सुस्वागतम्

நல்வரவு

ಸುಸ್ವಾಗತಂ

సుస్వాగతం

സുസ്വാഗതം

সুস্বাগতম

ସୁସ୍ୱାଗତମ

ਸੁਆਗਤਮ

સુસ્વાગતમ્

خوش آمدید

Ministry of Electronics and Information Technology
Government of India

Parallel and Distributed Training

A Deep Dive into Techniques for Scaling Deep Learning on High-Performance Computing

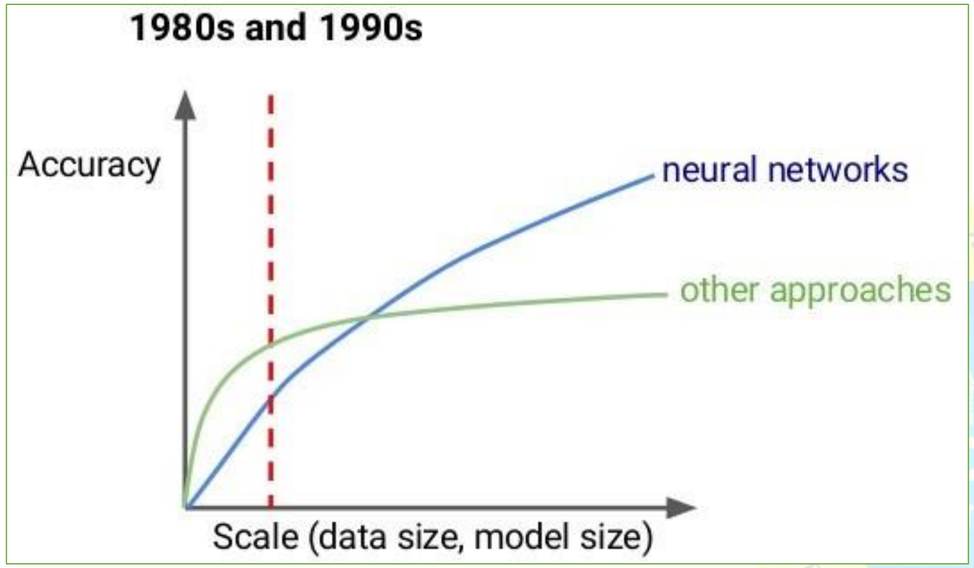
Kishor Y D
HPC Technologies
C-DAC Pune

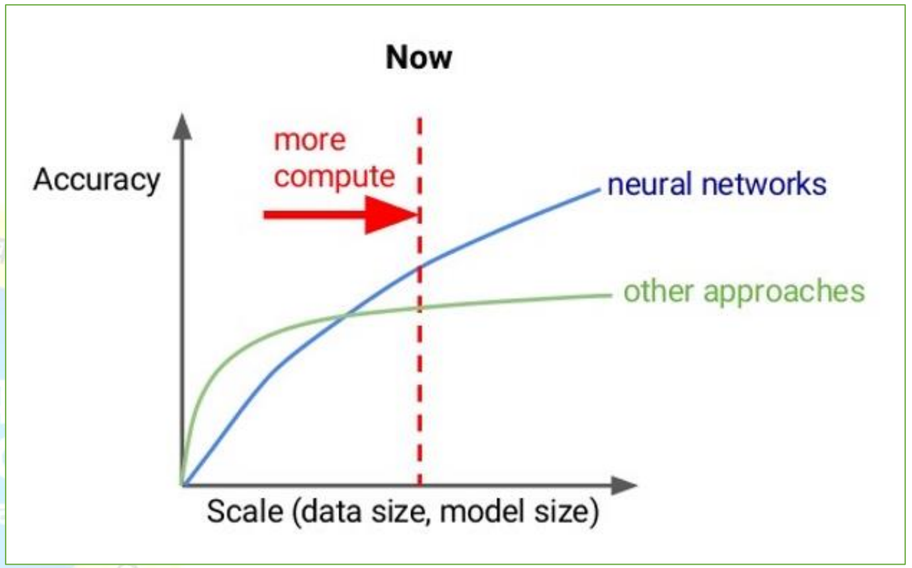
One Vision. One Goal... Advanced Computing for Human Advancement...

Agenda

- Introduction: Overview of Distributed Training.
- Why Distributed Training?: Challenges and benefits of scaling.
- Types of Parallelism: Data, Model, and Hybrid Parallelism.
- Challenges: Issues faced in different parallelism
- Distributed Techniques: PyTorch DDP and FSDP.
- Communication: Collective primitives and checkpointing.

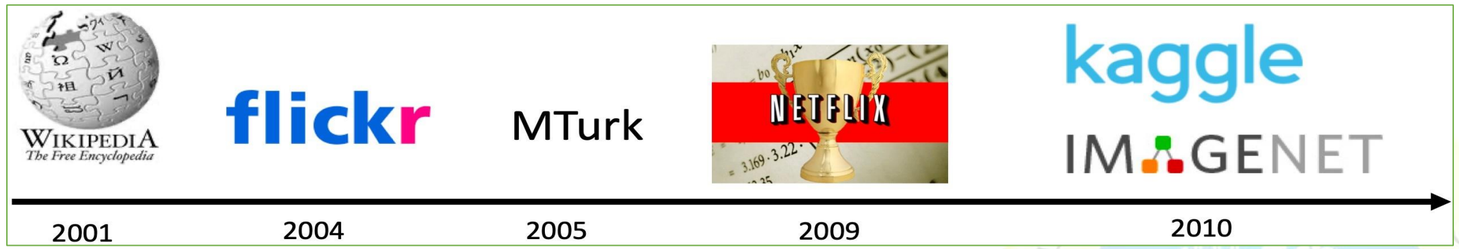
The Rise of ML and Neural Networks



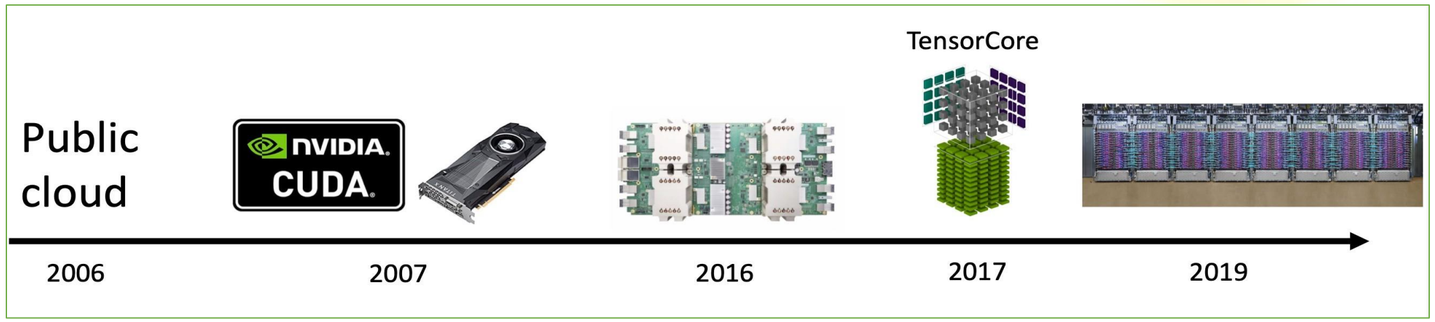


The Rise of ML and Neural Networks

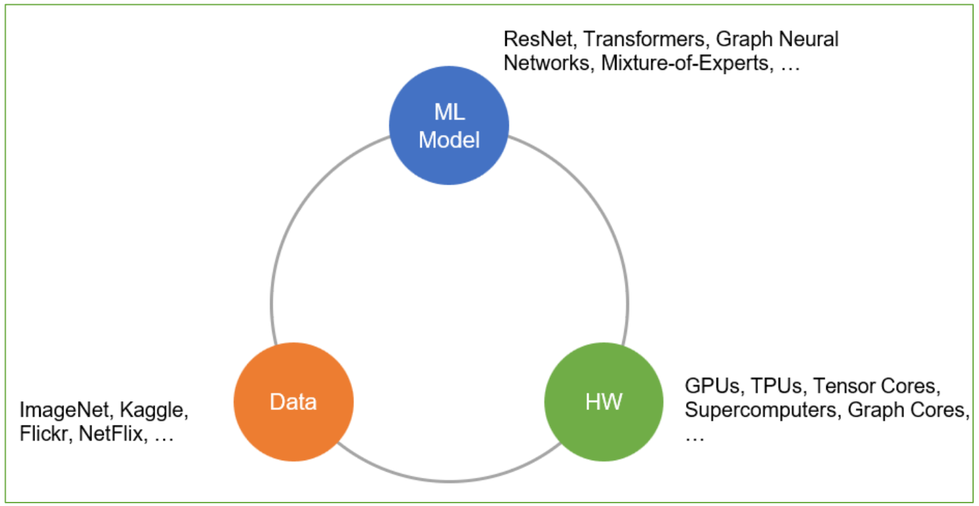
Big data arrives in early 2000



AI hardware becomes widely available in 2010s



The Secret Ingredients in ML Success



Why distributed Computing ?

1. Saves Time

Divides tasks among multiple machines, allowing them to be processed in parallel. This reduces the overall execution time compared to running tasks on a single machine.

2. Increases the Amount of Compute

Distributed systems pool computational resources from multiple machines or clusters, providing higher collective computational power. This is especially critical for resource-intensive applications like weather simulations, genome analysis, or high-performance AI training, which demand processing capabilities beyond what a single machine can provide.

3. Helps Train Models Faster

In AI and machine learning, distributed computing enables parallel training by splitting datasets and computations across multiple nodes or GPUs.

4. Enhances Fault Tolerance and Reliability

By distributing tasks across multiple nodes, the system can handle hardware or software failures more gracefully. If one machine fails, others can take over its workload. This redundancy ensures minimal downtime, which is essential for critical systems like financial services, e-commerce platforms, or healthcare applications.

Introduction to distributed training

What is distributed training ?

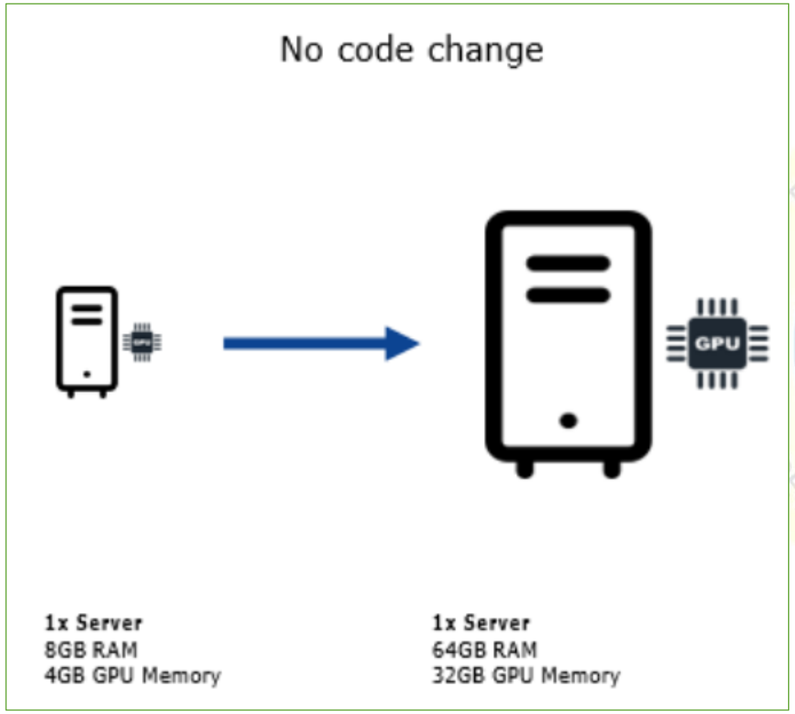
Imagine you want to train a Language Model on a very big dataset, for example the entire content of Wikipedia. The dataset is quite big, because it is made up of millions of articles, each of them with thousands of tokens. To train this model on a single GPU may be possible, but it poses some challenges:

1. **The model may not fit on a single GPU: this happens when the model has many parameters.**
2. **You are forced to use a small batch size because a bigger batch size leads to an Out Of Memory error on CUDA.**
3. **The model may take years to train because the dataset is huge.**

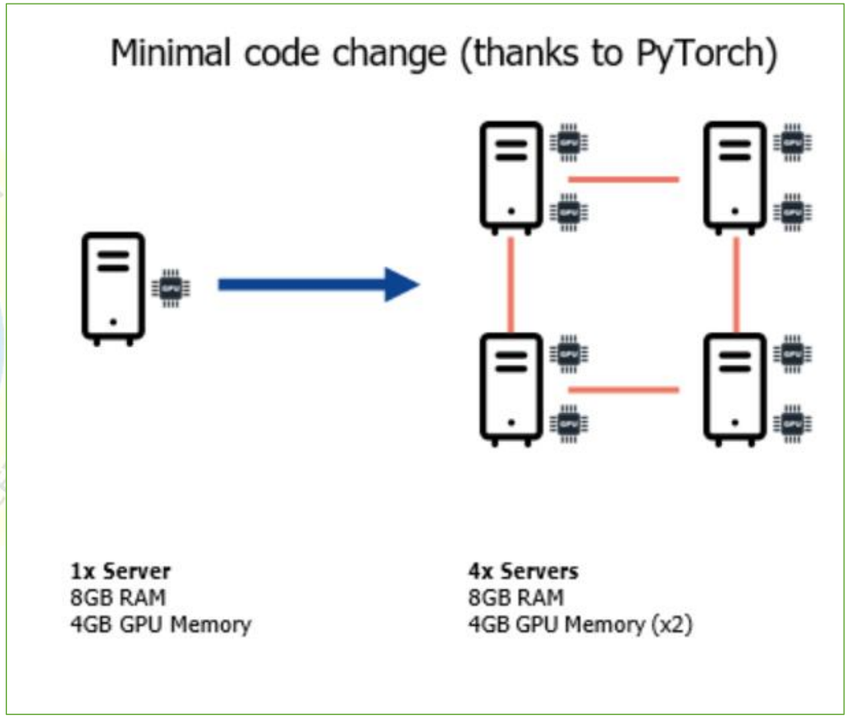
If any of the above applies to you, then you need to scale your training setup. Scaling can be done vertically, or horizontally. Let’s compare these two options.

Different Types of Scaling

Vertical Scaling



Horizontal Scaling



DNN Training Process

Train DL models through many iterations of 3 stages

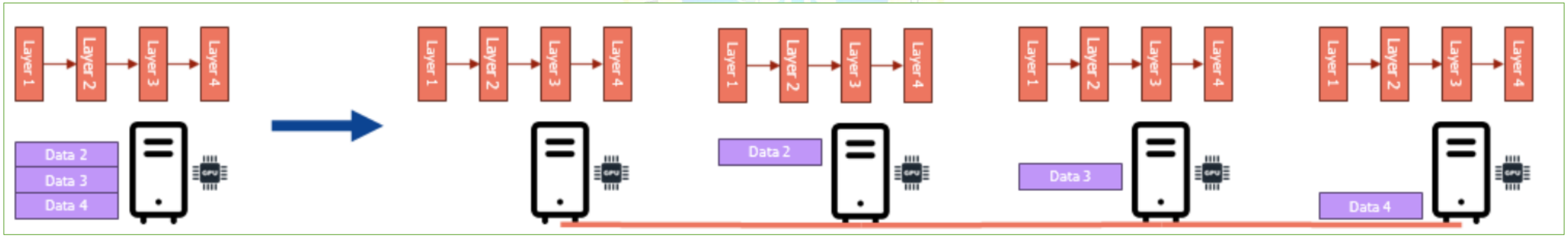
1. **Forward propagation**: apply model to a batch of input samples and run calculation through operators to produce a prediction
2. **Backward propagation**: run the model in reverse to produce a gradient for each trainable weight
3. **Weight update**: use the gradients to update model weights

$$w_i := w_i - \gamma \frac{\partial L(w)}{\partial w_i} = w_i - \frac{\gamma}{n} \sum_{j=1}^{n} \frac{\partial l_i(w)}{\partial w_i}$$

Gradients of individual samples

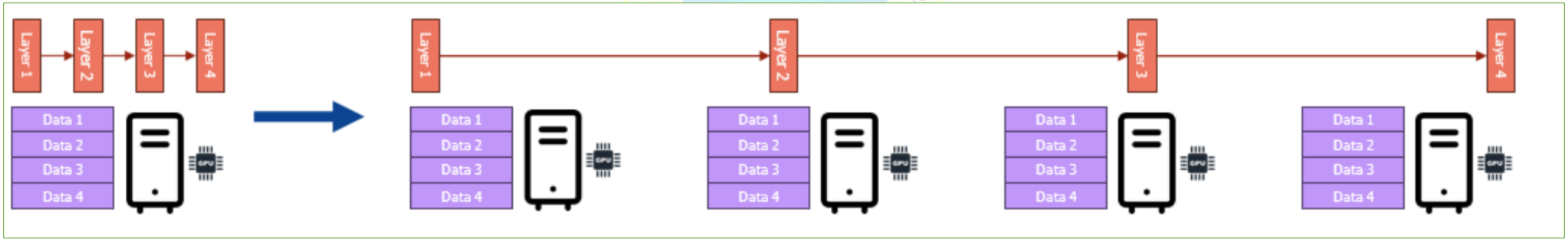
Data Parallelism

If the model can fit within a single GPU, then we can distribute the training on multiple servers (each containing one or multiple GPUs), with each GPU processing a subset of the entire dataset in parallel and synchronizing the gradients during backpropagation. This option is known as Data Parallelism.



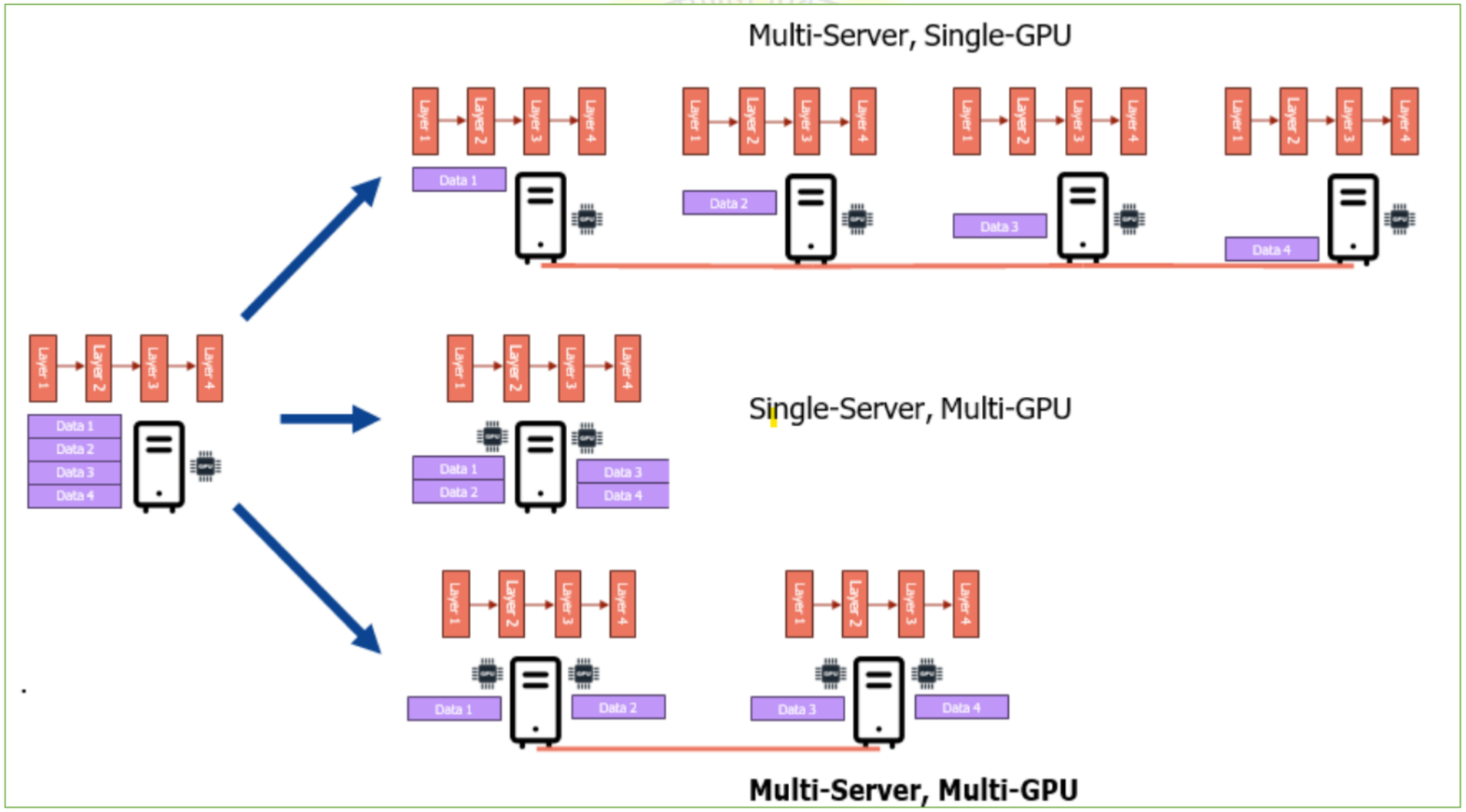
Model Parallelism

If the model cannot fit within a single GPU, then we need to “break” the model into smaller layers and let each GPU process a part of the forward/backward step during gradient descent. This option is known as Model Parallelism.



Distributed Data Parallel in detail

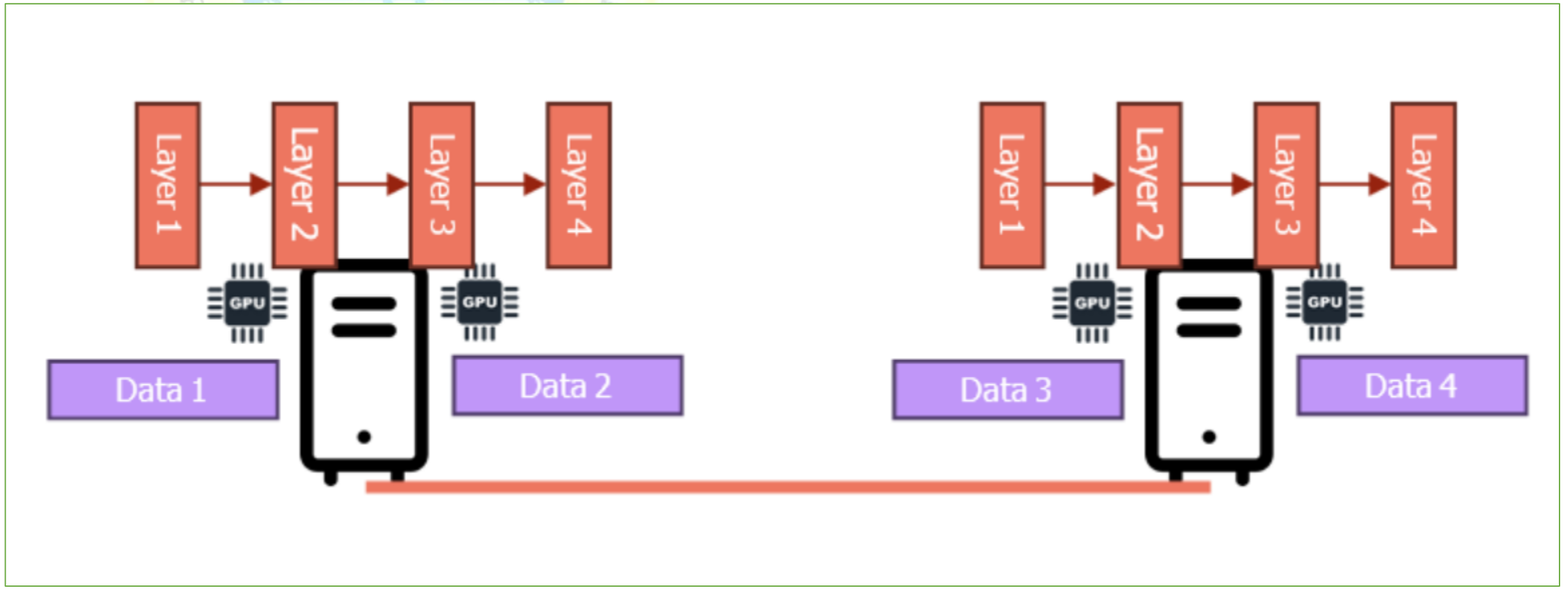
Imagine you have a training script that is running on a single computer/GPU, but it's very slow, because: the dataset is big and you can't use a big batch size as it will result in an **Out Of Memory error on CUDA**. Distributed Data Parallel is the solution in this case. It works in the following scenarios:



Distributed Data Parallel in detail

Distributed Data Parallel works in the following way:

1. At the beginning of the training, the model's weights are initialized on one node and sent to all the other nodes (**Broadcast**)
2. Each node trains the same model (with the same initial weights) on a subset of the dataset.
3. Every few batches, the gradients of each node are accumulated on one node (summed up), and then sent back to all the other nodes (**All-Reduce**).
4. Each node updates the parameters of its local model with the gradients received using its own optimizer
5. Go back to step 2

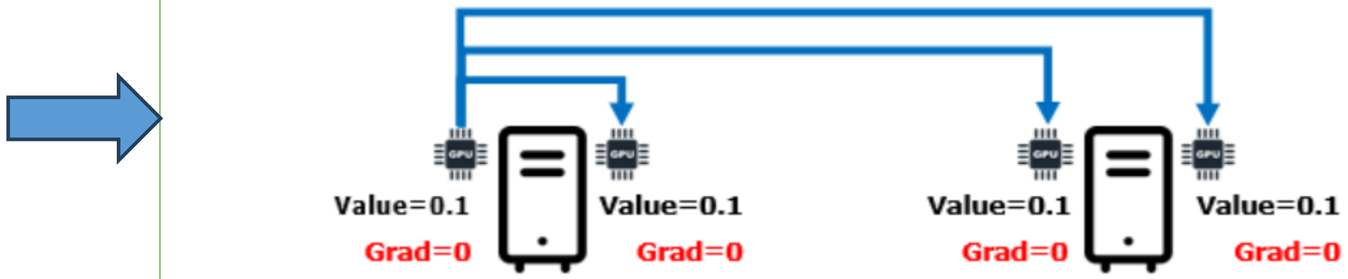


Distributed Data Parallel in detail

Distributed Data Parallel: step 1

Model weights are initialized here (e.g., randomly)

Initial weights are sent to all the other nodes (Broadcast)

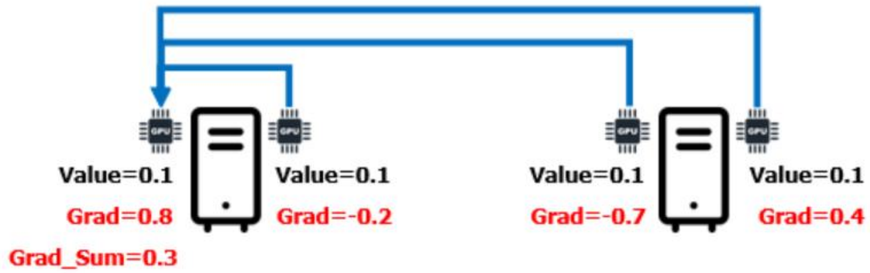


Distributed Data Parallel in detail

Distributed Data Parallel: step 2

Each node runs a forward and backward step on one or more batch of data. This will result in a local gradient. The local gradient may be the accumulation of one or more batches

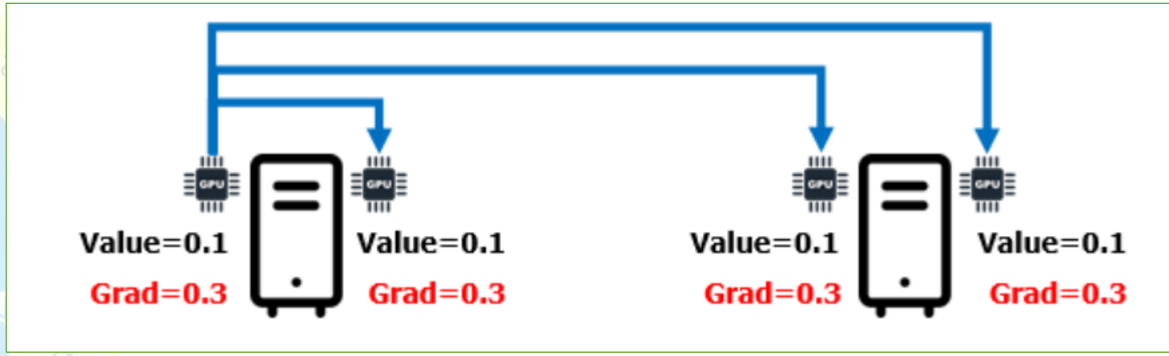
The sum of all the gradients is cumulated on one node (Reduce)



Distributed Data Parallel in detail

Distributed Data Parallel: step 3

The cumulative gradient is sent to all the other nodes (Broadcast).The sequence of Reduce and Broadcast are implemented as a single operation (All-Reduce).



Distributed Data Parallel: step 4

Each node updates the parameters of its local model using the gradient received.
After the update, the gradients are reset to zero and we can start another loop.

Collective Communication Primitives

In distributed computing environments, a node may need to communicate with other nodes. If the communication pattern is similar to a client and a server, then we talk about point-to-point communication, because one client connects to one server in a request- response chain of events.

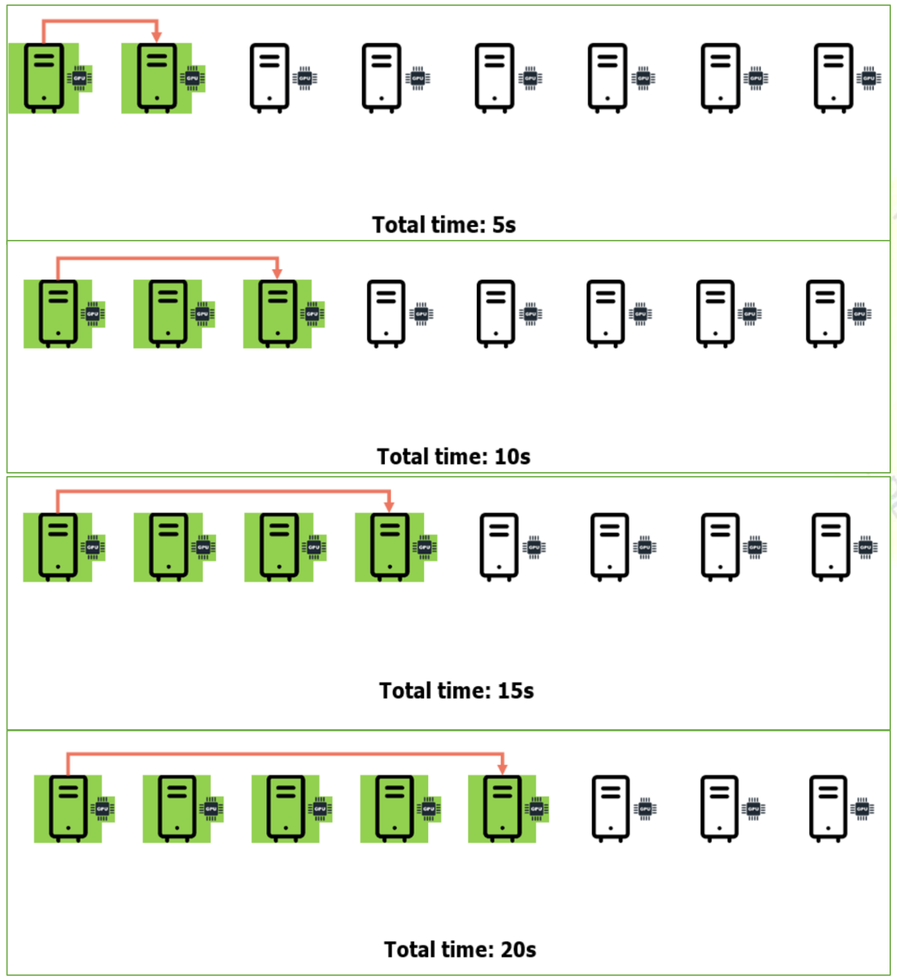
However, there are cases in which one node needs to communicate to multiple receivers at once: this is the typical case of data parallel training in deep learning: one node needs to send the initial weights to all the other nodes.

Moreover, all the other nodes, need to send their gradients to one single node and receive back the cumulative gradient. **Collective communication allows to model the communication pattern between groups of nodes.**

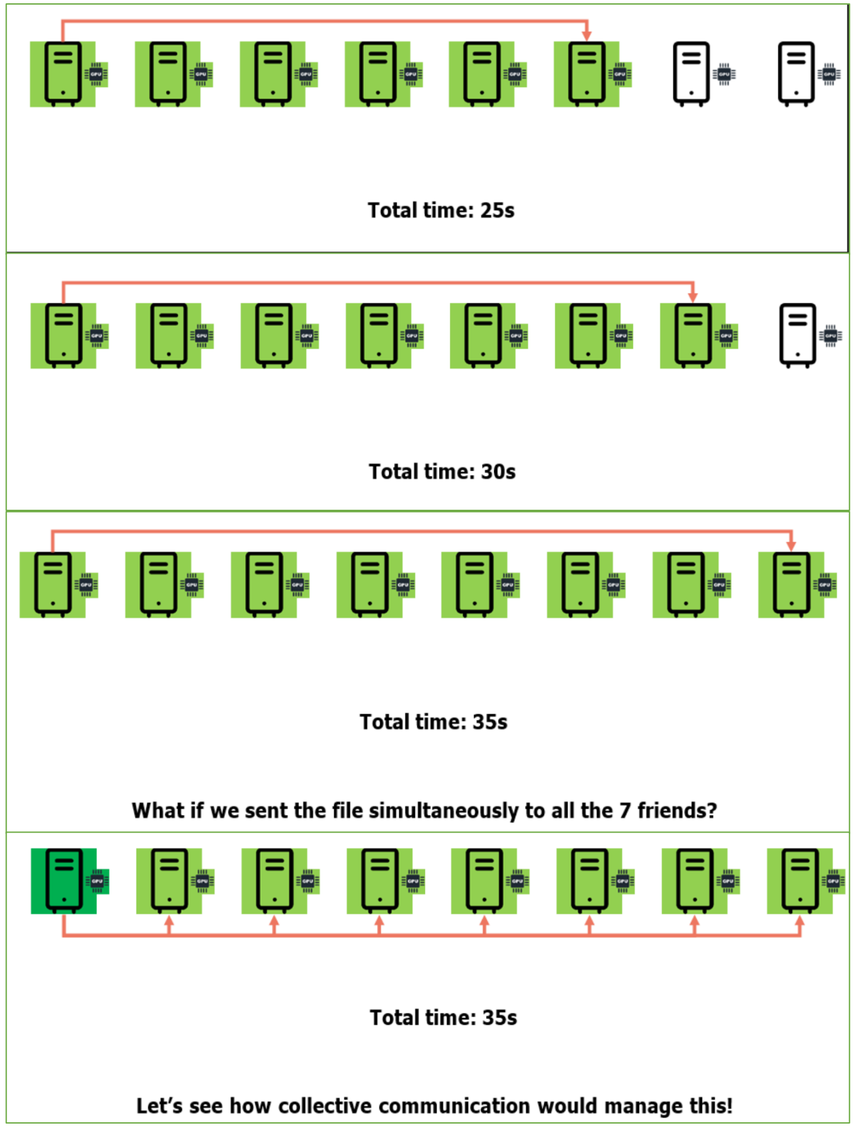
Let’s visualize the difference between the two modes of communication.

Point-To-Point

Imagine you need to send a file to 7 friends. With point-to-point communication, you'd send the file iteratively to each of the friend one by one. Suppose the internet speed is 1 MB/s and the file is 5 MB in size

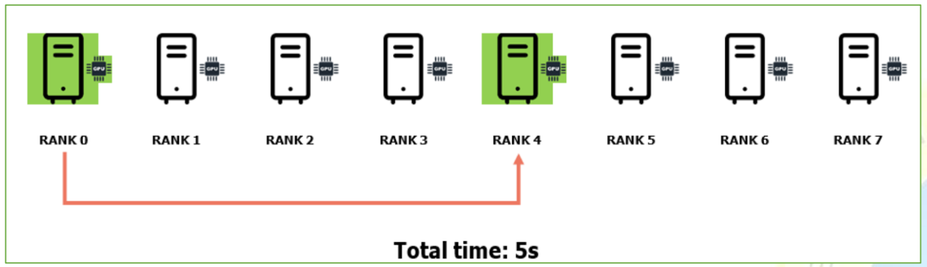


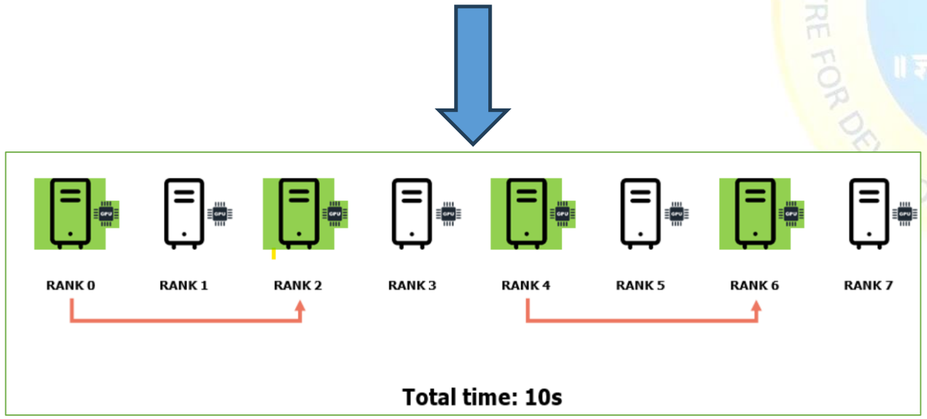
Since the internet communication is 1 MB/s and the file is 5 MB in size, your connection would be split among the 7 friends (each friend would be receiving the file at ~ 143 KB/s). The total time is still 35s.

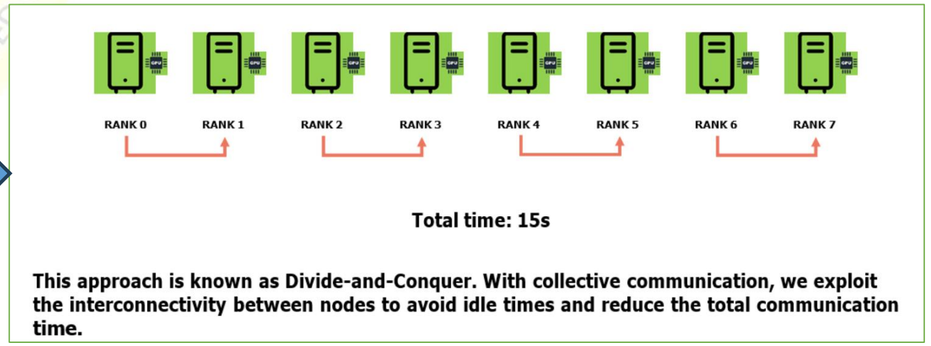


Broadcast

The operation of sending a data to all the other nodes is known as the Broadcast operator. Collective Communication libraries (e.g. NCCL) assign a unique ID to each node, known as RANK. Suppose we want to send 5 MB with an internet speed of 1 MB/s.

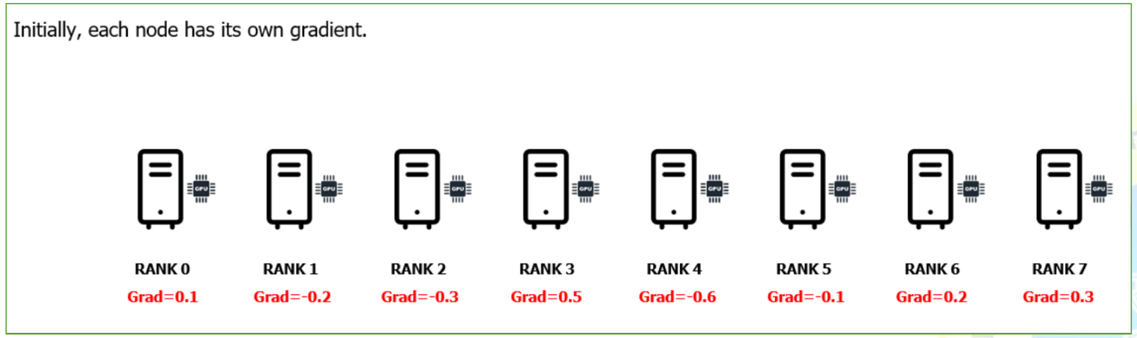


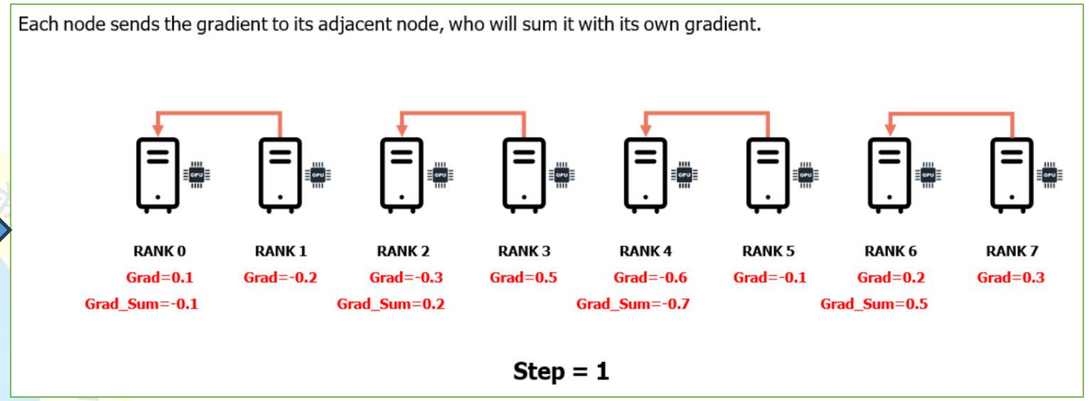


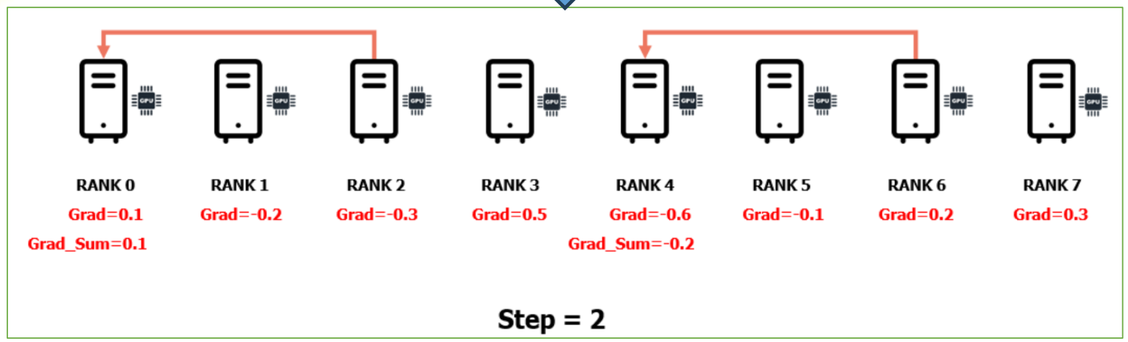


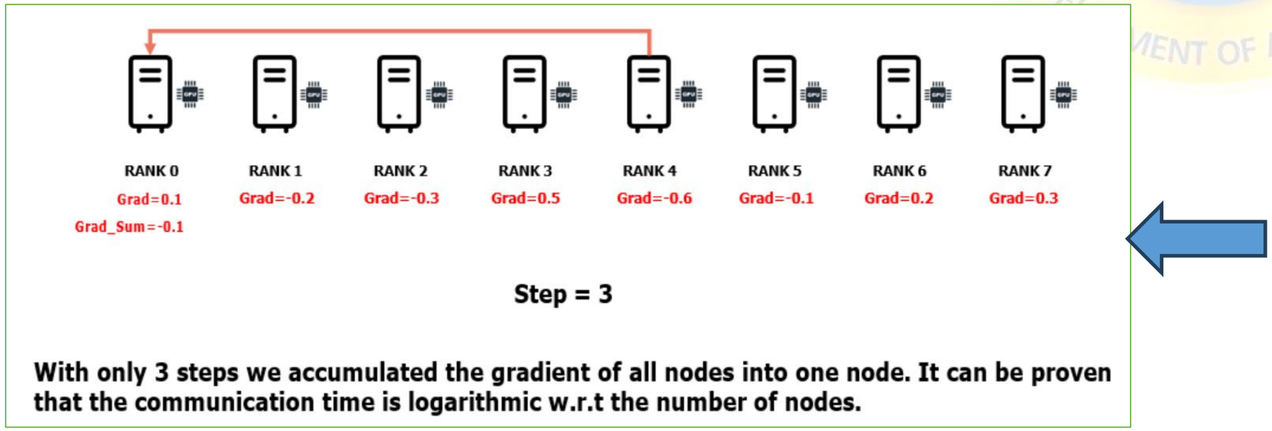
Broadcast

The Broadcast operator is used to send the initial weights to all the other nodes when we start the training loop.
At every few batches of data processed by each node, the gradients of all nodes need to be sent to one node and accumulated (summed up). This operation is known as Reduce.









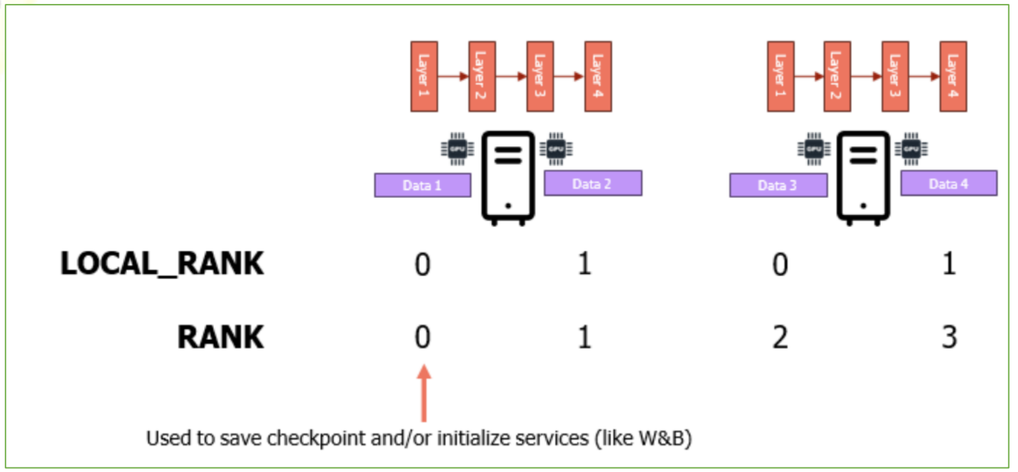
Collective Communication: All-Reduce

Having accumulated the gradients of all the nodes into a single node, we need to send the cumulative gradient to all the nodes. This operation can be done using a Broadcast operator. The sequence of Reduce-Broadcast is implemented by another operator known as All-Reduce, whose runtime is generally lower than the sequence of Reduce followed by a Broadcast.

I will not show the algorithm behind All-Reduce, but you can think of it as a sequence of Reduce followed by a Broadcast operation

LOCAL_RANK vs RANK

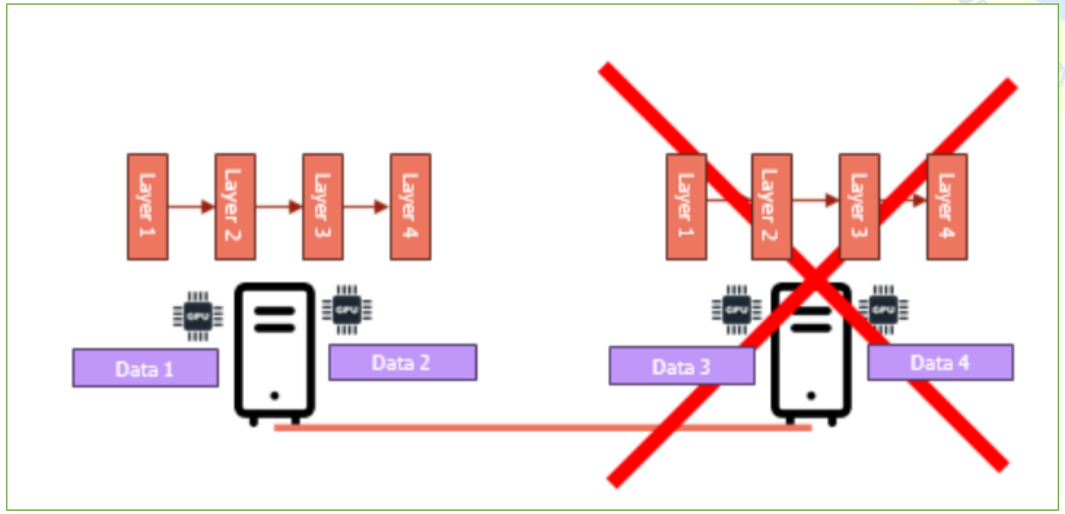
The environment variable LOCAL_RANK indicates the ID of the GPU on the local computer, while the RANK variable indicates the a globally unique ID among all the nodes in the cluster.
Please note that ranks are not stable, meaning that if you restart the entire cluster, a different node may be assigned the rank number 0.

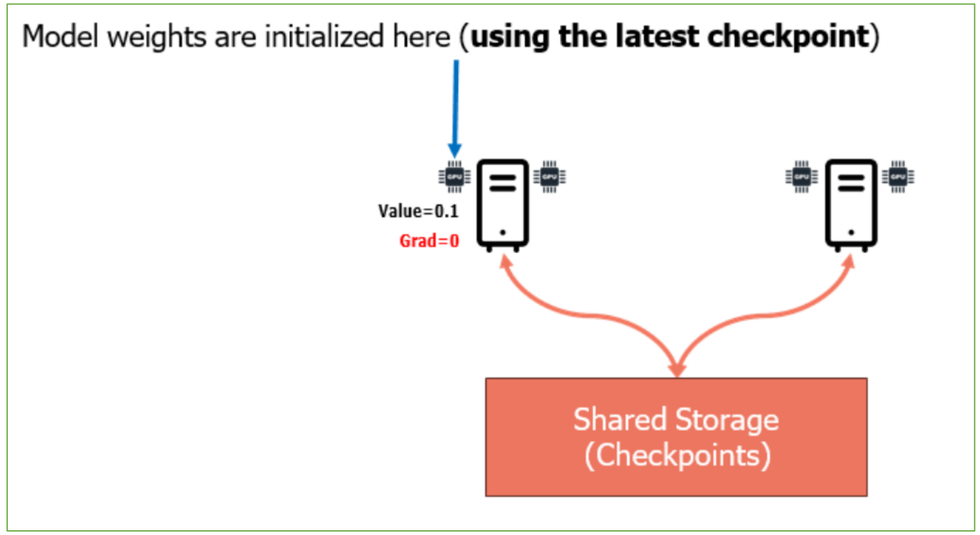


Failover: what happens if one node crashes?

Imagine you're training in a distributed scenario like the one shown below and one of the nodes suddenly crashes. In these case, 2 GPUs out of 4 become unreachable. How should the system react?

One way, would be to restart the entire cluster and that's easy. However, by restarting the cluster, the training would restart from zero, and we would lose all the parameters and computation done so far. A better approach is to use checkpointing.
Checkpointing means saving the weights of the model on a shared disk every few iterations (for example every epoch) and
resume the training from the last checkpoint in case there's a crash

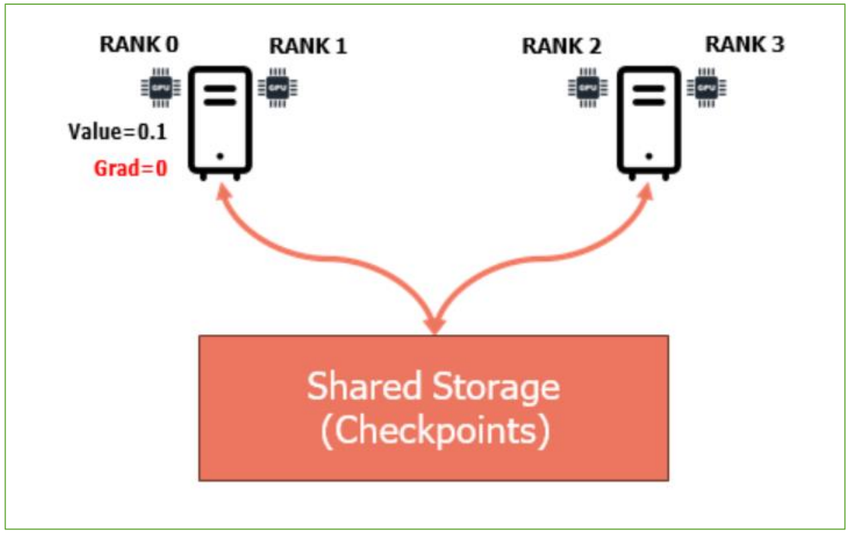




Failover: what happens if one node crashes?

We need a shared storage because PyTorch will decide which node will initialize the weights and we should make no assumption on which one will it be. So, every node should have access to the shared storage. Plus, it is good rule in distributed systems to not have one node more important than others, because every node can fail at any time.

When we start the cluster, PyTorch will assign a unique ID (RANK) to each GPU. We will write our code in such a way that whichever node is assigned the RANK 0 will be responsible for saving the checkpoint, so that the other nodes do not overwrite each other's files. So only one node will be responsible for writing the checkpoints and all the other files we need for training.



Issues with Data Parallelism

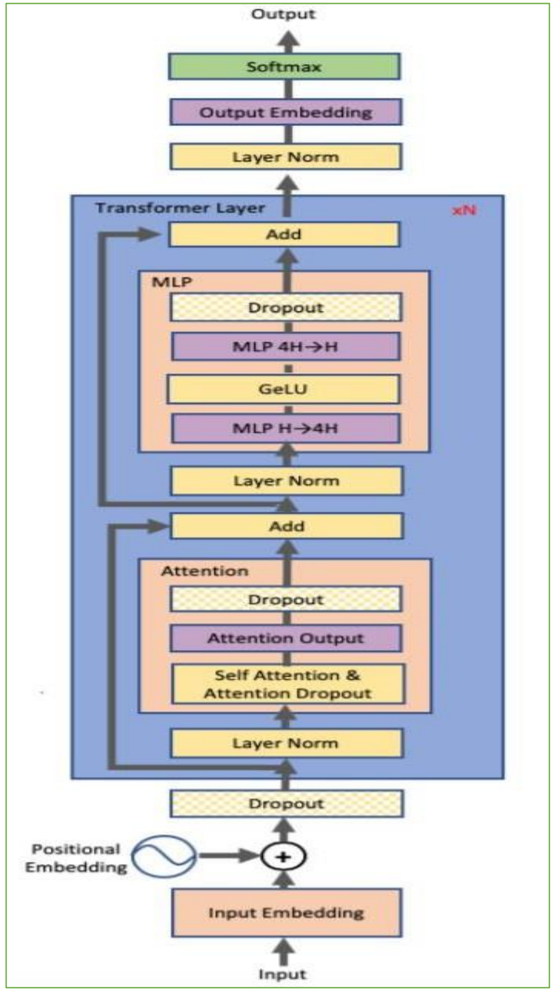
- Replication of Model Across GPUs
  - Each GPU holds a complete copy of the model. This replication consumes memory on all GPUs.
  - Example: If a model is 2GB in size, and there are 4 GPUs, 2GB of memory on each GPU will be occupied to store the same model.
- Limited by GPU Memory
  - Since each GPU needs to store the entire model, the maximum model size that can be trained is constrained by the memory capacity of the smallest GPU in the system.
  - Large models with millions or billions of parameters, such as GPT-3 or large vision transformers, can easily exceed the memory of a single GPU, making training infeasible with pure data parallelism.
- Communication Overhead
  - After each forward and backward pass, GPUs must synchronize their gradients with each other. This requires significant communication bandwidth, especially when using a large number of GPUs.

Large Model Training Challenges

	Bert-Large	GPT-2	Turing 17.2 NLG	GPT-3
Parameters	0.32B	1.5B	17.2B	175B
Layers	24	48	78	96
Hidden Dimension	1024	1600	4256	12288
Relative Computation	1x	4.7x	**54x**	**547x**
Memory Footprint	5.12GB	24GB	**275GB**	**2800GB**

NVIDIA V100 GPU memory capacity: 16G/32G
NVIDIA A100 GPU memory capacity: 40G/80G

Out of Memory



Model Parallelism

Model Parallelism

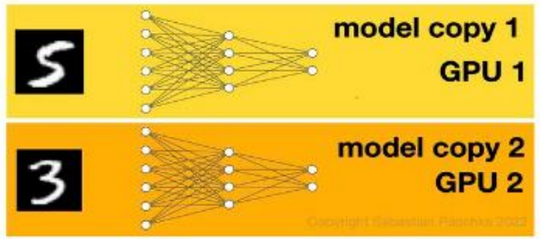
Model parallelism is a strategy for distributing the computation of a deep learning model across multiple GPUs or other computing devices. This approach is particularly beneficial when the model is too large to fit into the memory of a single GPU. Instead of duplicating the entire model on each GPU (as in data parallelism), different parts of the model are placed on different GPUs, allowing for the training of very large models.

2) Model parallelism:

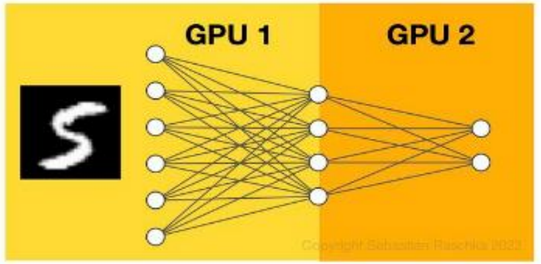
Divide model across separate GPUs.

Usually, this helps with limited VRAM. However, note that this doesn't imply that the training happens in parallel!

How? Just put individual layers onto different devices, e.g., `layer1.to('cuda:0')`, `layer2.to('cuda:1')` etc.



Data parallelism



Model parallelism

Issues with Model Parallelism

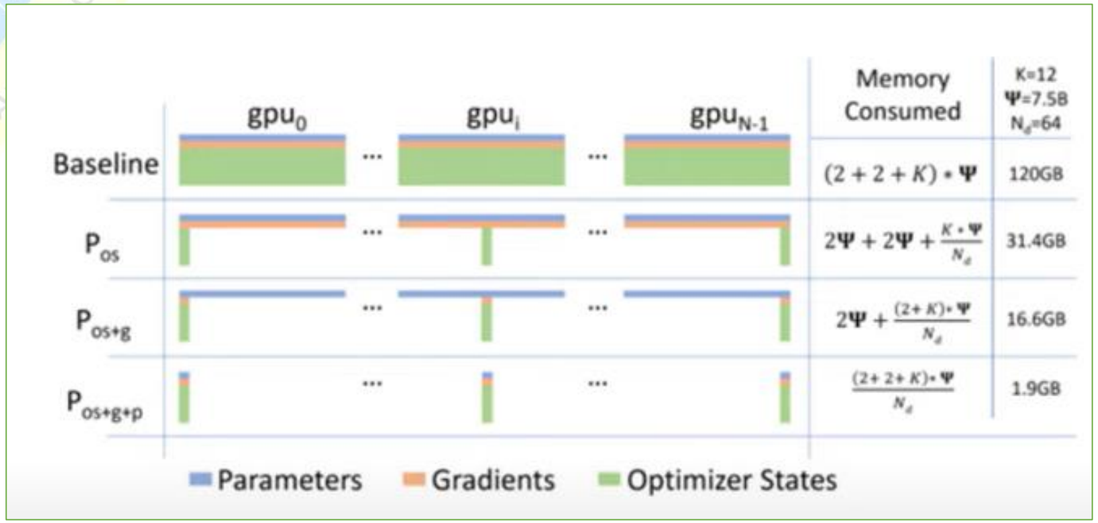
- ✓ **Sequential Dependency:** GPUs must wait for previous layers to finish, leading to idle time and inefficient utilization.
- ✓ **High Communication Overhead:** Constant data transfer of activations between GPUs slows down training.
- ✓ **Load Imbalance:** Uneven partitioning of layers causes some GPUs to finish tasks earlier, reducing efficiency.
- ✓ **Memory Bottlenecks:** Intermediate activations still consume significant memory on GPUs.
- ✓ **Complex Implementation:** Requires manual model splitting and careful synchronization.
- ✓ **Limited Scalability:** Adding more GPUs doesn’t always yield better performance due to dependencies and communication delays.

Hybrid (Model and Data) Parallelism

Fully Sharded Data Parallel (FSDP)

Fully Sharded Data Parallel (FSDP) is a feature in PyTorch that addresses the limitations of data parallelism by allowing more efficient utilization of memory and compute resources. FSDP shards (i.e., splits) both the model parameters and optimizer states across all available GPUs, significantly reducing memory usage and enabling the training of very large models that otherwise wouldn't fit into GPU memory.

Fully Sharded Data Parallel (FSDP) is primarily a form of model parallelism. However, it incorporates elements of data parallelism as well.



GPT-2 Model

GPT 2

- Context length: 1024
- Parameter size: 1.5 Billion
- Batch Size: 32
- Precision size: 32 Bits/4 Bytes

Calculate memory requirement

Params size = num params x precision = 4 x 1.5 = 6 Billion Bytes ~ 6 GB

Gradient size = num params x precision = 4 x 1.5 = 6 Billion Bytes ~ 6 GB

Adam = num params x precision x 3 ~ 18 GB

Token Embeddings = batch_size x sequence_length x embedding_dim x precision_size ~ 0.19 GB

Positional Embeddings = sequence_length x embedding_dim x precision_size ~ 0.006 GB

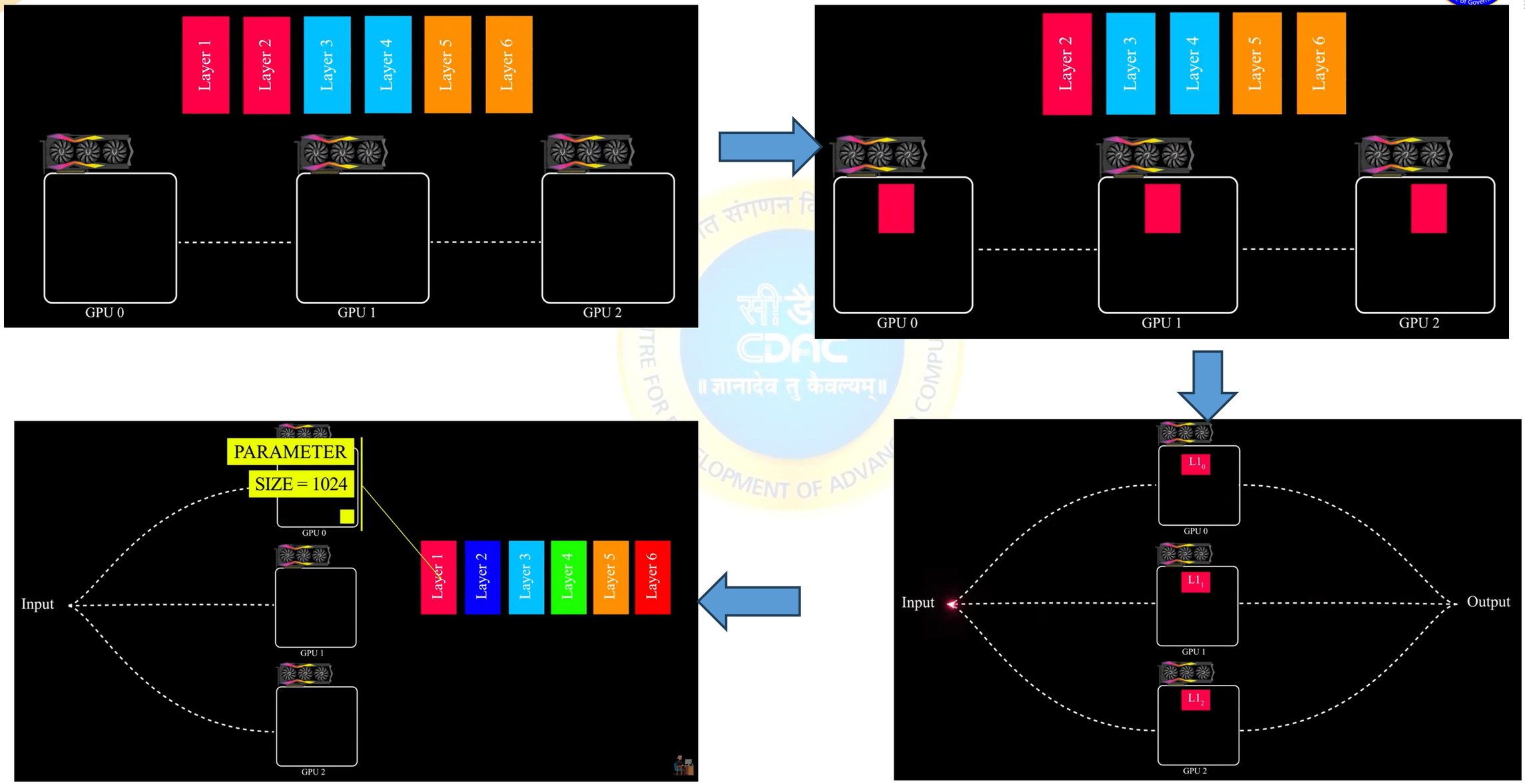
Layer Activation = Self attention + Feed Forward ~ 1.17 GB = 1.17 GB x 48 = 56.25 GB

Total activation memory = Token Embeddings + Positional Embeddings + Layer Activations + Layer Normalization ≈ 56.43 GB

Total GPU Memory = Parameters + Gradients + Optimizer state + Activations

= 6 + 6 + 18 + 56.43 = 86.43 ≈ 87 GB

Fully Sharded Data Parallel (FSDP)

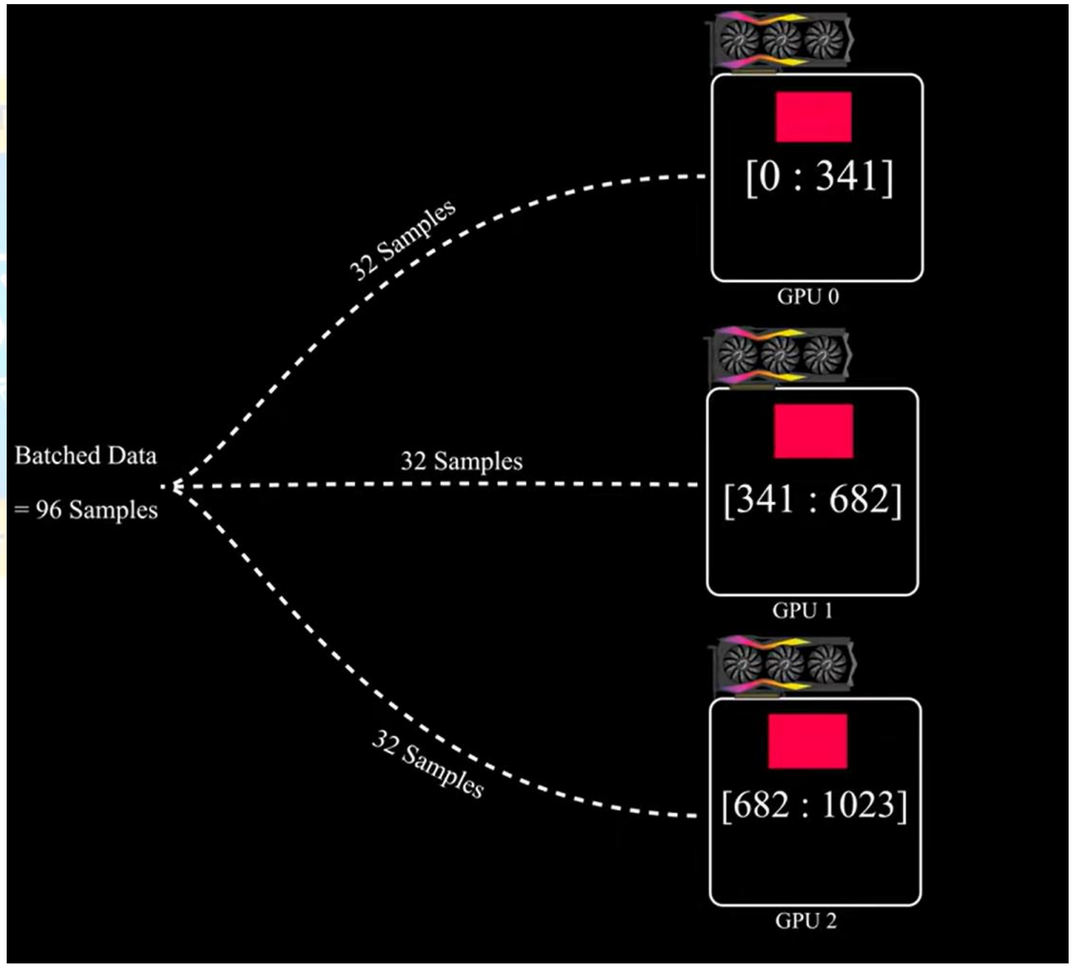


Fully Sharded Data Parallel (FSDP) in detail

Step 1: Send batch across GPUs

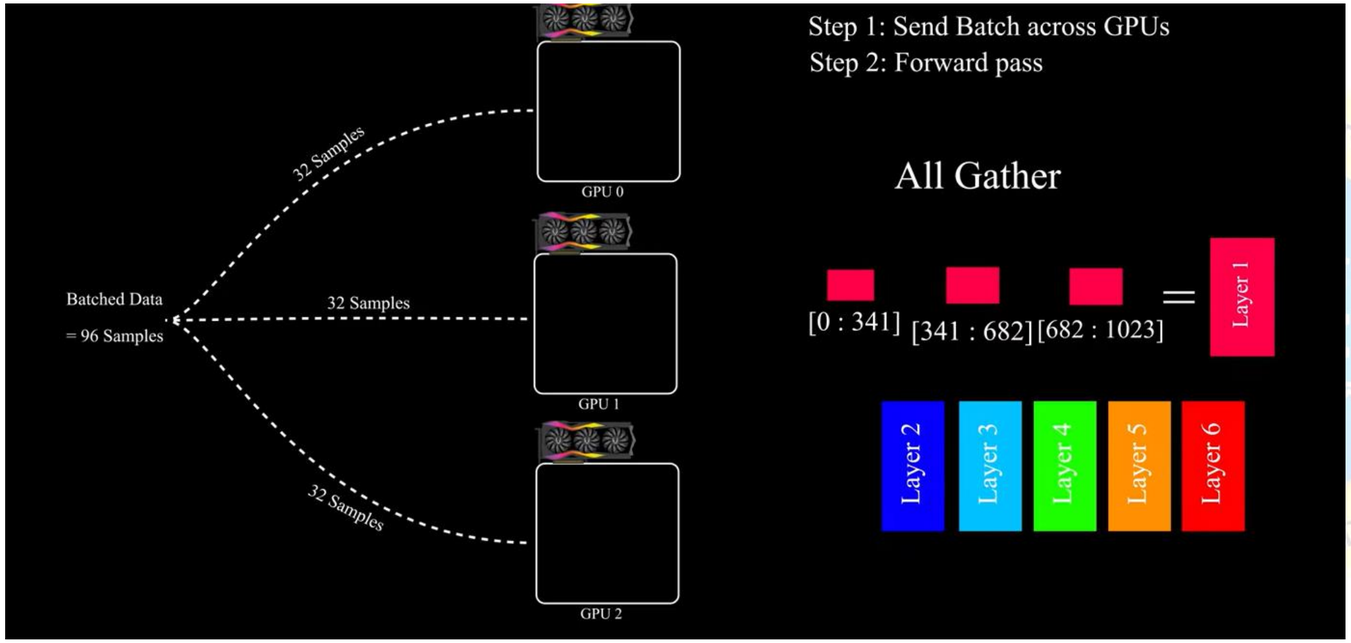
A large batch of data is divided into smaller chunks.
Each GPU processes only a subset of the batch.

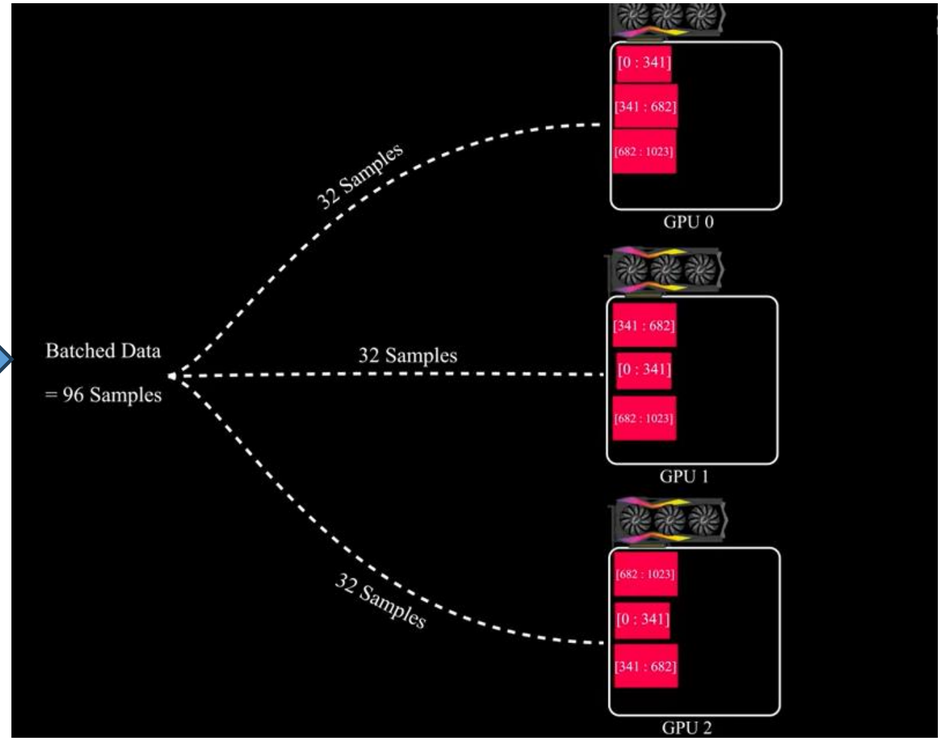
GPU 0 processes samples from indices [0:341].
GPU 1 processes samples from indices [341:682].
GPU 2 processes samples from indices [682:1023].



Fully Sharded Data Parallel (FSDP) in detail

Step 2 : Forward Pass



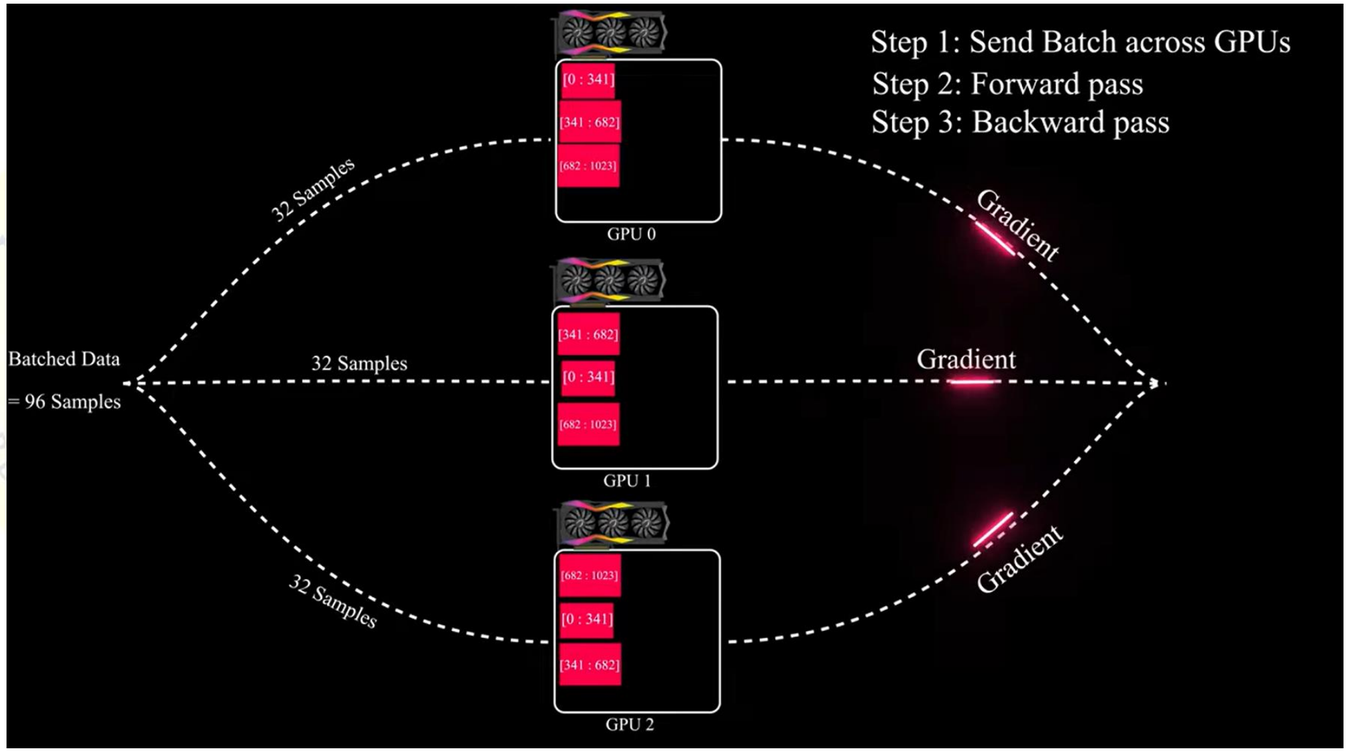


During the forward pass, GPUs collaboratively perform an All-Gather operation to retrieve required model shards from other GPUs, enabling parallel computation while optimizing memory usage by leveraging sharded model parameters and activations, thereby facilitating the training of large models that exceed single GPU memory capacity.

Fully Sharded Data Parallel (FSDP) in detail

Step 3 : Backward Pass

During the backward pass, each GPU computes local gradients for its assigned data, shards the gradients to reduce memory usage, and performs an All-Reduce operation to aggregate and synchronize gradients across GPUs, ensuring efficient and consistent model updates in a distributed training setup.

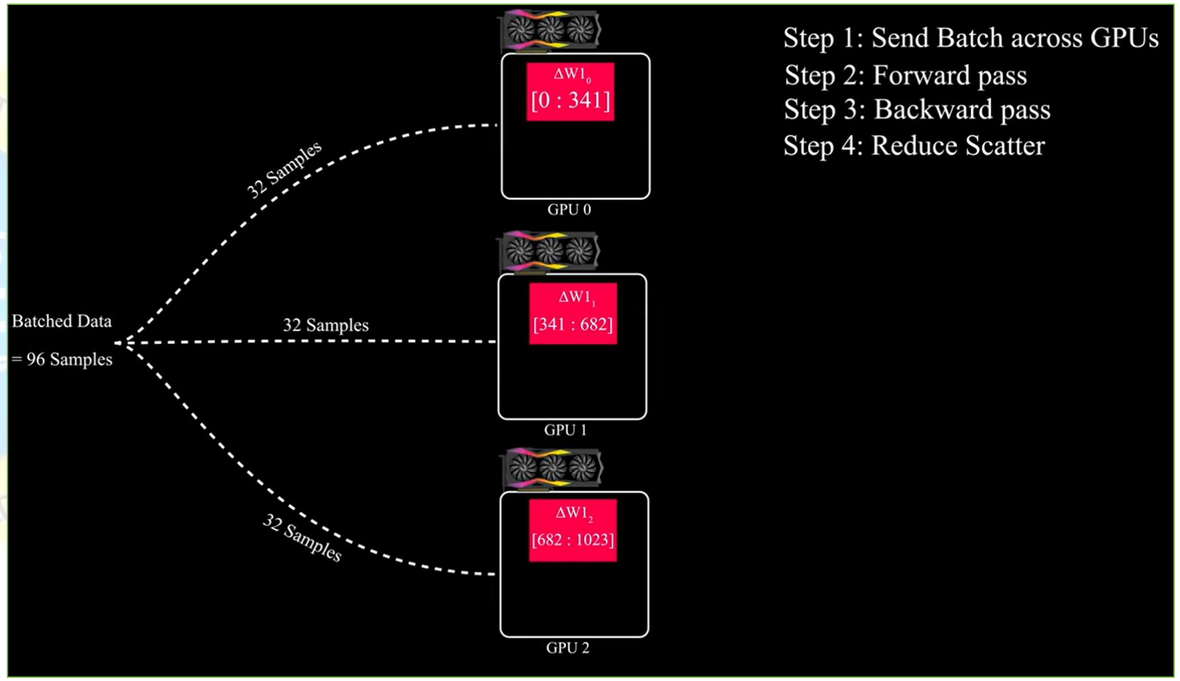


Fully Sharded Data Parallel (FSDP) in detail

Step 4 : Reduce Scatter

Begins by combining (reducing) the gradients from all GPUs, which ensures that the global gradient for each parameter is computed. Instead of each GPU receiving the full global gradient (as in an All-Reduce operation), the Reduce Scatter operation distributes (scatters) only the relevant portion of the global gradient to each GPU.

By combining the reduction and scattering into a single operation, this step minimizes both memory consumption and communication overhead, making it highly efficient for distributed training. The Reduce Scatter operation ensures synchronization across GPUs while maintaining the scalability required for large-scale models.




Thank You
प्रशंसावादः
धन्यवाद
આભાર