuracy of the first epoch to be equivalent to the synchronous case, while the following epochs run faster while still keeping up with the accuracy of the synchronous case.

Since we are now synchronizing multiple times within the same epoch, one might argue whether our implementation is the equivalent of gradient accumulation. Gradient accumulation involves accumulating a sum of gradients over multiple iterations before actually applying it to the weights. When using DDP, it is possible to not synchronize the model for some training steps by manually disabling synchronization. However, it is the user's responsibility to ensure that the models do not end up diverging before the synchronization happens. Gradient accumulation is one technique that can take advantage of this feature. The accumulated gradients are only averaged in the same training step in which they need to be applied to the weights. This is another means of increasing the effective batch size without increasing the amount of data brought into the GPU memory. However, unlike data parallel training, this approach still computes those batches sequentially, so the only time saved here is due to not synchronizing at every iteration, just like our asynchronous implementation. Therefore, we have compared our implementation to using DDP with gradient accumulation. Figure 7 shows the results of these experiments. We observe that gradient accumulation with DDP, even when synchronizing every alternate training step, still performs noticeably worse than DDAP with aggressive model merging and DDP. Furthermore, this case also exhibits a noticeable increase in the per-epoch runtime as synchronization happens every other iteration. When using gradient accumulation and synchronizing at 0.1% completion thresholds, we obtain a similar per-epoch runtime as DDAP with model merging at 1% completion thresholds. The gradient accumulation case is slightly faster here, as it can take advantage of computation-communication overlap, as it can average gradients while the backward pass is ongoing. However, despite the better per-epoch runtime, the accuracy obtained for this case is significantly worse than all the other cases, and it never even catches up to the convergent accuracy over all 6 epochs. These results clearly show that gradient accumulation is worse than our model merging approach and that these two approaches are not equivalent. The fundamental difference between these two approaches is that while gradient accumulation is the equivalent of increasing the batch size, DDAP is not. In the case of DDAP, the models actually update to a new value and continue refining themselves towards a solution even before they are merged. On the other hand, in gradient accumulation, the model remains static and does not update until a synchronization actually happens. Therefore, it gets fewer opportunities to precisely tune itself to a solution, as the higher batch size causes heavy swings in the training direction. In the DDAP case, the models do get finely tuned, but to different solutions, which causes divergence. However, merging the models by finding commonalities in those different solutions leads to better results than not trying to tune them in the first place.

Finally, we have compared DDP with DDAP when training on a larger 3% partition of the C4 dataset. Unlike the previous experiments, these were performed using A100 GPUs instead of RTX A6000 GPUs for faster training time. Figure 8 shows these results. Here we once again observe comparable accuracies when using DDAP as compared to DDP, but obtained in a shorter amount of time. Furthermore, as the number of training steps per epoch increases further with a larger dataset, we can synchronize more aggressively with relatively lower losses. We observe that performing a TIES merge every 100 iterations when the number of training steps per epoch is greater than 10000 gives slightly better results than merging at completion thresholds with a very small additional overhead. We also observe that as the dataset size increases, the overhead due to synchronization also increases in the DDP case, and therefore, using the asynchronous training approach gives an even greater improvement in training time. This shows the scalability of the asynchronous training approach in terms of the model size.

5 Conclusion and Future Work

In this work, we have implemented a PyTorch module that can be seamlessly integrated into any generic PyTorch training pipeline and enables asynchronous training. We have explored some approaches to merge model weights to obtain an asynchronously trained model rather than by using stale

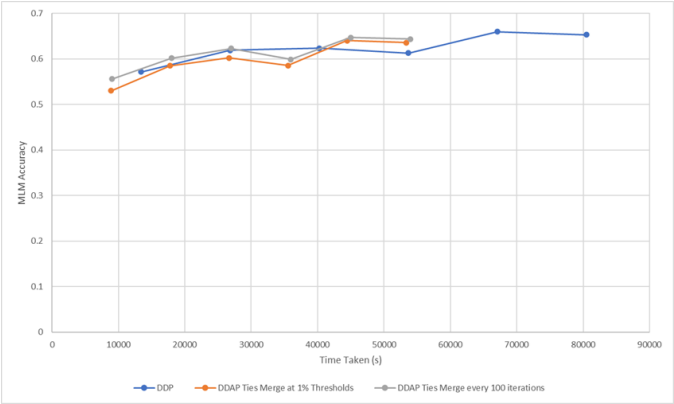


Fig. 8: Comparing DDP and DDAP for a larger dataset

gradients. We have shown that the model merging mechanisms used to merge models fine-tuned on different datasets or hyperparameters can also be used to effectively merge models trained on different partitions of the same dataset, thus enabling asynchronous training. We have shown that by training asynchronously using this approach, one can obtain similar accuracies as synchronous training but at a lower epoch runtime as long as the model merging is performed frequently enough. Furthermore, the frequency of model merging at which the accuracies match up with the synchronous training approach is low enough that the communication overhead due to synchronization is very small as compared to synchronous training. Therefore, asynchronous training of models on large datasets using model merging after periods of asynchronous training is a feasible approach to train a model efficiently.

There are certain experiments we were not able to perform due to the time constraints of this internship and technical difficulties in the systems that can further augment this work. The first would be a comparison of DDP and DDAP in a multi-node setup, with the inter-node communication latency causing further overheads in the allreduce communications. Asynchronous training could show even more advantages in this setup due to its low synchronization overheads. Another approach could be to perform model shifting while not merging the optimizer states when performing multiple model merges in a single epoch. When models are merged before the end of a training epoch, shifting models becomes equivalent to just shuffling the batches in a new order. However, if we choose not to merge the optimizer state instead, a shuffled batch could potentially pull a model to a completely different direction based on the momentum metrics, potentially giving better accuracy. Experiments comparing our asynchronous training approach to the other gradient-based approaches can also be conducted. We could also further experiment with other model merging approaches and try to obtain an even better solution when merging models.

References

[1] Li, S., Zhao, Y., Varma, R., Salpekar, O., Noordhuis, P., Li, T., Paszke, A., Smith, J., Vaughan, B., Damania, P., and Chintala, S. Pytorch distributed: experiences on accelerating data parallel training. *Proc. VLDB Endow. 13*, 12 (Aug. 2020), 3005–3018.

[2] Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Köpf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., Bai, J., and Chintala, S. *PyTorch: an imperative style, high-performance deep learning library*. Curran Associates Inc., Red Hook, NY, USA, 2019.

[3] Raffel, C., Shazeer, N., Roberts, A., Lee, K., Narang, S., Matena, M., Zhou, Y., Li, W., and Liu, P. J. Exploring the limits of transfer learning with a unified text-to-text transformer. *J. Mach. Learn. Res. 21*, 1 (Jan. 2020).

[4] Shanmugaperumal Periasamy, M., Meenakshi Sundaram, B., Gurajala, T., Rai, S., Swaran, A., and Sahoo, A. Efficient and secure deep learning asynchronous training in serverless p2p networks. In *2024 Second International Conference on Emerging Trends in Information Technology and Engineering (ICETITE)* (2024), pp. 1–6.

[5] Shoemake, K. Animating rotation with quaternion curves. *SIGGRAPH Comput. Graph. 19*, 3 (July 1985), 245–254.

[6] Socher, R., Perelygin, A., Wu, J., Chuang, J., Manning, C. D., Ng, A., and Potts, C. Recursive deep models for semantic compositionality over a sentiment treebank. In *Proceedings of EMNLP* (2013), pp. 1631–1642.

[7] Wang, A., Singh, A., Michael, J., Hill, F., Levy, O., and Bowman, S. R. GLUE: A multi-task benchmark and analysis platform for natural language understanding. In the Proceedings of ICLR.

[8] Wang, H., Jiang, Z., Liu, C., Sarkar, S., Jiang, D., and Lee, Y. M. Asynchronous training schemes in distributed learning with time delay. *Transactions on Machine Learning Research* (2024).

[9] Warstadt, A., Singh, A., and Bowman, S. R. Neural network acceptability judgments. *arXiv preprint arXiv:1805.12471* (2018).

[10] Wortsman, M., Ilharco, G., Gadre, S. Y., Roelofs, R., Gontijo-Lopes, R., Morcos, A. S., Namkoong, H., Farhadi, A., Carmon, Y., Kornblith, S., and Schmidt, L. Model soups: averaging weights of multiple fine-tuned models improves accuracy without increasing inference time. In *Proceedings of the 39th International Conference on Machine Learning* (17–23 Jul 2022), K. Chaudhuri, S. Jegelka, L. Song, C. Szepesvari, G. Niu, and S. Sabato, Eds., vol. 162 of *Proceedings of Machine Learning Research*, PMLR, pp. 23965–23998.

[11] Wu, X., Rao, J., and Chen, W. Atom: Asynchronous training of massive models for deep learning in a decentralized environment, 2024.

[12] Yadav, P., Tam, D., Choshen, L., Raffel, C., and Bansal, M. TIES-merging: Resolving interference when merging models. In *Thirty-seventh Conference on Neural Information Processing Systems* (2023).