CSE-234: Data Systems for Machine Learning, Winter 2025

Data Parallelism

Lecturer: Hao Zhang *Date: Feb 18th*

1 Why Data Parallelism First

Data parallelism is a specific form of intra-parallelism and is widely adopted in most parallelism strategies.

The paper *Disbelief*, authored by Jeffrey Dean, introduced one of the first distributed systems for training neural networks. The primary parallelism approach used in Disbelief is data parallelism.

Data parallelism was the first parallelism feature introduced in `PyTorch`. The operation behind the `DDP` interface is essentially an all-reduce operation.

As shown in Figure 1, each rank must share its value with all other ranks. After receiving values from all other ranks, each rank can compute its final all-reduce result.

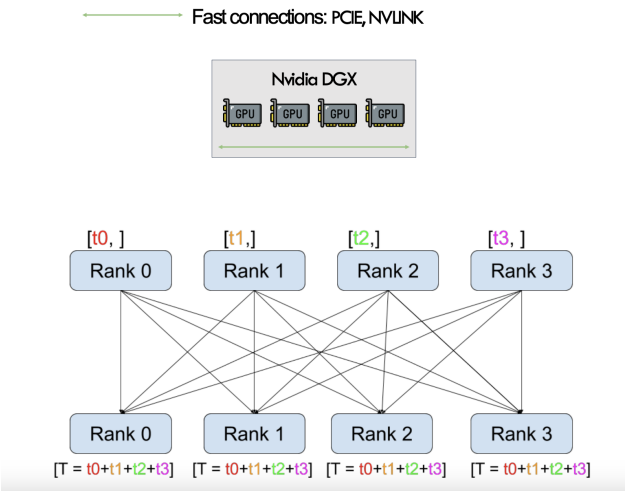

Figure 1: Illustration of the All-Reduce operation.

2 Data Parallelism

2.1 Data Parallelism and Communication Challenge

Figure 2 illustrates a model with two layers, where input data is partitioned across multiple computing devices. Each device maintains a local copy of the model parameters and processes a different subset of the data.

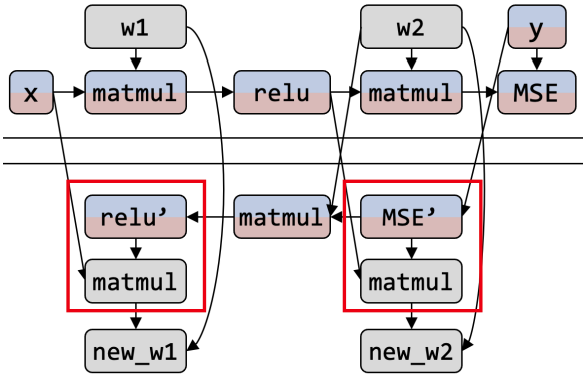

Figure 2: Data Parallelism

During the forward pass, computations occur independently on each device without requiring communication. However, in the backward pass, each device computes gradients based on its assigned batch, resulting in partial gradients that are local to each device. The challenge arises because updating the model correctly requires each device to access the full gradient information from all devices. This necessitates an efficient communication mechanism in the red-framed part in Figure 2 to aggregate gradients across devices.

2.2 Two Solutions for the Challenge

To address this communication challenge, two widely used approaches are introduced, which are Parameter Server Approach and AllReduce Approach. They should all satisfy one assumption: The model can fit into an (GPU) worker memory hence we can create many replica.

3 Parameter Server Approach

$$\theta(t+1) = \theta(t) + \epsilon \sum_{p=1}^{P} \nabla L(\theta(t), D_p(t)) \tag{1}$$

where θ represents the model parameters, P is the number of workers, and $\nabla L(\theta(t), D_p(t))$ represents the gradient computed on each worker.

Architecture The parameter server approach can be mathematically represented as equation (1). In this approach, each worker independently computes gradients without any communication during this step. Once the gradients are computed, all workers send them to the central parameter server. The server aggregates the received gradients, updates the model parameters accordingly, and then sends the updated parameters back to all workers. This ensures that every worker uses the same model in the next iteration.

Two Key Assumptions There are two key assumptions underlying the parameter server approach. First, each iteration involves heavy communication overhead, as all workers must continuously exchange data with

the central server. Second, in the era of 2012, the compute-to-communication ratio was approximately 1:10, meaning that communication costs were significantly higher compared to computation.

Two major challenges the parameter server approach faces two major challenges. The first is the communication bottleneck—since all workers must send gradients to and receive updated parameters from the same server, scalability becomes a critical issue. As the number of workers increases, communication overhead grows exponentially, limiting efficiency. The second challenge is the single point of failure—if the parameter server crashes, the entire training process is disrupted. Such failures may result in loss of gradients or even require restarting the training, making the system less robust and reliable.

3.1 Server bottleneck

In 3, we first perform a reduced scatter across all nodes, we have to let each worker send that shard of gradients to the server that holds the shard. After that, we have to get a new copy of the new parameter. It elevates the bottleneck

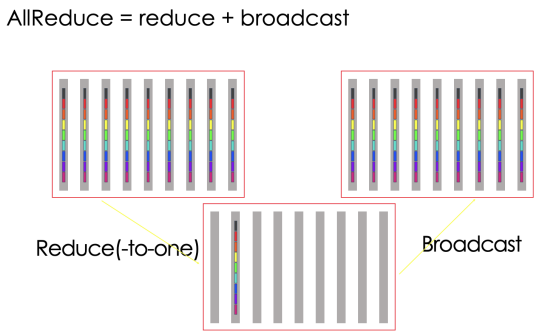

Figure 3: Illustration of AllReduce.

We need to do a synchronization operation. Every time each worker finishes the gradient, it has to send the gradient to the server, server aggregate that and send it back. In reality, some workers may be slower than others. The overall speed is determined by the slowest one.4

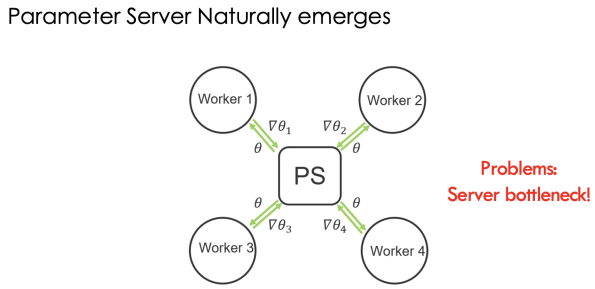

Figure 4: Illustration of AllReduce.

3.2 Consistency Models for Parameter Server

In distributed systems, consistency model characterizes when you should synchronize your copy of gradients in machine learning. As shown in Fig.5, there are three devices A, B, and C, where each device is a worker. At the end of computation, workers need to communicate with each other. In a naive setting, we assume that workers will never experience faults and every worker will run at the same pace. Following the parameter update equation in the upper part of Fig.5, each worker will calculate its own copy of gradients and send the gradients to the parameter server. Then workers will wait until the updated parameters are sent back to them, which is illustrated by the black bar (Global Synchronization Barrier) in the figure. The above procedures will repeat until the end of execution.

$$\theta^{(t+1)} = \theta^{(t)} + \varepsilon \sum_{p=1}^{P} \nabla_{\mathcal{L}}(\theta^{(t)}, D_p^{(t)})$$

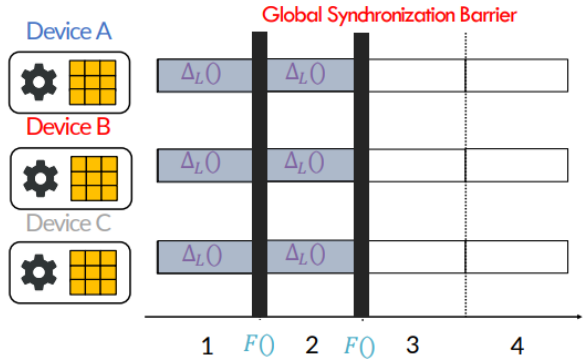

Figure 5: Illustration of a Consistency Model.

In reality, the behavior is illustrated in Fig.6, where device B performs the fastest, followed by device C, while device A is the slowest. Under this scenario, devices B and C must wait for device A to complete its computation, creating idle periods called **Bubbles**. This synchronization approach is known as box synchronous consistency. As the number of devices increases, the probability of encountering a slower device (straggler) also increases.

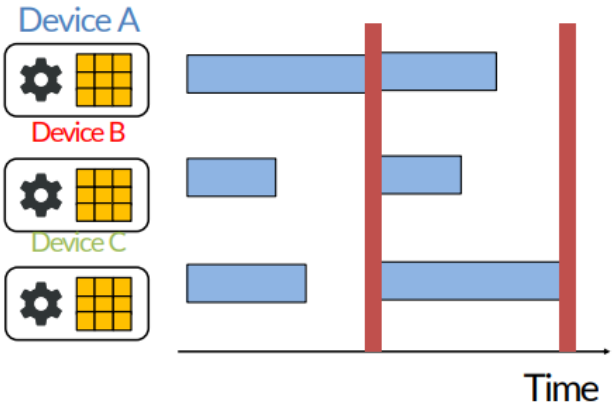

Figure 6: Illustration of a Consistency Model in reality.

To reduce bubbles, we can relax the box synchronous consistency. The intuition is that each worker computes a copy of gradients, and the server aggregates these gradients before performing gradient descent. Since machine learning algorithms are highly error-tolerant, meaning that even with some introduced errors the model can still function effectively, we can allow some deviation without compromising convergence. This suggests we can use a "slightly wrong version" of the gradients to update parameters under certain conditions, as shown in Fig.7.

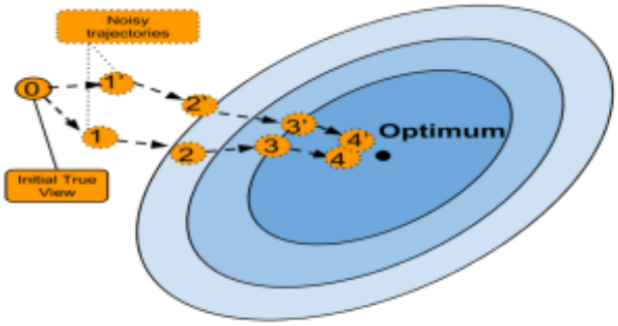

Figure 7: Machine learning algorithms are error-tolerant.

3.3 Asynchronous Communication (No Consistency)

One way to relax the consistency model is to not have consistency. Every worker will compute at their own pace and would only update every 100 or 200 gradients. Every worker will take their own pace to update their parameters and it is likely that all workers will be divided.

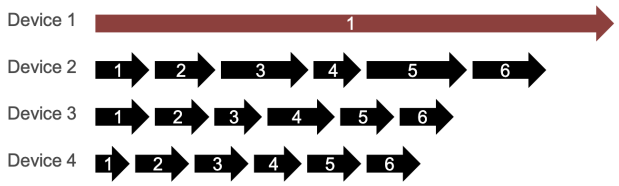

Figure 8: Asynchronous Communication Example

In the above example, Device 1 is the slowest and only performs one iteration in the entire window time. Devices 2, 3, and 4 perform multiple iterations in the window time and there is no barrier between the devices. However, this does not work because it defeats the purpose of data parallelism since you are not aggregating gradients at all.

3.4 Bounded Consistency (SSP)

The middle ground between strong consistency and no consistency is stale consistency. This introduce a new staleness variable which characterizes how stale the parameters are.

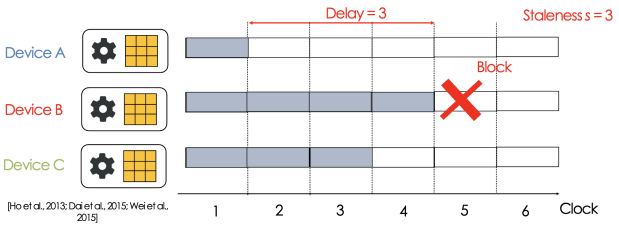

Figure 9: Bounded Consistency Example

In the above example, Device B is faster than Device A and reaches the fourth iteration while Device A is only at the first. Because this has exceeded the staleness variable of 3, Device B is blocked so that the devices can then synchronize during this window.

3.5 Impacts of Consistency/Staleness

The staleness variable impacts whether your model converges or not and we have found that a staleness variable of 0, 2, 3, or 5 are all acceptable as shown in the graph below.

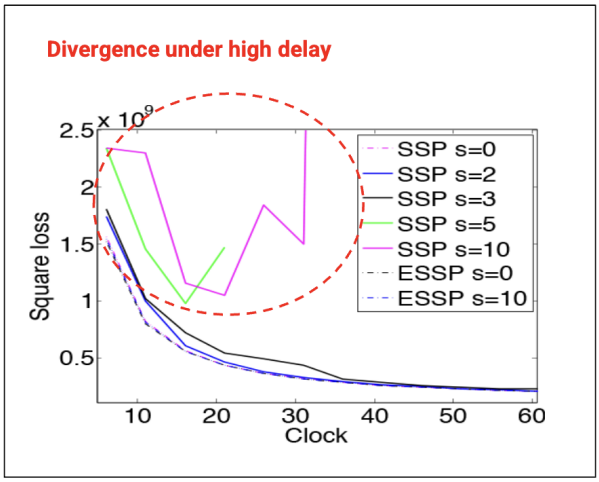

Figure 10: Performance of Different Staleness Values

3.6 Theory: SSP Expectation Bound

The difference between a staleness estimate and true estimate can be bounded by the term on the right side of the equation below which is a function of the staleness itself. As long as your staleness is not too large, the result will be optimal. This is a very active area of research and many people are still working on this today.

$$R[\mathbf{X}] := \overbrace{\left[\frac{1}{T}\sum_{t=1}^{T} f_t(\tilde{\mathbf{x}}_t)\right]}^{\text{Difference between SSP estimate and true optimum}} - f(\mathbf{x}^*) \leq 4FL\sqrt{\frac{2(s+1)P}{T}}$$

Figure 11: SSP Expectation Bound Equation

The problem is that this does not work well in neural networks because the equation requires a lot of assumptions about your objective. For example, there needs to be some convex characteristics about the objective and you have to do gradients given learning rate. Empirically, when you apply staleness to neural networks it does not work too well.

4 AllReduce

Another way of implementing data parallelism is AllReduce. It consists of two main phases: 1. Reduce, where data from multiple nodes is aggregated using an operation such as sum, max or min. 2. Broadcast, where the aggregated result is distributed back to all nodes.
The most popular AllReduce interface is DDP(Distributed Data Parallel), provided by PyTorch. In DDP, we declare ranks among all the processes and then for each rank a duplicated model will be declared.
AllReduce was initially implemented in a framework called Horovod by Uber. Then Nvidia optimized

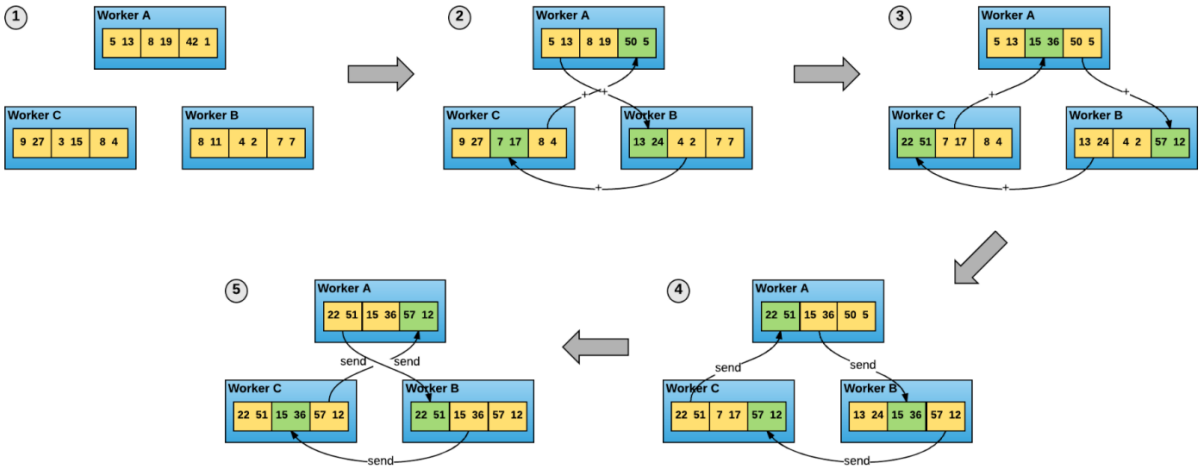

Figure 12: Scheme for Performing AllReduce.

AllReduce in their own library NCCL (NVIDIA Collective Communications Library). Later PyTorch adopted NCCL in its DDP.
The weakness of AllReduce is that it is not fault tolerant. If one worker fails, the whole AllReduce will fail. However, in the machine learning field, fault tolerance is not so important, therefore AllReduce can still dominate parameter servers nowadays.
One of the reasons that AllReduce can succeed is that it is simple enough to implement. Another reason is because of the development of hardware, which has made the communication bottleneck among workers no longer a problem.

5 Computational Graph (Neural Networks) → Stages

If you remember from previous lectures, a model is represented as a computational graph——where the nodes represent operations and edges represent data flow. An example can be shown below:

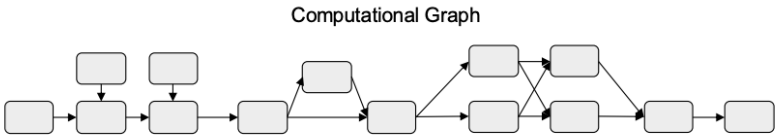

Figure 13: Computational graph

Now lets assume we have some 4 devices available. The question now remains, what is the most straightforward way to split up this computational graph?

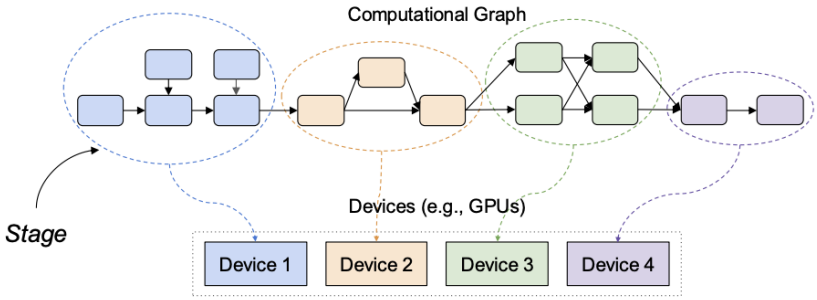

Figure 14: Partitioned Graph

The most straightforward way is shown above——where nodes are equally distributed across devices (or close to equal). This partitioning method resembles inter-parallelism and each device now holds one *stage*——meaning this whole graph was broken up into 4 *stages*. Given this, the execution of the model is represented by:

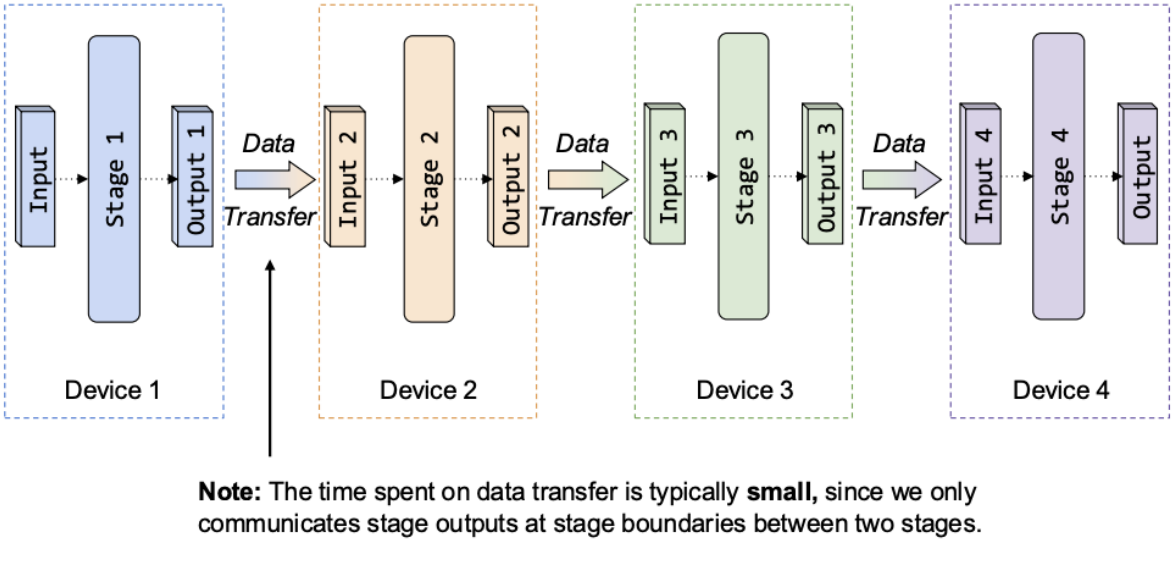

Figure 15: Enter Caption

As you can see, the stages are operated in a sequential manner——where the output of one stage becomes the input of another. The data transfer between stages is small because only the outputs are being transferred. Since this data transfer is a bottle neck, having smaller bandwidths are better for inter-parallelism.

Although this concept is simple to implement, devices often remain idle when one stage is executing. For example, While stage 1 is running, stages 2, 3, and 4 all remain idle due to the dependence on stage 1——meaning these stages can't function without stage 1's output.

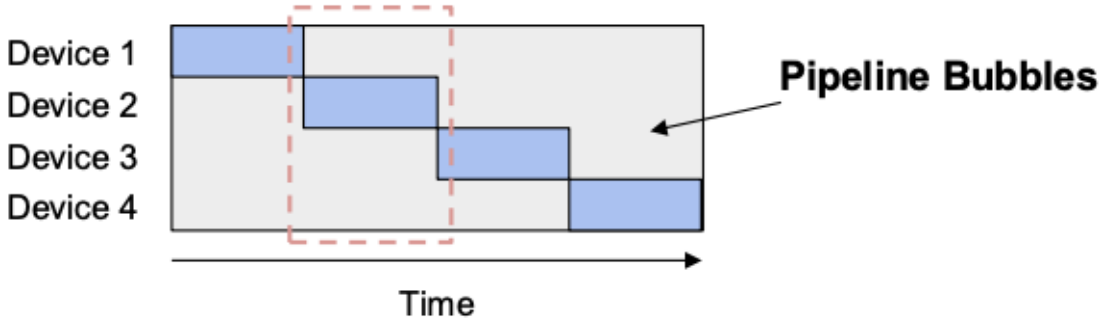

Figure 16: Inter-Parallelism

The gray bubbles present in figure 4, are pipeline bubbles——meaning those devices are idle. As Professor Zhang exclaimed, "Very Bad." To calculate the pipline bubble percentage you can use this formula: $\frac{bubble\_area}{total\_area}$. For this specific case, the equation is $\frac{D-1}{D} \rightarrow \frac{4-1}{4} \rightarrow \frac{3}{4} = 75\%$ since only one device is running in the circled stage.

5.1 Reduce Pipeline Bubbles

Here we find a way to reduce pipeline bubbles, that is, by pipelining inputs. By pipelining inputs at different stages, we can parallel execution among devices and reduce pipeline bubbles.

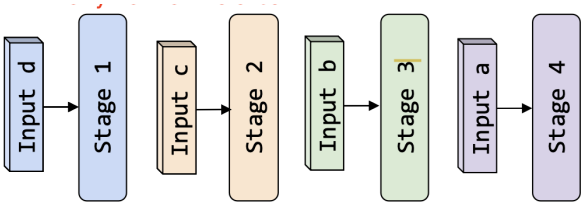

Figure 17: reducing bubbles by pipeline

The bubble percentage now becomes $(D-1)/(D-1+N)$ with D devices and N inputs. As we can see from the equation, when N goes to infinity, the bubble will diminish, but this only works for the inference stage because each computation is independent of others. At training stage, layer dependency is introduced by back propagation, such waiting may introduce more bubbles.

Here're several approaches to reduce bubbles for training.

- Device Placement: Ensure that each stage in the pipeline has a similar computation time to prevent idle time in certain stages.
- Synchronous Pipeline Parallel Algorithms: Introducing strong consistency by providing stable training but have bubbles due to synchronization. **(GPipe, 1F1B and Chimera)** .
- Asynchronous Pipeline Parallel Algorithms: Introducing relax consistency by reducing bubbles but introduce stale gradients, which may impact model convergence. **(AMPNet, Pipedream/Pipedream-2BW)**

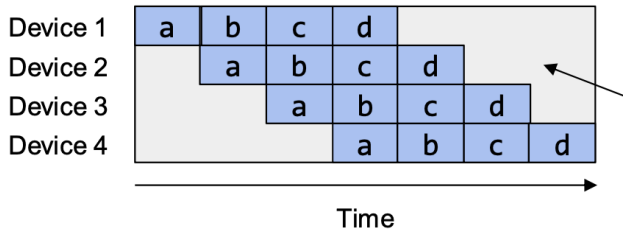

Figure 18: pipeline bubble via pipelining inputs

5.2 Limitations of Device Placement

Although device placement can reduce pipeline bubbles by enabling concurrent execution of multiple stages, it is only effective for certain neural networks with branched architectures, such as the Inception module [3] and contrastive models [2]. For other architectures like conventional CNNs and transformers, we cannot utilize device placement strategies to reduce bubbles directly.

Furthermore, even when device placement is applied, device utilization remains suboptimal as there are still significant pipeline bubbles. The staggered execution of forward and backward passes still leaves gaps in device usage, leading to inefficiencies. To further improve utilization, device placement needs to be combined with advanced pipeline scheduling techniques, which will be discussed in later sections.

6 Synchronous Pipeline Parallel Schedule

Synchronous pipeline parallel schedule aims to enhance efficiency by modifying the pipeline execution strategy while preserving the exact computation and convergence semantics as if training were conducted on a single device. This approach ensures that parallel execution does not introduce inconsistencies in gradient updates or training dynamics.

6.1 GPipe

GPipe [1] is a pipeline parallelism approach that partitions the input batch into multiple *micro-batches* and processes them sequentially across different devices. As shown in the bottom of Figure 19, instead of waiting for the full batch to complete before starting the backward pass, GPipe enables overlapping computation by pipelining the forward and backward passes of different micro-batches. The gradients from each micro-batch are accumulated, ensuring that the final parameter update is equivalent to training with a single batch:

$$\nabla L_\theta(x) = \frac{1}{N}\sum_{i=1}^{N} \nabla L_\theta(x_i)$$

where N is the number of micro-batches.

As illustrated in the example, each batch is split into six micro-batches, and computation is distributed across four devices. A key advantage of increasing N (the number of micro-batches) is the reduction of *pipeline bubbles*, which are idle periods caused by the staggered execution of different stages. The proportion of pipeline bubbles is given by:

$$\frac{D-1}{D-1+N}$$

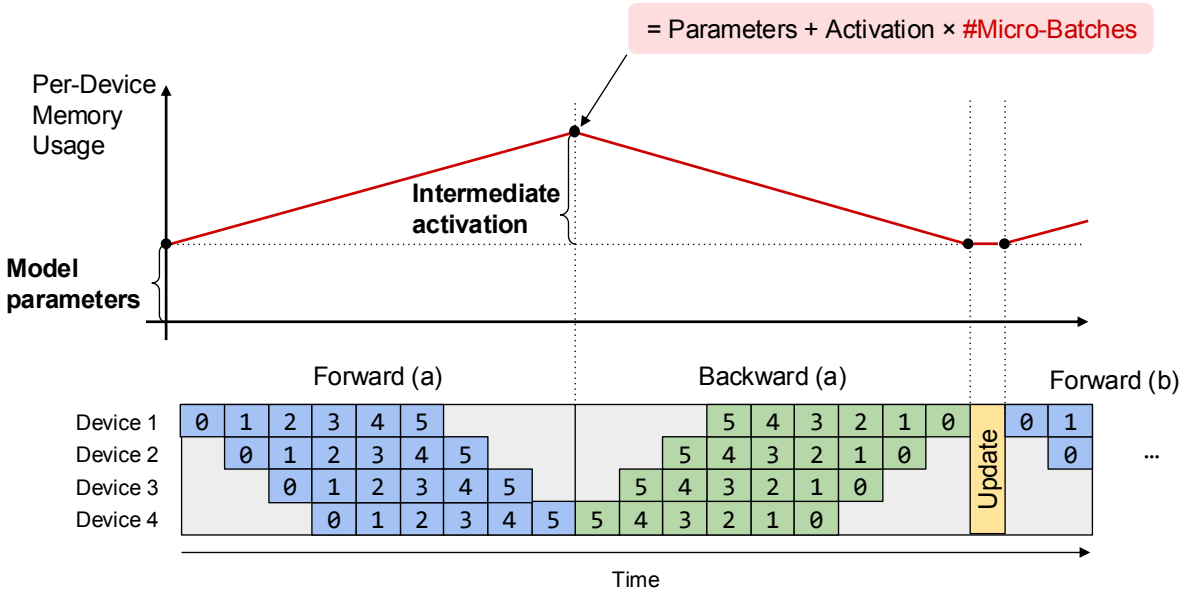

Figure 19: Illustration of the pipeline parallelism with GPipe. The top part of the figure represents the device memory consumption during the pipeline.

where D is the number of pipeline stages (devices). As N increases, this fraction decreases, improving overall device utilization.

However, increasing N also reduces the size of each micro-batch, which can negatively impact computational efficiency. A smaller micro-batch leads to lower arithmetic intensity, reducing the device utilization due to inefficient memory access patterns and increased CPU overhead. Thus, a trade-off must be struck between minimizing pipeline bubbles and maintaining a sufficiently large micro-batch size to maximize per-device computational efficiency.

Table 1: Normalized training throughput using GPipe with different device numbers and micro-batches [1].

	#TPUs = 2	**#TPUs = 4**	**#TPUs = 8**
#Micro-batches = 1	1.0	1.07	1.3
#Micro-batches = 4	1.7	3.2	4.8
#Micro-batches = 32	1.8	3.4	6.3

As shown in Table 1, by partitioning the input batch into multiple micro-batches, we can increase device utilization significantly, thus improving training throughput. As the number of micro-batches increases, the pipeline becomes more efficient by reducing idle time across devices.

The top of Figure 19 illustrates GPipe's per-device memory consumption, which consists of model parameters and intermediate activations. While model parameters remain constant, activation memory grows with the number of micro-batches. Memory peaks during forward propagation as activations accumulate and decreases in the backward pass when they are freed. The total memory per device is:

$$\text{Memory Usage} = \text{Parameters} + \text{Activation} \times \#\text{Micro-Batches}$$

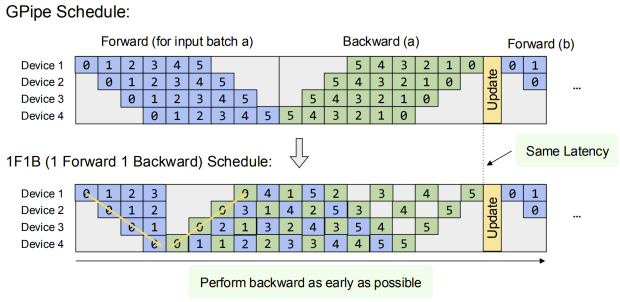

Figure 20: 1F1B Schedule

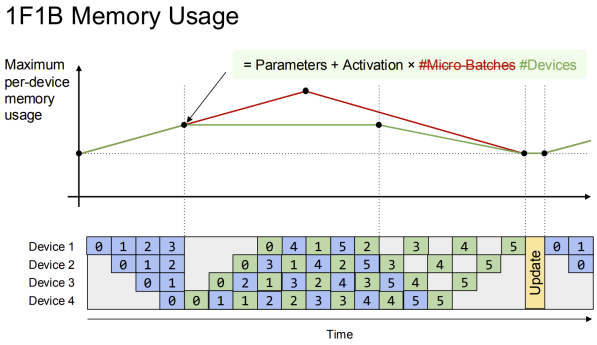

Figure 21: 1F1B Memory Usage (Green) vs GPipe Memory Usage (Red)

Thus, while increasing the number of micro-batches improves device utilization and training throughput, it also raises memory consumption. The number of micro-batches is therefore constrained by the available memory on each device. A balance must be maintained between maximizing parallelism and ensuring memory feasibility.

6.2 1F1B Schedule

The 1F1B schedule is an optimization to the GPipe schedule that reduces memory usage through more efficiently scheduling backwards calculations. The 1F1B schedule prioritizes the execution of the backwards pass, meaning as soon as there becomes a microbatch available for the backwards pass, it is then processed. This can be seen in the figure below. It is shown that the forward and backward passes are intervewved with each other. The 1F1B schedule has the same latency as the GPipe schedule, but where the optimizations come in is that it allows for memory to be realeased earlier. A plot of the maximum per-device memory usage of both a GPipe schedule and 1F1B schedule is seen in the figure above. It is seen that the 1F1B schedule memory usage plateaus once a microbatch becomes available, whereas the GPipe schedule peaks as the microbatches accumulate to be processed all at once. The 1F1B method uses less memory, and only needs to store the number of devices rather than the number of microbatches. It is the method that is adopted today for training GPTs and LLMs.

6.3 Interleaved 1F1B Schedule

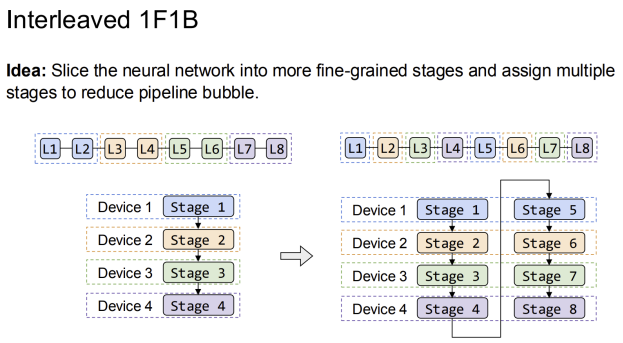

Figure 22: Interleaved 1F1B

A further optimization of the 1F1B schedule is the interleaved 1F1B method. Here, the neural network is sliced into finer stages and each device has multiple stages, as shown above. Previous methods would cluster contiguous layers to form 4 stages, but here 8 stages are formed. The main benefit of this is that the latency of a single stage becomes shorter.

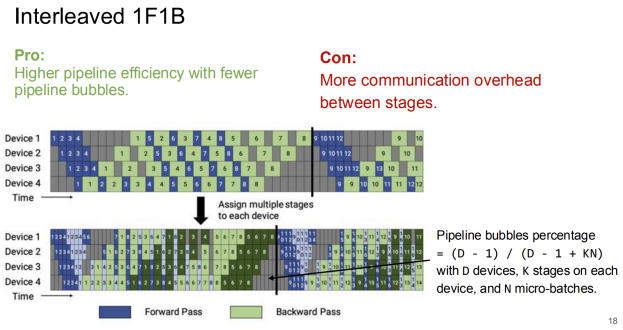

Figure 23: Interleaved 1F1B Pros and Cons

The new schedule further reduces the pipeline bubble. Whereas the regular 1F1B method had the same latency as GPipe, the interleaved 1F1B schedule reduces latency. The cost of this is that since there are more stages there is a more communication between devices, adding additional communication overhead. However, the communication cost is generally small so the method is overall more efficient.

7 Terapipe & Chimera

Terapipe is pipeline parallelism for autoregressive models (GPT, Transformers). The key insight here is that the next token only depends on previous tokens, so we can put each layer on a device and pipeline like the image shows:

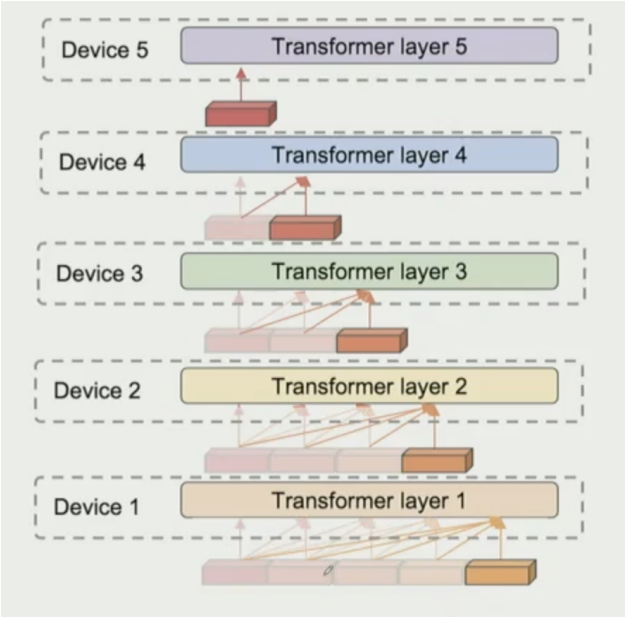

Figure 24: Pipeline of Terapipe

Another important work is Chimera. Chimera is the pipeline parallelism used in Deepseek. It attempts to further optimize the 1F1B pipeline by having one set of batches go in the opposite direction on the devices as the image below shows.

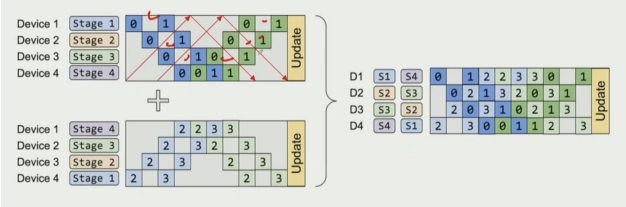

Figure 25: Overview of Chimera

The downside is that there are copies of parameters on each device, so more memory intensive. However, with current accelerators this is acceptable.

References

[1] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Dehao Chen, Mia Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V Le, Yonghui Wu, et al. Gpipe: Efficient training of giant neural networks using pipeline parallelism. *Advances in neural information processing systems*, 32, 2019.

[2] Alec Radford, Jong Wook Kim, Chris Hallacy, Aditya Ramesh, Gabriel Goh, Sandhini Agarwal, Girish Sastry, Amanda Askell, Pamela Mishkin, Jack Clark, et al. Learning transferable visual models from natural language supervision. In *International conference on machine learning*, pages 8748–8763. PmLR, 2021.

[3] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1–9, 2015.

8 Contributions

1. Weijia Zeng: Section 15-18
2. Kyle Shao: Section 1-6
3. Anais Zhu: Section 23-26
4. Aniket Pratap: Section 27-31
5. Edward Jin: Section 45-48
6. Zhi Xu: Section 11-14
7. Zihe Liu: Section 7-10
8. Zaifeng Pan: Section 36-40
9. Yaxuan Li: Section 32-35
10. Kenneth Vuong: Section 41-44
11. Thomas Rexin: Section 19-22
12. Zhuoxin Liu: Final merging and compilation
13. Dhylan Patel: Final editing/review