

Distributed Neural Networks Final Report

Overview

Importance

With the advent of ChatGPT, our world has become increasingly saturated with AI systems. Modern code editors utilize large language models to autocomplete code segments and speed up developers. Phones are now increasingly incorporating AI features like automatic photo enhancements. Even the most popular search engines like Google and Bing now have AI features built directly into them. In short, our world is becoming more AI-dependent by the day.

At the backbone of all this are the companies producing and selling these models to consumers and corporations. Companies like OpenAI and Anthropic train the models and house the infrastructure needed to utilize them, which are then licensed or directly integrated into the products we use. While early versions of these models were on the scale of millions of parameters, modern models use trillions of parameters, making them much more sophisticated, but at the same time more costly to train and inference. As a result, developing ways of leveraging large computing infrastructure has been essential in the widespread adoption of AI. This report will analyze and compare 2 popular libraries for distributed training: PyTorch DDP and DeepSpeed.

Background information

Due to the immense size of these models, it has become infeasible to train them on a single GPU.

As a result, these models utilize parallel training techniques to massively decrease the overall training time and memory needed for training. There are 2 main parallel training methods.

The first is called model parallelism. Model parallelism takes a model and splits it vertically or horizontally across many GPUs. Some references refer to the horizontal splitting as pipeline parallelism, as you can use pipelining to make inferences on many inputs, which can shorten the overall runtime. Regardless, the central idea is to reduce the overall memory footprint of the model by dividing it among several GPUs. Communication is then used as needed to move data between the different GPUs.

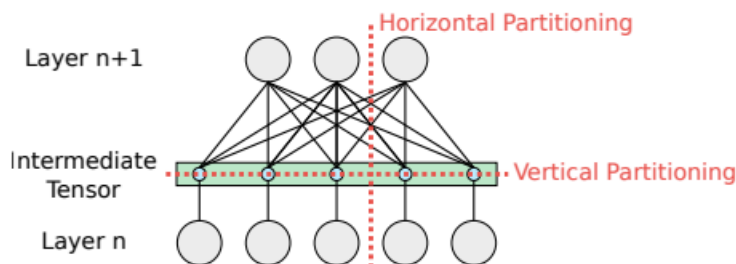


Figure 1: Model Parallelism Example

As stated, this can reduce the memory needed to train the model at the cost of increased communication overhead. This approach isn't preferred for modern AI systems, though, because of the communication overhead. Instead, the preferred approach is to use another parallel training technique: data parallelism.

While model parallelism distributes a model onto many GPUs, data parallelism instead duplicates the model onto many GPUs. At the cost of using more memory, this enables the model to more easily scale modern training by enabling entire batches of training data to be spread across the GPUs with very little communication required to facilitate training. In this situation, you only need to communicate gradients between the batches. This is very beneficial as modern AI models not only have trillions of parameters, making each training example difficult to process, but they also have enormous datasets, making training time for the dataset very costly. Due to these savings, this has become the primary method for modern AI training.

Underpinning both methodologies is the way communication is handled between the GPUs. Having efficient communication is essential to ensure training is done quickly. Many libraries have been developed for this task, but a few of the most popular are NVIDIA's NCCL, Facebook's Gloo, and MPI. Each of these libraries implement a few key communication strategies. The most important strategy is the AllReduce operation. AllReduce takes data from every GPU and performs some type of operation on it, then distributes the result back to all the GPUs.

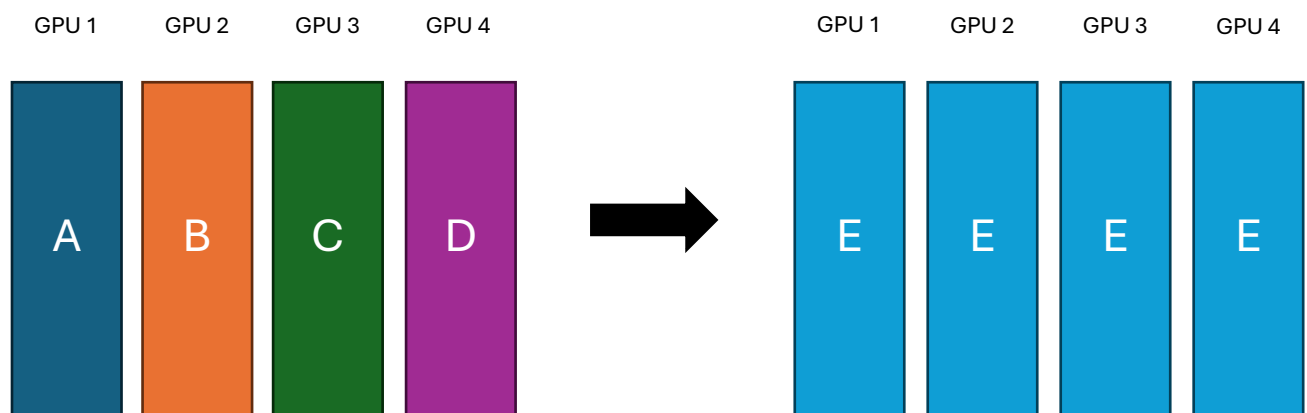


Figure 2: AllReduce Example

This operation is important as much of the communication required for data parallelism is to communicate gradients and sum them together. There are also various ways this can be done, each of which influences the memory used. The most common are Ring AllReduce and Tree AllReduce. Other important operations include AllGather, which essentially spreads a combined array to all GPUs, each of which contributes an even part of the data to the final array, and scatter, which takes an array of data housed on one GPU and evenly partitions it across many GPUs. There are also variants that only spread data to one GPU for AllReduce and AllGather called just Reduce and Gather, respectively. All these strategies can be implemented in a variety of ways to reduce the memory footprint on the systems as well.

PyTorch DDP Background

Built on top of PyTorch, PyTorch DDP is an easy way to scale single-threaded single-GPU models onto many GPUs with little code modifications. At its core, PyTorch DDP training is the same as training a normal PyTorch model. It is done in 3 stages. The first is the forward pass, which takes training data and computes a number of loss tensors to guide the model's second step: the backward pass. As stated, the backwards pass guides how the model's parameters will be updated by computing their gradient. Finally, the optimization stage actually takes the data computed in the backwards pass and updates the model's parameters.

Within the PyTorch ecosystem, there are 3 communication libraries that are offered. PyTorch Data Parallel is a multi-threaded approach to train on multiple GPU's but is limited to a single process. As a result, it can't scale to many machines. Another is PyTorch RPC, which is used for more advanced training scenarios. Finally, there is PyTorch Distributed Data Parallel (PyTorch DDP), which is like PyTorch Data Parallel but is a multi-process library and can scale to multiple machines or nodes. We will be doing our analysis on PyTorch DDP.

Even within the realm of distributed training, there are different ways to train the model itself. The first is called parameter averaging, which, instead of passing reduced gradients to the optimizer stage, you take the results from the optimizer stage and average those results together. This has a few issues, though. The first is that this isn't mathematically equivalent to the single-threaded approach we discussed earlier. The second is that since we are reducing at the end of the

training step, we must have the updated parameters before we can proceed with training. As a result, communication from reducing the resulting parameters together cannot overlap with any of the computations. As a result, PyTorch DDP does not use parameter averaging. Instead, it reduces during the backwards pass. (We will discuss why and how we reduce during the backwards pass instead of after the backwards pass later.)

To parallelize this naively, we can simply replicate the model onto many machines and split our training data between these replicas. Then we can compute the forward and backward passes for each of the datasets passed to the models, then AllReduce the results together, then pass that result to the optimizer step. Then we repeat this process for many epochs until our model is fully trained. While this approach works, it can be improved in order to reduce the communication overhead required. PyTorch DDP aims to provide these optimizations in order to produce better training results without forcing users to do the fine-grained optimizations themselves.

PyTorch DDP starts by creating hooks in the model. These hooks are utilized throughout training in order to facilitate distributed training. When the backward pass runs, these hooks fire and facilitate AllReduce operations to synchronize gradients together. As stated earlier, we synchronize during the backwards pass. This is done through a strategy called gradient bucketing. Instead of simply doing one big AllReduce at the end of the backward pass, we can do many smaller AllReduce operations during the backwards pass because once the gradients are computed, they are not needed again since we only need the reduced result for the optimizer stage. As a result, when one gradient bucket finishes, it fires the AllReduce and continues

processing the other gradients. At the end, once all gradients are reduced, it proceeds to the optimizer stage.

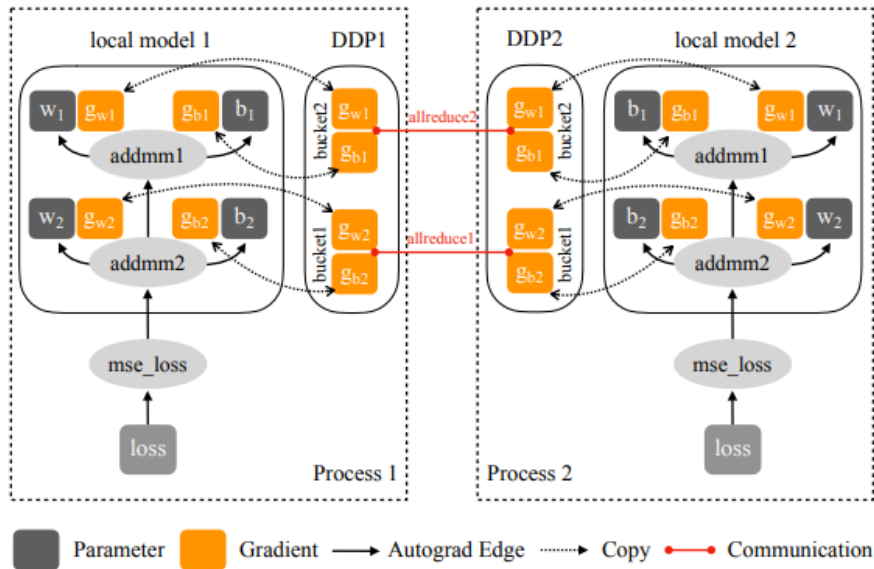


Figure 3: PyTorch DDP Gradient Bucketing

For this strategy to work, PyTorch must ensure a few things. Some models do not require every parameter for every pass. As a result, if a bucket contains a parameter that will never be marked as completed in a pass, it will hang as it waits for a gradient to be computed that never will be. To ensure that this does not happen, PyTorch DDP marks those parameters as ready during the forward pass. The other issue is that some buckets may complete in different orders if one isn't specified. As a result, PyTorch DDP enforces an order on the gradient buckets that are computed, which is the reverse order of the parameters since this is usually the order they will be computed in during the backwards pass.

DeepSpeed Background

The other system we will be looking at is DeepSpeed. DeepSpeed is a lightweight PyTorch wrapper that provides an easy way for users to implement state-of-the-art training techniques like mixed precision training, distributed training, and many more. As a result, users are more easily able to train larger models in more advanced ways over other libraries like PyTorch DDP.

One of the biggest contributions DeepSpeed has made has been the introduction of ZeRO (Zero Redundancy Optimizer) which is responsible for improving the memory efficiency of AI models during distributed training. At a high level, ZeRO vertically partitions the model states onto many machines, but training functions like traditional data parallelism. The main difference between it and traditional model parallelism is that ZeRO can process complete inputs while traditional vertical model parallelism can only process corresponding input partitions. As a result, you get the memory savings as model parallelism but with the benefit of being able to train your model as if it were using data parallelism.

As we have discussed, these models have continued to get larger over time. Modern models use trillions of parameters, and at that scale there isn't any room for models to waste memory. For example, GPT-2 has 1.5 billion parameters. If these parameters use 16-bit floating-point precision, the parameters alone need 3 GB of memory. The gradients will also need to be calculated for each parameter, so you need an additional 3 GB. Certain optimizers also need memory per parameter. For example, the ADAM optimizer needs 2 states per parameter for the

momentum and variance data. If you want to use mixed precision training, you need even more memory. For this, your optimizer state memory may go from the initial 16-bit data to 32-bit data, causing you to need an additional 6 GB of memory. Furthermore, you need another copy of your parameter data in 32-bit precision to fix issues with underflowing values in 16-bit precision, so you require an additional 6 GB. Your GPT-2 model has gone from a measly 3 GB to 24 GB. It gets worse though. During the backwards pass, you need the intermediate results of the forward pass called activations to compute the gradients. That means every intermediate result in your model needs memory allocated to it. This can be alleviated through activation checkpointing where you only store the activations for certain checkpoints during the forward pass and then recompute the remaining ones during the backward pass. For GPT-2 you can get activation memory to about 8 GB without a substantial hit to performance. There are other factors that also cause memory to be used during distributed training as well, but just with these factors we have gone from 3 GB to 32 GB. If we wanted to scale the parameters into trillions of parameters, we are looking at potentially terabytes of memory needed to fit the model onto a GPU. A model of this size isn't feasible to train if we want to use traditional data parallelism. With ZeRO this issue can be greatly reduced. As we said, ZeRO partitions the model states across GPUs to reduce the overall footprint without sacrificing the performance gains of data parallelism.

.

ZeRO is done at 3 stages depending on the needs of the user. The first stage partitions only the optimizer state memory across GPUs. As a result, when we increase the number of GPUs, we see a proportional decrease in optimizer state memory. As a result of this partitioning, parameter updates need to be done only to those parameters that correspond with the GPU that holds those

optimizer states. As a result, we must synchronize the parameters at the end of the optimizer stage through an AllGather operation.

ZeRO Stage 2 includes the partitioning from Stage 1 but also partitions the gradients between the GPUs. This can be done since the optimizer is only updating the parameters corresponding to those optimizer states it houses, so we only need those corresponding reduced gradients for the optimizer to determine how to update the parameters. This still requires communication during the backwards pass to reduce the gradients before the optimizer stage starts, but instead of reducing all gradients to every GPU, we only reduce them to the GPU that houses those parameters making this similar to a ReduceScatter operation. As a result, we are communicating less information overall compared to traditional AllReduce since each GPU only needs part of the final reduction. DeepSpeed also employs a bucketization strategy like PyTorch DDP to help improve this. Like Stage 1, ZeRO Stage 2 also sees a proportional saving in gradient memory as the number of GPUs grows.

ZeRO Stage 3, similar to Stage 2, includes the partitioning scheme of Stage 2, so it partitions optimizer states and gradients among the GPUs. Zero Stage 3 also partitions parameter memory among the GPUs though. As a result, we see a proportional drop in total parameter memory needed; however, this step does require communication whenever a parameter is needed during the forward and backwards pass. ZeRO has other optimizations for activation memory along with other things that cause memory issues in training, but my testing only used these optimizations.

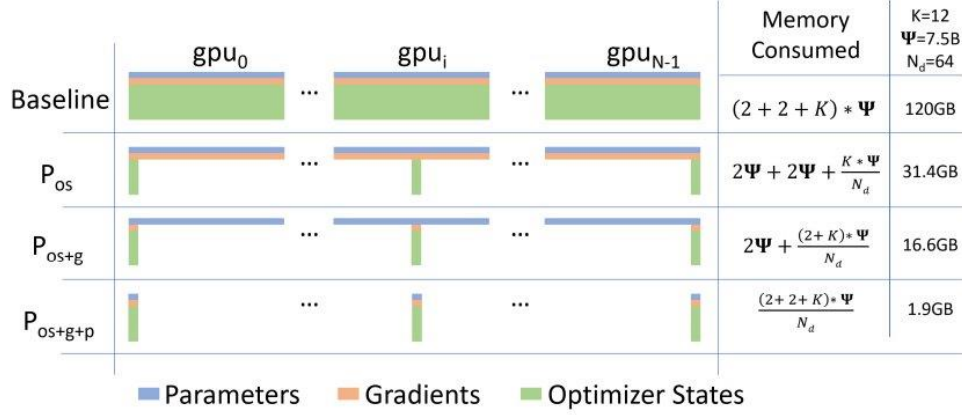


Figure 1: Comparing the per-device memory consumption of model states, with three stages of *ZeRO*-DP optimizations. Ψ denotes model size (number of parameters), K denotes the memory multiplier of optimizer states, and N_d denotes DP degree. In the example, we assume a model size of $\Psi = 7.5B$ and DP of $N_d = 64$ with $K = 12$ based on mixed-precision training with Adam optimizer.

Figure 4: *ZeRO* Memory usage from references

As we have discussed, *ZeRO* can make a significant impact on memory savings, but it does need to utilize communication strategies to effectively facilitate this. If this causes a massive performance burden, then *ZeRO* may not be as effective as normal data parallelism in terms of speed. To see the impact this has, the paper introducing *ZeRO* looks at the volume of data communicated during a training step and compares it against the traditional data parallelism approach. In Ring AllReduce, each GPU essentially reduces a part of the data, then we communicate those results to every other GPU. This causes us to exchange 2 times the total amount of data. Within *ZeRO* Stage 1, we require an AllGather operation to synchronize parameters which are 1 times the parameter size. Within *ZeRO* Stage 2, we also need to do a ReduceScatter to synchronize reduced gradients before the optimizer stage, which also requires essentially 1 times the parameter size to be sent since the gradient memory and parameter

memory are equal. As a result, the total information sent with ZeRO Stage 2 is equal to normal distributed training. Furthermore, with ZeRO Stage 3 we require an additional AllGather to get the parameters for the forward and backward passes, requiring an additional 1 times the parameter data for a total of 3 times the parameter data. As a result, the volume of data sent in ZeRO Stage 3 isn't as detrimental as it might appear. We can also pipeline communication and spread it throughout the forward pass to reduce overall overhead as well.

PyTorch and DeepSpeed Testing

Run Time Data and Analysis

To compare the two systems most effectively, we tried to pick a training situation that would require a non-trivial amount of computing resources to complete but would also not be so expensive that it would require an extreme number of resources. As a result, we settled on training an image recognition model with a large dataset. The model trained is a slight modification to the standard PyTorch ResNet101 model. The only difference is that it uses 4 activation checkpoints to reduce the activation memory as this became a major bottleneck when activation checkpointing was not used. The model has 44.5 million parameters, so it is large enough to see how the two systems would behave. The dataset utilized is the ILSVRC 2012 dataset which has over a million training samples with a validation set of 50,000 samples each of which can be classified into 1 of 1,000 distinct classes. We only used 131,072 images in testing the run times. These were selected using a randomly generated seed value and were fixed on all data collection runs. Furthermore, the dataset images were all transformed prior to training to fit the model input specifications as Resnet models typically work on 3x224x224 tensors. For the data collection, we did a total of 5 runs. 1 control run which utilized a single GPU, 2 PyTorch DDP runs which used 2 and 4 GPUs, and 2 DeepSpeed runs which also used 2 and 4 GPUs. The GPUs were all NVIDIA RTX 3090's. Collection ran for 100 epochs and had a batch size of 64 images per GPU. The DeepSpeed runs were utilizing ZeRO Stage 3. We were unable to control node placement in the environment we were training in as we did not have permission to specify an exact node to run on, so the total allotted CPU memory and the individual node hardware may have added variations that cannot be attributed to the individual systems. Furthermore, since the systems cannot be ported 1 to 1, that may have also been a minor but negligible difference that

can't be attributed to the systems themselves. The training scripts are extremely similar though, so this is negligible.

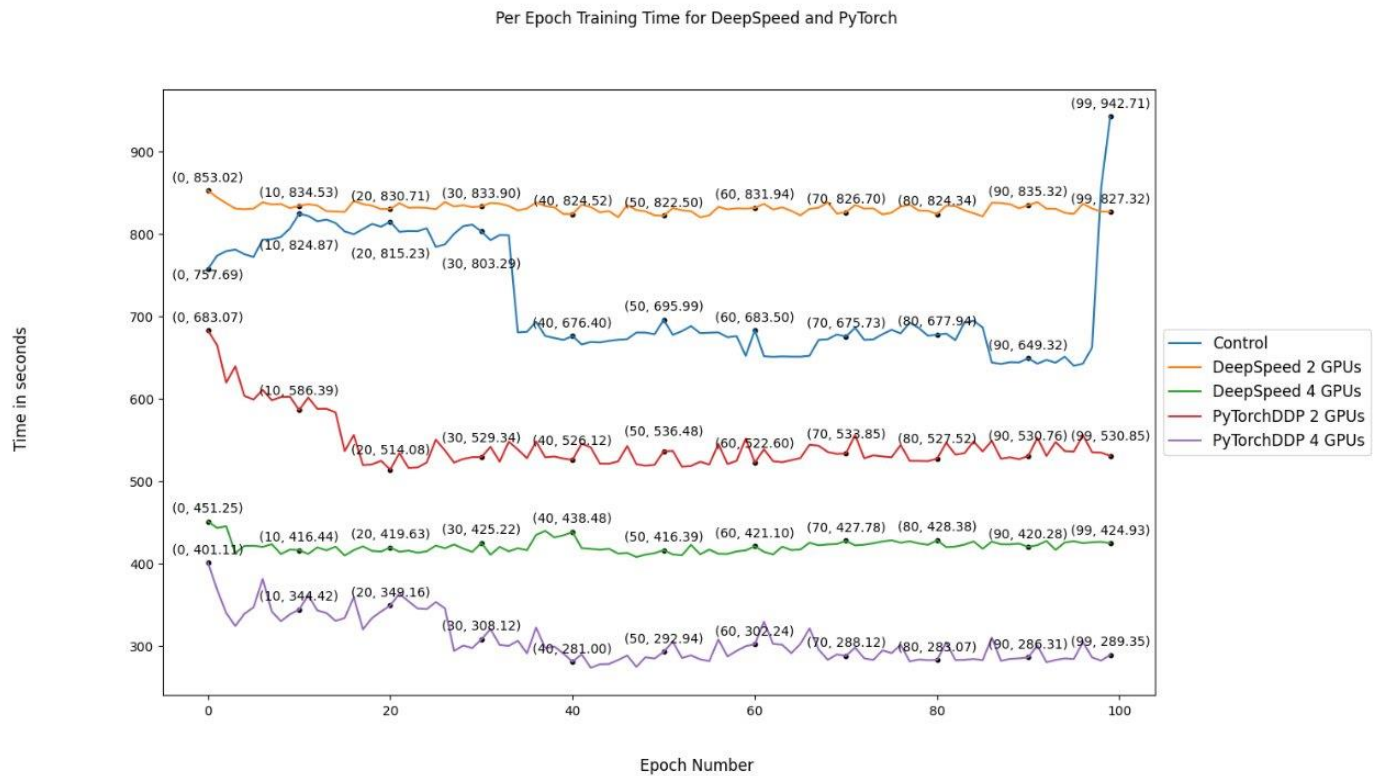


Figure 5: Per Epoch Training time



Figure 6: Per epoch training time running average

The first thing to notice is that PyTorch DDP had by far the best runtime results. We see a clear speedup over both the control and 2 GPU DeepSpeed setup for the 2 GPU PyTorch run.

Furthermore, the 4 GPU PyTorch run is the fastest. Another interesting result is the 2 GPU DeepSpeed result. Surprisingly, it performs worse running roughly 100 seconds longer per epoch when compared to the control. This is likely due to the additional ZeRO communication as the model is likely spending a significant amount of time transmitting and receiving data that is dominating the overall training time which is not made up for by splitting the model states onto only 2 GPUs. This also likely explains why the 4 GPU DeepSpeed run performs so much better, which sees an almost 2x speedup over the 2 GPU example since the model's additional communication overhead is being offset by the increased number of GPUs. There does seem to be a lot of variation in per epoch training time for some of the runs, which could be due to IO contention between the individual job and others on the computing environment so it is hard to say if these results would be as clear cut if they were properly isolated.

Memory Data and Analysis

The way we collected memory data was done almost the exact way we collected data for run time results. The only major difference is that we enabled a memory profiler in the script which causes a massive amount overhead. This was disabled during the run time collections, so it did not affect those results. Additionally, the data collected is quite large, so we scaled the dataset size from 131,072 images to only 1,024 images and decreased the number of epochs from 100 to 10. The profiler also needs to warm up for accurate results, so we only have collection data for 2 of those 10 epochs. These results appear to be consistent from my independent testing though, so we don't think this is a major issue. Finally, we only have the memory data for 1 of the processes that are executed. If a run utilizes 4 GPUs, it will execute 4 processes, but we only collect the data for 1 of those 4 processes. The results should be similar if not the same though since they are running the same training script.

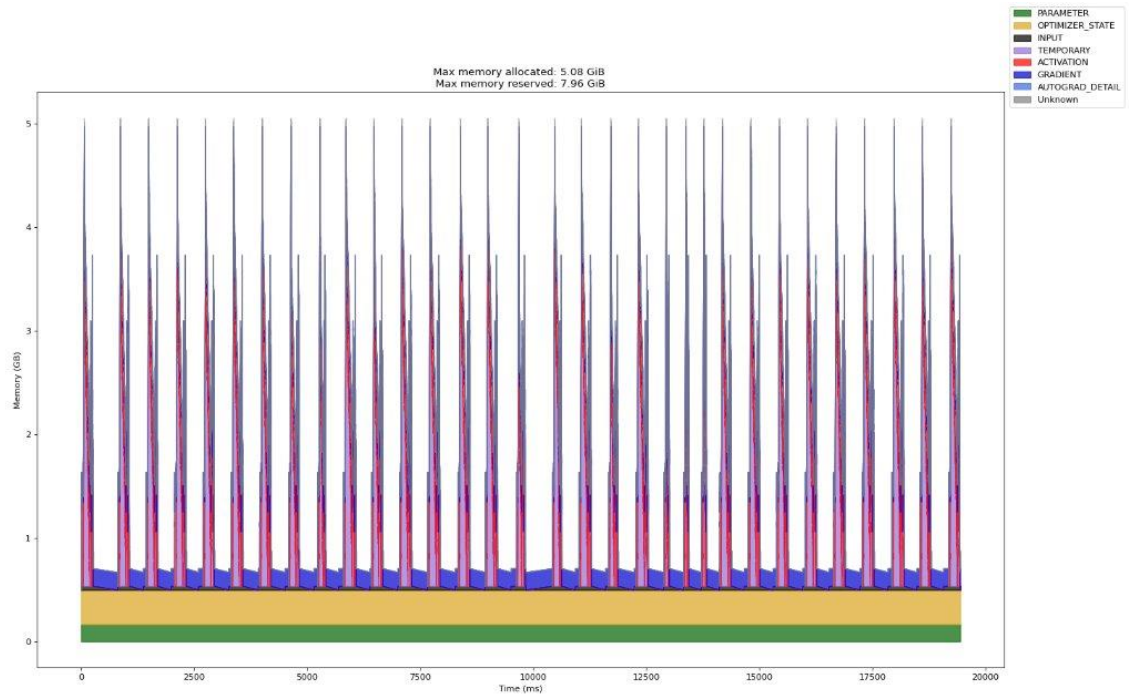


Figure 7: Control Memory usage over time

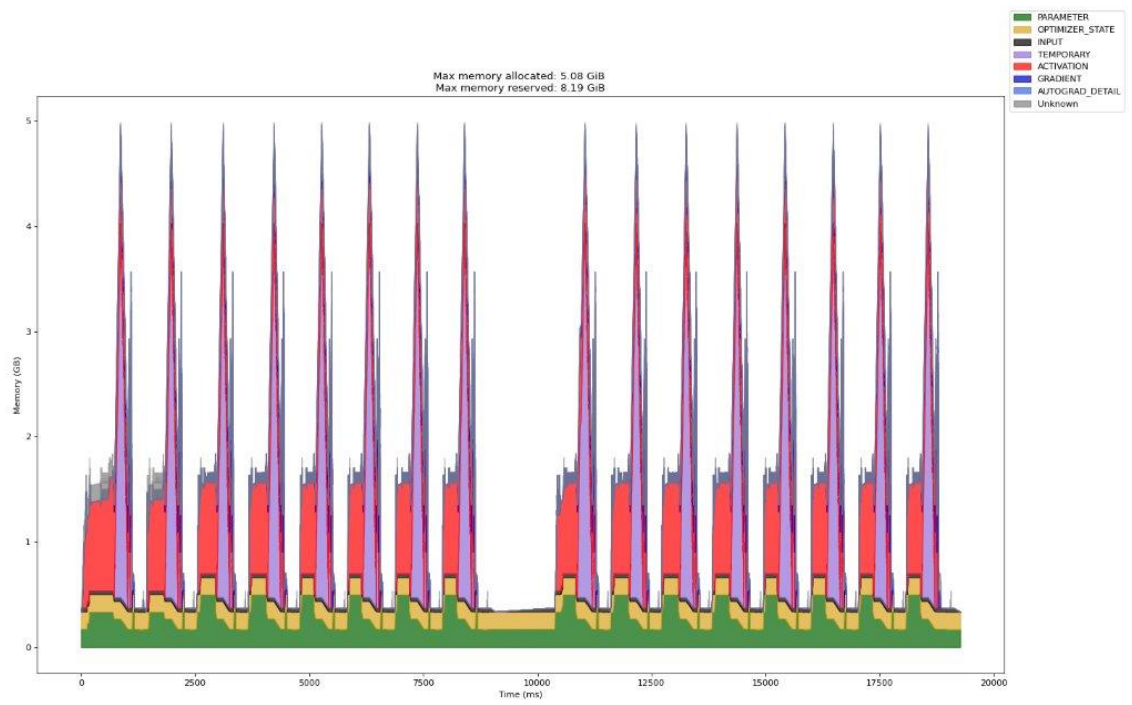


Figure 8: DeepSpeed 2 GPU memory usage over time

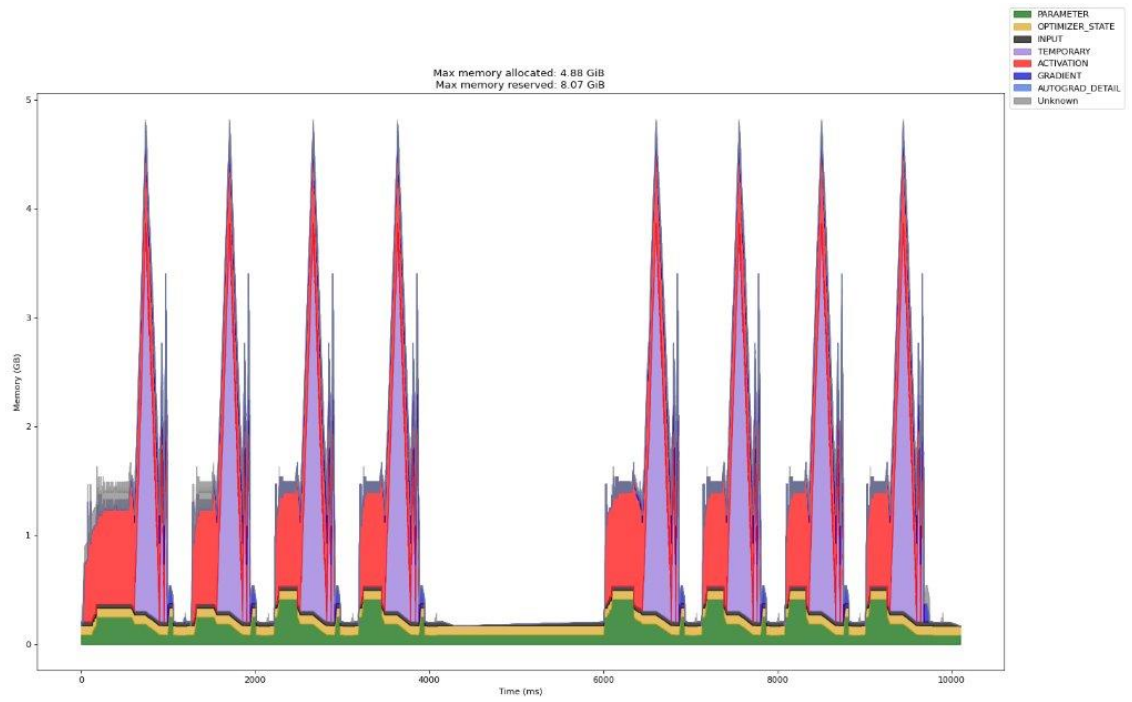


Figure 9: DeepSpeed 4 GPU memory usage over time

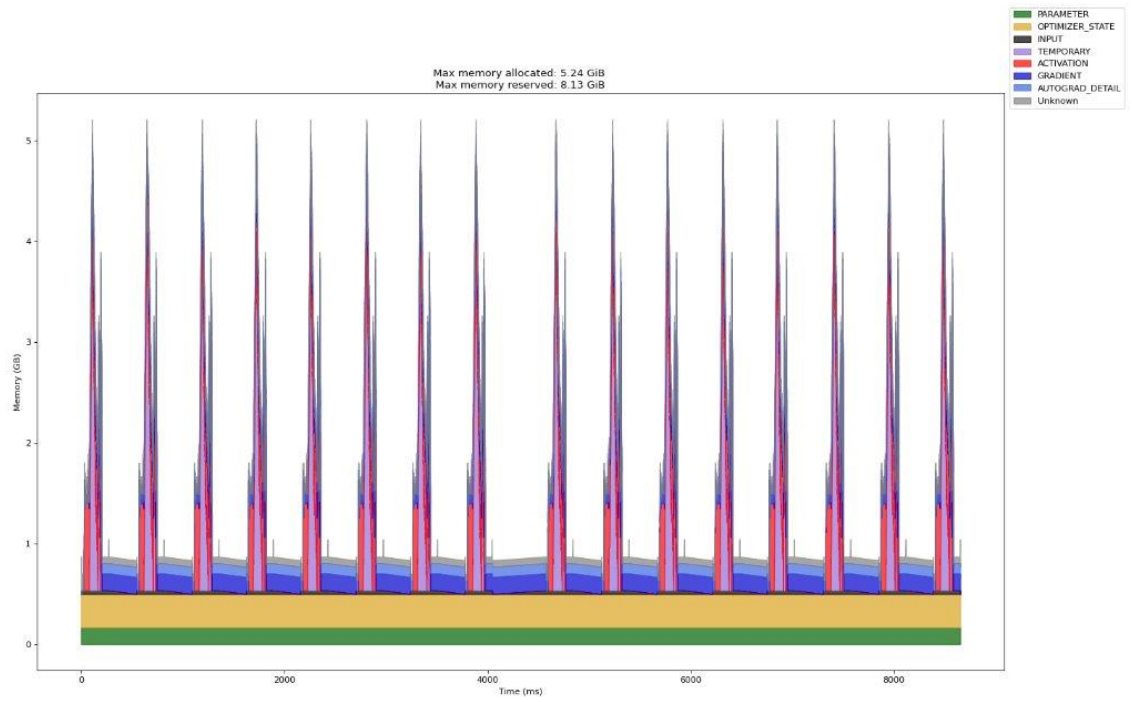


Figure 10: PyTorch DDP 2 GPU memory usage over time

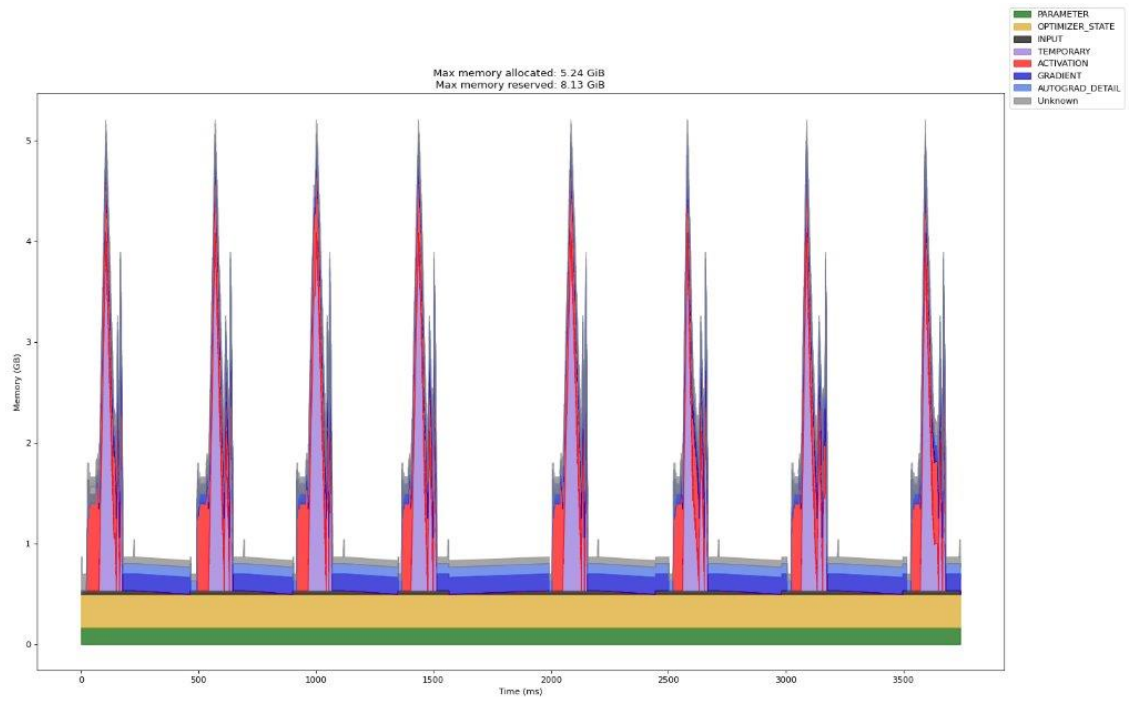


Figure 11: PyTorch DDP 4 GPU memory usage over time

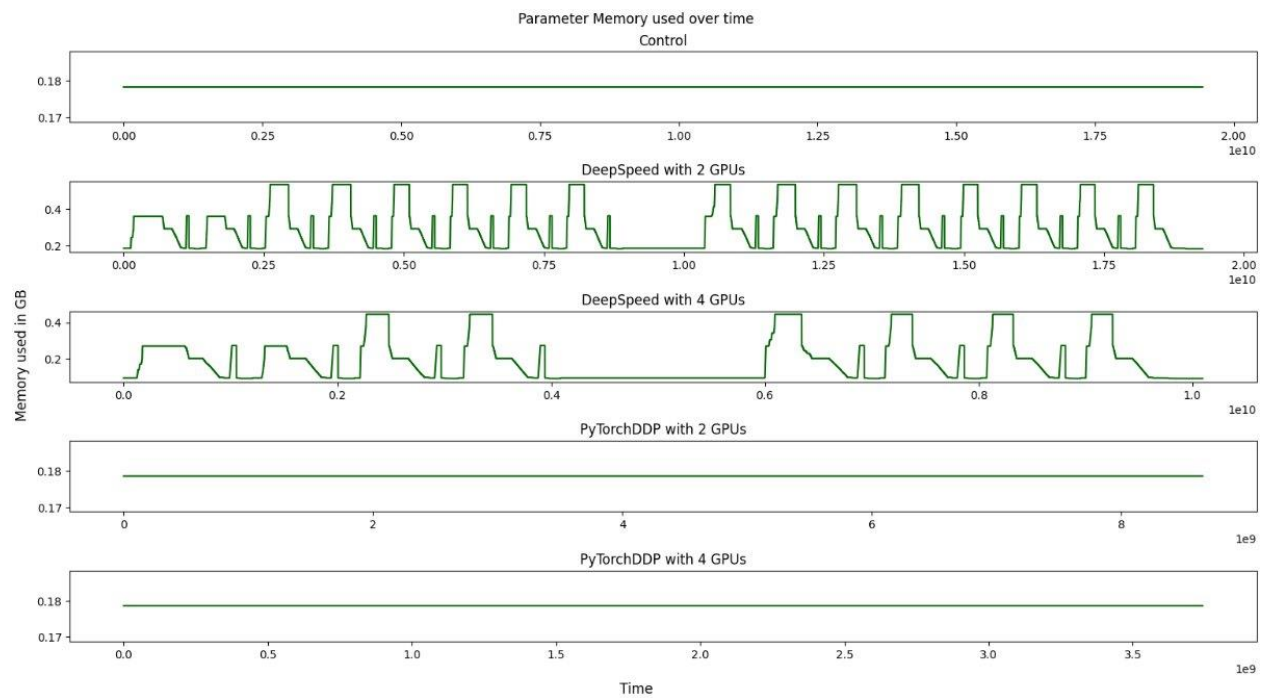


Figure 12: Parameter Memory used over time



Figure 13: Gradient memory used over time

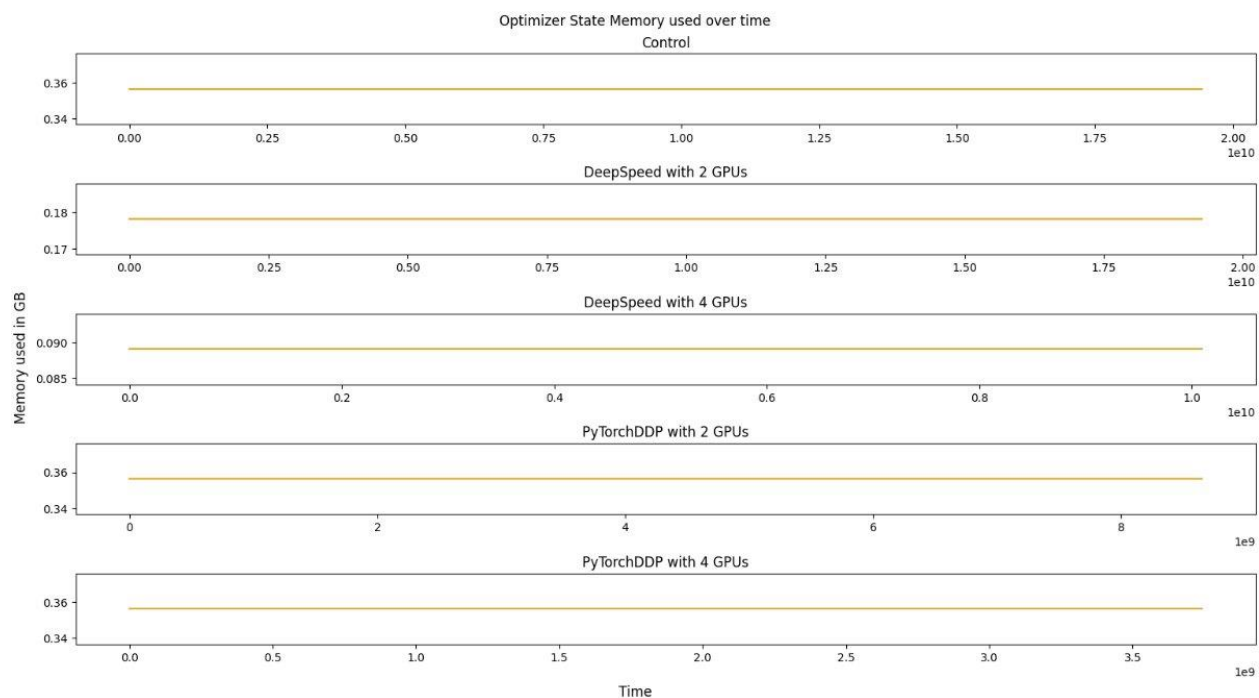


Figure 14: Optimizer state memory used over time

The first thing we should point out is what the memory data is showing. From the aggregated data, we can see clear peaks and valleys on each of the plots. Each peak represents memory used for 1 batch in the epoch. Since each GPU processes an equal portion of the dataset, the more GPUs we have the less batches we need for the epoch which is why the control has twice the number of peaks as the runs with 2 GPUs and those runs have twice the number of peaks as the 4 GPU runs. We could have proportionally decreased the batch size instead of the number of batches, but we chose to do it this way to get a consistent view of how each system handles a batch hence why there is almost no variation in the PyTorch and Control results other than the number of peaks since they are all doing essentially the same thing except a different number of times. The first major result we can see is that DeepSpeed does show a decrease in optimizer state memory. This is unsurprising as ZeRO is evenly distributing this data onto multiple GPUs. The second major result is gradient memory. In the worst case, the memory utilized by DeepSpeed seems to equal the worst case of PyTorch which is quite surprising to us. DeepSpeed does seem to be freeing memory more effectively though, so the overall usage isn't as impactful as it is in the PyTorch runs. The third result we found surprising was the parameter memory. DeepSpeed seems to be using almost twice the PyTorch and control results for Parameter memory. This may be a configuration issue, or it may just be defaulting the weights as 32-bit floats instead of 16-bit floats. It does seem to be using this memory effectively by freeing and discarding parameter memory before memory becomes the scarcest. Overall, DeepSpeed uses memory more efficiently. It would be interesting to see how this would scale to a bigger model with more parameters. From my testing, all other memory sources seem to be the same between the two systems.

Conclusion

Between the two systems, they both perform similarly. Overall, PyTorch DDP performed much better given similar memory usage and superior run time results, but it is hard to say if that would be the case if we scaled the model to more parameters. Because of this, we think that the results are unfairly prejudicial against DeepSpeed. In terms of configuring and setting them up, PyTorch DDP was significantly easier to set up. Setting up DeepSpeed may have been difficult because of our lack of experience working with the computing environment, and once it was installed porting the PyTorch model to DeepSpeed was very easy. The one major advantage of DeepSpeed is the ease of adding more complex training features. To enable ZeRO for instance, all you need to do is just enable a configuration setting, and there are a host of other features and settings that you can adjust that will make your model perform drastically differently. Some of them do require code changes. For instance, since ZeRO Stage 3 partitions parameters, saving the model and reloading it becomes more difficult since you either need to save every single device model state then recombine them later or recombine the model states first. With the PyTorch DDP model though, all you need to do is call a single function. Other features like mixed precision training also need changes, but this is more beneficial than trying to implement the entire technique directly into your model which would be required if you were to do the equivalent operation in PyTorch. Overall, we believe that DeepSpeed has clear advantages in larger scale training that would make it the better production system.

References

Langer, M., He, Z., Rahayu, W., & Xue, Y. (2020). Distributed Training of Deep Learning Models: A Taxonomic Perspective. *IEEE Transactions on Parallel and Distributed Systems*, 31(12), 2802–2818.

Collective operations. Collective Operations - NCCL 2.28.9 documentation. (n.d.). <https://docs.nvidia.com/deeplearning/nccl/user-guide/docs/usage/collectives.html>

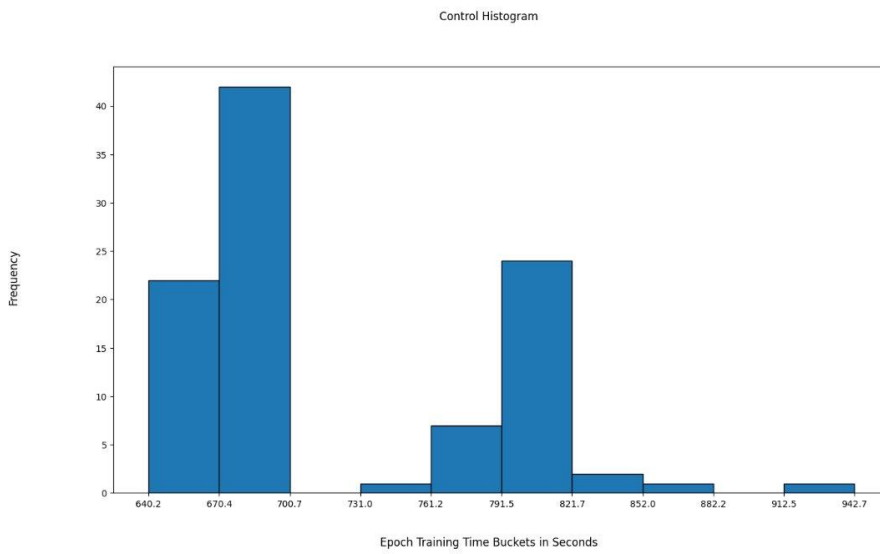
Shen Li, Yanli Zhao, Rohan Varma, Omkar Salpekar, Pieter Noordhuis, Teng Li, Adam Paszke, Jeff Smith, Brian Vaughan, Pritam Damania, & Soumith Chintala. (2020). PyTorch Distributed: Experiences on Accelerating Data Parallel Training.

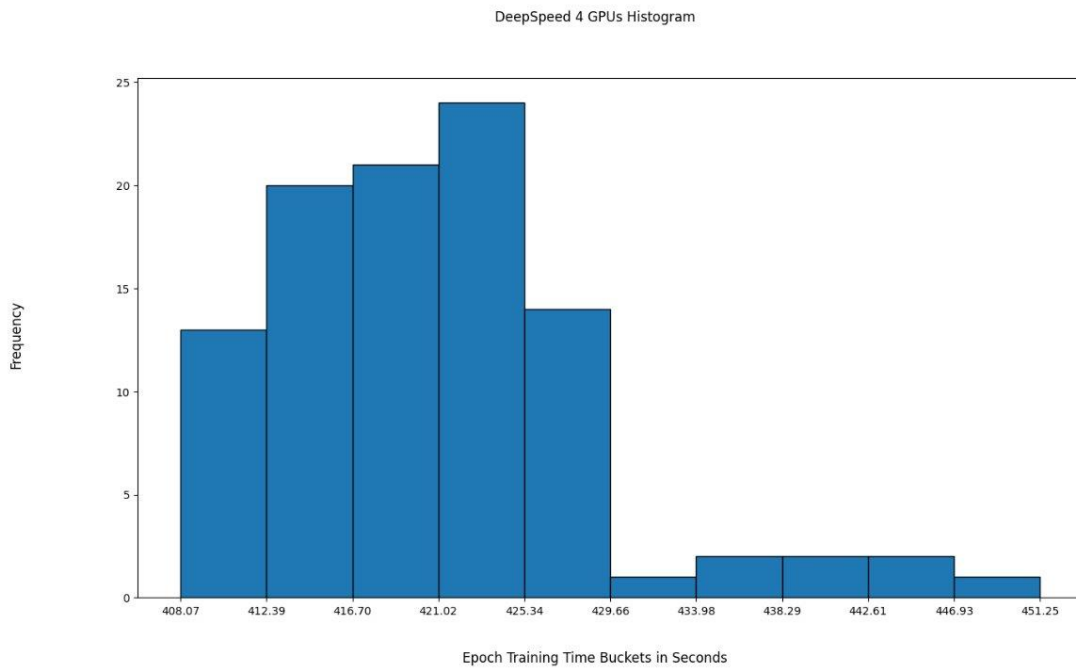
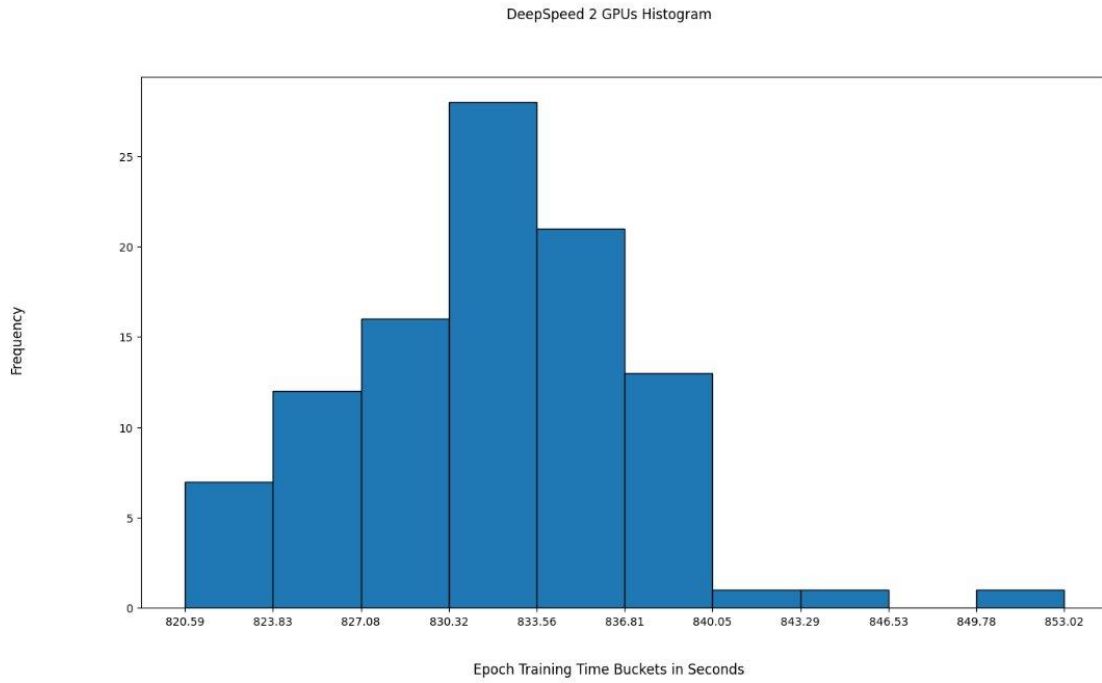
Training overview and features. DeepSpeed. (n.d.). <https://www.deepspeed.ai/training/>

Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, & Yuxiong He. (2020). ZeRO: Memory Optimizations Toward Training Trillion Parameter Models.

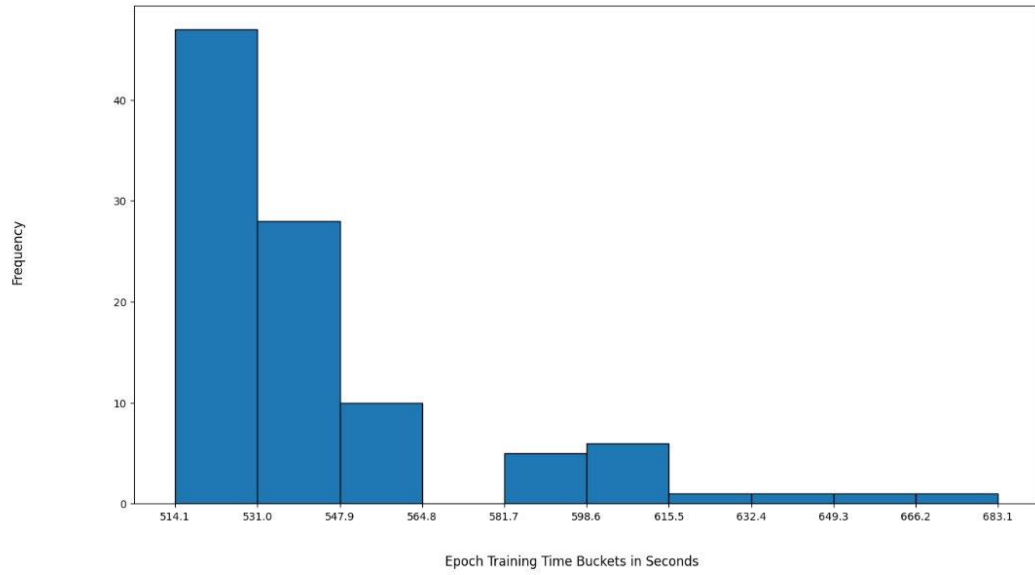
Source Code located at: <https://github.com/EthanHarness/CloudComputingRepo>

Additional Charts





PyTorchDDP 2 GPUs Histogram



PyTorchDDP 4 GPUs Histogram

