PROJECT REPORT: IMPROVING THE CODEBASE WITH PARALLELISM FEATURES

Jiaheng Li & Yisong Wang
Department of Computer Science
ETH Zurich
{jiahli,yiswang}@ethz.ch

1 MOTIVATION

Training a state-of-the-art model (like a large Transformer) on a huge dataset creates two main problems: the model is too big to fit on a single GPU's memory, and the training takes too long to finish within the project timeline. We must use parallel strategies to overcome these memory and time constraints and reach successful convergence.

2 METHOD

2.1 DISTRIBUTED DATA PARALLELISM

To accelerate training, we first employed DistributedDataParallel (DDP), which parallelizes data processing by training with multiple GPUs where each GPU holds the model and processes distinct data batches. While DDP offers excellent training throughput for models that fit into the GPU memory, it is limited by the model size and batch size as each GPU needs to hold a copy of the model weight, corresponding intermediate values, and optimizer states. To address this, we integrated Fully Sharded Data Parallel (FSDP) into our training pipeline. FSDP eliminates memory waste by partitioning model states across the GPU cluster. This reduction in memory overhead not only enables the training of our larger architecture but also frees up VRAM for larger local batch sizes, thereby maximizing GPU utilization and maintaining training stability.

2.2 TENSOR PARALLELISM

Complementing the memory optimizations of FSDP, we employ Tensor Parallelism (TP) to decompose the model's computational graph. Following the Megatron-LM paradigm, we partition the Multi-Head Attention and Feed-Forward Network blocks across available devices. Specifically, the weight matrices of the attention layers and the feed-forward layers are sharded in a combination of column-wise and row-wise. The partial results are then synchronized and combined using an AllReduce operation in the forward pass and gradients are distributed to the responsible devices in the backward pass. This granular distribution enables the training of model architectures with hidden sizes that exceed the memory capacity of any single accelerator.

2.3 DISTRIBUTED CHECKPOINTING

Finally, the scale of the model parameters precludes the use of standard serialization methods, which typically require gathering the full global state dict to a single process—a strategy that inevitably leads to memory exhaustion and significant I/O bottlenecks. To mitigate this, we implement PyTorch Distributed Checkpointing. Unlike legacy approaches, distributed checkpointing enables each rank to persist its local shard of the model and optimizer states directly to storage in parallel. This saving strategy saturates the storage bandwidth, drastically reducing the time required for snapshotting. Furthermore, by writing sharded tensors asynchronously, we minimize the blocking time on the computation path, ensuring that frequent checkpointing for fault tolerance does not degrade overall training throughput.

3 EVALUATION

We will evaluate our improved system based on two main criteria: computational efficiency and model convergence. Efficiency and scalability is crucial for enabling training of memory-intensive models with limited resources and within a reasonable timeframe. Evaluation of model convergence will demonstrate the training quality is maintained. It will further provide an example of the saved training time by using parallelism features. The code is available on `https://github.com/pigeon23/large-scale-ai-engineering`.

3.1 EXPERIMENTS SETUP

For most of the experiments, we will compare our performance with the original codebase. The parameters will remain unchanged, unless necessary, to ensure a fair comparison. We will keep batch size of 1 (per GPU), unless otherwise specified, for consistency due to memory constraints in some cases. We have also supported iterable dataset to have better efficiency. However, since this is not included in the original code base, we will disable this and use regular dataset loader for a fair comparision.

For most efficiency experiments, we will run each configuration for at 200 training steps, and report the average metrics over after the initial 50 steps, for stable measurements. This turns out to be stable enough to reflect the training performance, while keeping the total experiment time manageable. For convergence experiments, we will run each configuration for a longer period of 1000 training steps, and report the training loss dynamics over steps.

We will run the experiments on the provided CLARIDEN cluster. Our experiments will consist of a maximum of 8 nodes, each with 4 GPUs (96GB VRAM each). Therefore, our experiments will sacle up to 32 GPUs in total. For experiments with up to 4 GPUs, we will use a single node to measure the performance without inter-node communication overhead. For experiments with more than 4 GPUs, we will use multiple nodes, each with 4 GPUs allocated.

We will compare our improved system with different parallelism configurations (combination of different parallelism degrees on each parallelism dimension, and therefore different number of total GPUs used). We will also include the original codebase as a baseline, to measure the overhead introduced by implementing parallelism features, since we do not specially treat and optimize for the non-parallel cases in our system.

3.2 COMPUTATIONAL EFFICIENCY AND SCALABILITY

We will first evaluate the computational efficiency and scalability of our improved training system. We will focus on the total throughput (measured in tokens/second) and per-GPU efficiency (measured in TFLOPS per GPU). In addition, we will measure the memory usage per GPU to demonstrate the further scalability enabled by our parallelism features. For distributed checkpointing, we will measure the extra time taken for regular checkpointing operations, to demonstrate the feasibility of reasonably frequent checkpointing without significant overhead.

We will first conduct the evaluation with each of Distributed Data Parallelism (DP) and Tensor Parallelism (TP) individually, to measure their respective performance. Then, we will evaluate a combined system with both DP and TP enabled for a more practical scenario. Finally, we will measure the performance impact of distributed checkpointing with other parallelism features enabled in our system.

3.2.1 DISTRIBUTED DATA PARALLELISM (DP)

As discussed, we will start with the evaluation of Distributed Data Parallelism (DP) using FSDP. We measured the throughput, per-GPU efficiency, and memory usage when training the model with varying number of GPUs. The results are shown in Figure 1.

We could see some overhead introduced by our implementation at degree 1 (single GPU) case, compared with the original codebase. This is expected, since we did not specially treat and optimize for the non-parallel cases in our system. Beyond that, we could also observe some overhead when scaling to multiple GPUs from a degree of 2, but the throughput still considerably increases. It

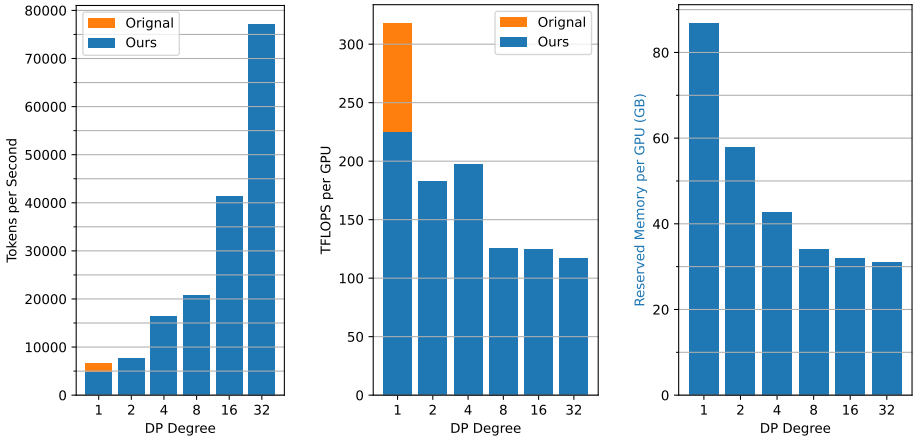

Figure 1: Evaluation results of Distributed Data Parallelism (DP) using FSDP.

then scales pretty well from 2 to 4 GPUs on a same node, maintaning a similar per GPU efficiency. We then scale from one node (4 GPUs) to multiple nodes, and could observe additional inter-node communication overhead, leading to a drop in per-GPU efficiency, while our total throughput still gets some improvement from 4 to 8 GPUs. After this, our system continues to scale pretty well (almost proportionally) in multi-node cases, with only a very slight drop in per-GPU efficiency as the degree goes up. Therefore, we conclude that our DP implementation with FSDP scales pretty well overall, despite some overhead introduced at the beginning of intra-node and inter-node scaling.

Thanks to FSDP, we could also see saving in memory usage per GPU as we scale up with data parallelism. We can see in the figurae that the memory usage per decreases (almost inverse proportionally plus some constant cost) as we increase the number of GPUs, especially when we start from smaller number of GPUs. We can also see the memory usage per GPU approaches to a limit and stablizes as we further scale up. This is also expected, since sharding could reach its limit, and there has to be some constant non-shardable memory cost and overhead per GPU.

3.2.2 Tensor Parallelism (TP)

Directly Applying TP

We then evaluate the performance of Tensor Parallelism (TP) individually. We run the experiments by simply varying degrees of TP, and measure the total throughput, per-GPU efficiency, and memory usage. The results are shown in Figure 2.

We could see the throughput does not scale well with increasing number of GPUs. Consequently, the measured per-GPU efficiency also drops significantly as we increase the TP degree. This is likely due to the small batch size (1 per GPU) we used in the experiments, which limits the amount of computation per GPU, and therefore the communication overhead becomes significant when scaling up. This result is still reasonable because of this configuration and other benefits of TP as we would see.

By contrast, we could see the memory usage per GPU scales down pretty well as we increase the TP degree. The memory usage per GPU decreases significantly until reaching a limit, which is also much lower compared to the DP case in Figure 1. This demonstrate the effectiveness of the sharding methods in TP for saving memory per GPU.

This results are reasonable given the typical usage of TP, for example as suggested in the PyTorch documentation[1]. We will likely use TP after the scale of DP is constrained. In addition, we could

[1] `https://docs.pytorch.org/tutorials/intermediate/TP_tutorial.html#when-and-why-you-should-apply-tensor-parallel`.

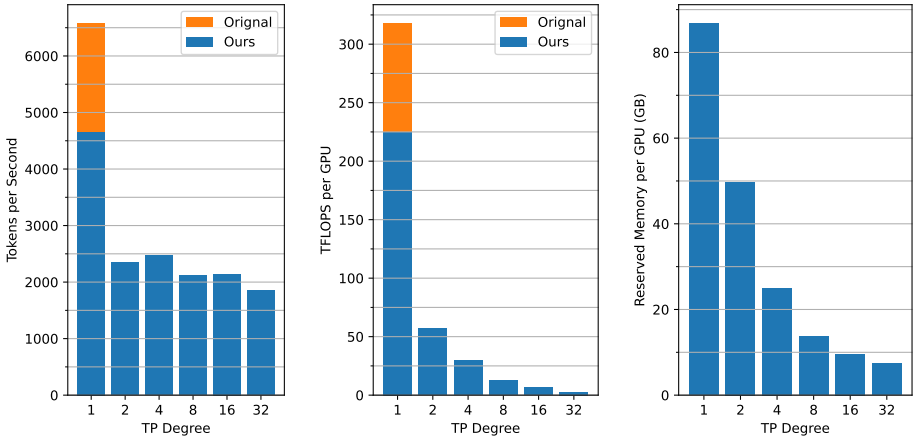

Figure 2: Evaluation results of Tensor Parallelism (TP).

see the significant memory saving enabled by TP, which allows us to further scale the model or data size. The memory scalability could also be further utilized for efficiency.

Adjusting Batch Size

With the memory saving enabled by TP, we could further increase the batch size to improve the computation-to-communication ratio, and therefore the efficiency. This is also reasonable, and fair in some sense to be compared with others, since the global batch size also grows with the degree of DP. As implicitly done by DP, it is pretty reasonable to process more batches globally in one iteration as the number of GPUs scales up to mitigate the communication overhead.

The results are shown in Figure 3, where we tried increase the batch size per GPU proportionally with the TP degree in order to keep the global batch size constant, under the memory constriants. The batch sizes are scaled up to 16, while we fail to further scale it up at 32 GPUs due to the constant memory overhead per GPU. We could then observe the throughput reasonably scales up with the TP degree, before approaching the limit, despite the communication overhead introduced especially at the beginning. We could also see the memory usage is maintained at a reasonable level when increasing the batch size, before we reach the memory constraints because of the constant memory overhead even with TP.

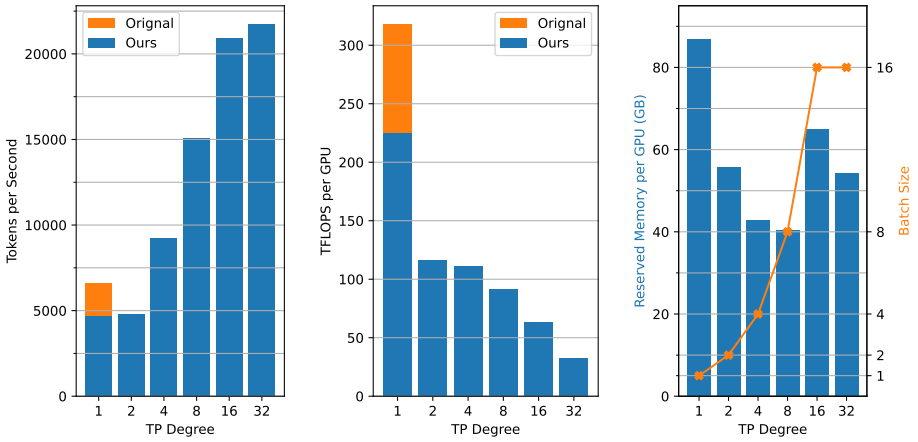

Figure 3: Evaluation results of Tensor Parallelism (TP) with adjusted batch size.

3.2.3 Combining DP and TP

As a more practical scenario and usage of TP, we now evaluate a combined system with both DP and TP enabled. Due to the still limited number of GPUs, we would fix the DP level as 4, and vary the TP level to see the performance impact in this scenario. Similar results also hold with different configurations,but we would focus on this reasonable scenario as an example. As discussed before, we would also adjust the local batch size per GPU to keep the global batch size constant.

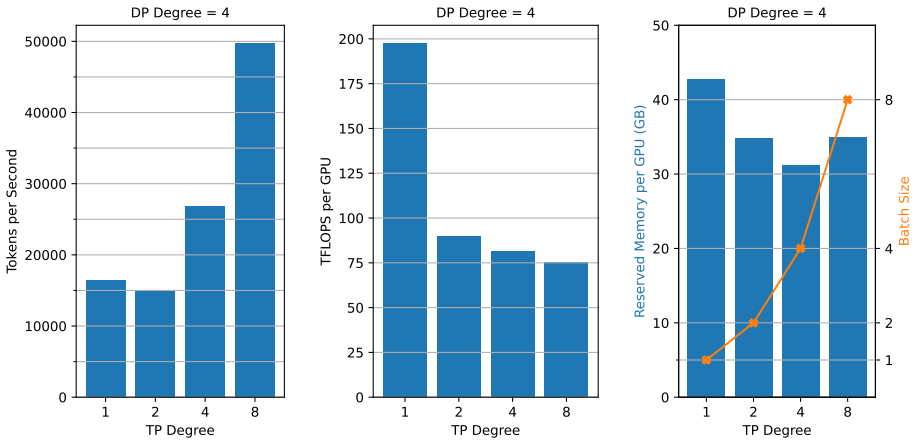

Figure 4: Evaluation results of combined Distributed Data Parallelism (DP) and Tensor Parallelism (TP).

The results are shown in Figure 4. We could see, despite the communication overhead introduced at the beginning of both inter-node communication and TP, the throughput scales reasonably well (almost linearily with only slight drop on per-GPU efficiency). The memory usage per GPU is also maintained at a reasonable and stable level. This demonstrate the effectiveness of our system in a practical scenario combining both DP and TP to further scale up.

3.2.4 Distributed Checkpointing

We finally evaluate the performance impact of distributed checkpointing with other parallelism features enabled in our system to conclude the evaluation of this criteria. We measure the total training time with different regular checkpointing intervals, and compare them with a baseline without checkpointing. We run 1000 steps in total to fit different checkpointing intervals. We enable both DP and TP with degrees DP=4 and TP=8 as an example. The results are shown in Figure 5.

There could be some noise from the filesystem, but we could still see the trend of extra training time introduced by checkpointing. We could see the extra time increases as we checkpoint more frequently, as expected. However, the extra time is still quite reasonably limited even with relatively frequent checkpointing, especially considering that each step in our scenario only takes few seconds. This indicates that our distributed checkpointing is an efficient and feasible solution for enabling fault-tolerant training without significant overhead.

3.3 Model Convergence

We would then evaluate the model convergence of our improved training system. This is crucial to demonstrate that the training quality is preserved with our parallelism features implemented. We will also evaluate the improved training quality from the improved efficiency as expected. Finally, we will consider the overall training time saving from our improved efficiency to reach similar convergence results.

However, achieving good convergence results may need non-trivial hyperparameter tuning, which may be out of the scope of this project. Therefore, our experiment is more a verification of correct-

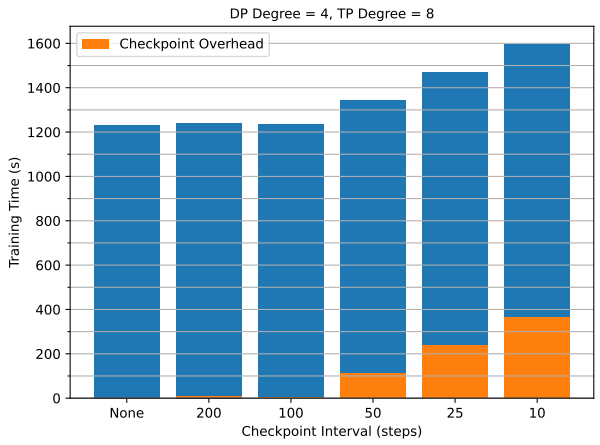

Figure 5: Evaluation results of distributed checkpointing with DP=4 and TP=8.

ness, as well as an example and demonstration of the potential benefits that could be gained from our improved system, rather than a rigorous measurement of the performance limit.

3.3.1 Preserved Convergence Quality

We would not expect significant difference in convergence quality with our parallelism features implemented, since they are mostly orthogonal to the optimization process. We will verify this by comparing the training loss dynamics over steps between single GPU training and TP enabled. The results of the original codebase is also included as a sanity check, since it is expected to be closed to the single GPU case. Note that DP is not included here since it changes the global batch size, which is expected to affect the convergence dynamics. Achieving such large global batch size is not feasible with limited memory on a single GPU.

The results are shown in Figure 6. As we could see, the training loss dynamics are all pretty close to each other, despite some noise. This verifies that our TP implementation does not introduce significant difference in convergence quality, as expected.

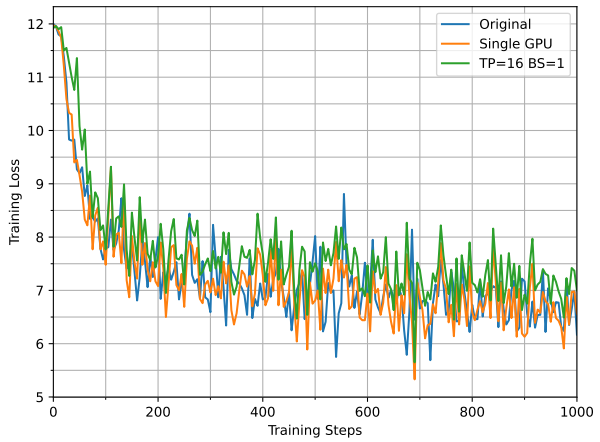

Figure 6: Training loss dynamics with the same global batch sizes.

3.3.2 Quality Improvement with More Data

When DP is enabled, however, the global batch size is increased, and we could therefore expect some improvement in convergence quality due to more data processed in each iteration (with similar

amount of time). We could see the result as expected in Figure 7, where we compare the training loss dynamics between single GPU training and two configurations with increased global batch sizes. Note the local batch sizes are always 1 unless specified.

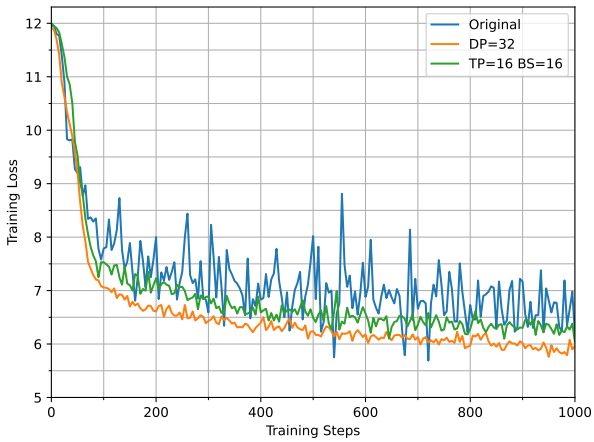

Figure 7: Training loss dynamics with increased global batch sizes.

We could see the improvement in convergence quality (smaller loss) as we increase the global batch size, as expected. The training loss not only decrease faster, but also seems to approach a smaller limit, indicating the better training quality. In addition, we could also observe the much more stable training dynamics with larger global batch sizes, because of the reduced noise from the more data processed in each iteration, which is also a desirable property.

The verified result is pretty much expected that we could achieve better convergence quality with more GPUs used in the same time. To have more insights, we could also consider this problem on the other side, to measure the overall training time saving to reach a similar convergence quality with increased number of GPUs.

3.3.3 Saving of Training Time

With more data processed in each iteration, we would expect the model to converge faster in terms of time. However, to achieve this, we also need to tune other hyperparameters accordingly for the best performance, which may be out of the scope of this project. As an example here, we adjusted the learning rate warmup period proportionally with the global batch size, following the common practice, from 100 to 4 with DP=32, to expect a faster convergence.

We could see the result in Figure 8, where we scale the steps axis of the pralallized training by 5x to match the baseline training quality. We could see that the adjusted parameter is important for a fast convergence, even after the warmup. We could see that we could spend 5x less iterations to reach a similar or better convergence quality with 32 GPUs used, which is a considerable saving in training time.

We would expect more time or iteration saving, as we are processing 32x data in each iteration. However, achieving this may need more careful hyperparameter or optimization tuning. It could be similar to analyzing the convergence dynamics with larger batch sizes, which may be out of the scope of this project. We provide this result without non-trivial parameter modification more as an example and a demonstration of the potential training time saving enabled by our improved system.

In addition, the potential benefits of the parallelism could be more than this, when we have a larger baseline batch size. We could then split the batch and expect pure speedup from parallelism without much affecting the convergence dynamics. However, this is not feasible in our current setup, since we cannot quite increase our single GPU batch size due to memory constraints.

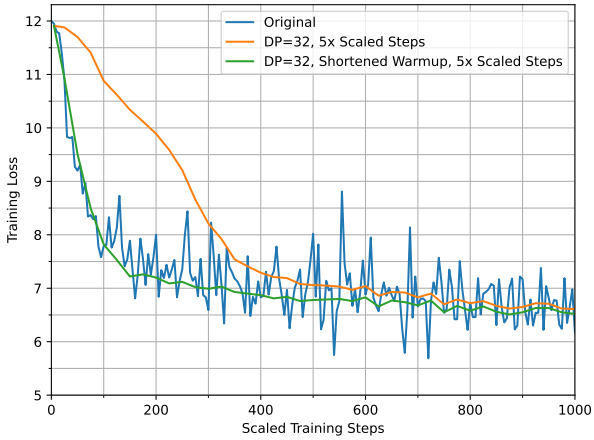

Figure 8: Training loss dynamics with time-scaled steps axis.

4 Conclusion

By transitioning from a standard Pytorch training framework to a composite architecture featuring FSDP, Tensor Parallelism, and Distributed Checkpointing, we have established a robust infrastructure capable of sustaining long-duration training runs. This approach mitigates the distinct failure modes associated with scaling: FSDP resolves memory fragmentation, TP allows wider intermediate layers, and sharded checkpointing prevents I/O timeouts. This ML training system design not only enables large-scale model training but also ensures that the expansion of model depth and width is limited only by the aggregate resources of the cluster.