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

12: Parallelization-2, Collective Communication

Lecturer: Hao Zhang

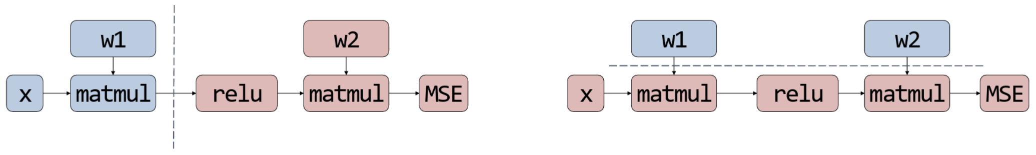
Scribe(s): Amogh Patankar, Oren Ciolli, Luca Labardini, Jahnavi Patel, Ruoxin Xiao, Weixiao Zhan, Pranav Kumar, Kiran Medleri Hiremath, Krishan Ponnaganti, Archana Pradeep

Scribe List: Amogh Patankar, Oren Ciolli, Luca Labardini, Jahnavi Patel, Ruoxin Xiao, Weixiao Zhan, Pranav Kumar, Kiran Medleri Hiremath, Krishan Ponnaganti, Archana Pradeep

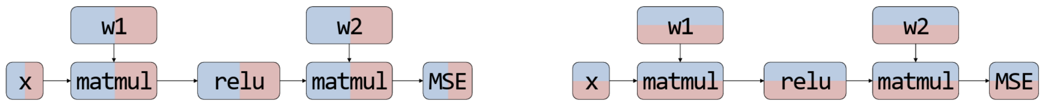
1 Parallelism Overview

Inter-op and Intra-op parallelism are the two types

Inter-op assigns different operators to different devices



Intra-op assigns different regions of a single operator to multiple devices



1.1 ML Parallelism Paradigm

The terms Data and Model parallelism are not as clear. They are defined with examples below.

1.1.1 Data Parallelism

1. **Distributing Data Batches Across Devices:** Each device processes a different subset (mini-batch) of the input data but uses the same model.
2. **Weight Synchronization:** After computing gradients on each device, they are aggregated, and updated weights are synchronized across all devices.
3. **Applicable for Fully-Connected or Convolutional Layers:** Works well when the model size is manageable, and the focus is on large datasets.

4. **Reduces Time Per Batch:** By splitting data, it speeds up training without changing the model structure.
5. **Examples:** TensorFlow's MirroredStrategy, PyTorch's DataParallel, and DistributedDataParallel (DDP).

1.2 Model Parallelism

1. **Splitting the Model Across Devices:** Different parts of the model are placed on different devices. Each device computes a portion of the forward and backward passes.
2. **Layer-wise Partitioning:** Commonly used for models that are too large to fit into a single device's memory. For example, placing different layers of a neural network on different GPUs.
3. **Pipeline Parallelism:** Extends layer-wise partitioning by executing partitions in a pipelined manner to improve efficiency.
4. **Tensor (Sharded) Parallelism:** Splits large tensors (e.g., weights) across multiple devices, allowing parallel computation of individual operations.
5. **Examples:** Megatron-LM's tensor parallelism, DeepSpeed's ZeRO, and FairScale's pipeline parallelism.

1.3 New ML Parallelism Paradigm

The terms Inter-op Parallelism and Intra-op Parallelism are used to provide a clearer and more structured understanding of parallelism, focusing on the computational graph and device cluster rather than a simple data vs. model distinction. This classification facilitates the development of more efficient parallelism strategies.

1.3.1 Inter-Op Parallelism

As a whole, Inter-Op parallelism splits operators amongst devices. Thinking in terms of a computational graph, Inter-Op parallelism "cuts" the computational graph itself, splitting pieces between devices. This type of parallelism uses P2P (Point-to-point) communication where GPU's almost sequentially do some work and pass on results to the next worker.

Benefits of Inter-Op parallelism include:

1. **Parallel Execution of Independent Operations:** Different operations (nodes) in the computational graph that have no dependencies are executed simultaneously on different devices.
2. **Graph-Level Parallelism:** Focuses on scheduling tasks from the computational graph across multiple devices, improving overall throughput.
3. **Efficient for Multi-GPU Systems:** Allows better utilization of all available hardware resources by distributing independent tasks.
4. **Reduces Latency for Complex Graphs:** This type of parallelism performs well when models have multiple parallel and independent sub-graphs. Thus, it is especially useful for neural networks with multiple branches or parallel components (e.g., ResNet, Inception models).

5. **Examples:** TensorFlow's XLA Compiler and optimized task schedulers for deep learning frameworks.

Some trade-offs of this approach include:

1. **Device Idling:** This use of P2P communication can result in some devices idling while they wait for others to complete.
2. **Device Communication:** On the other hand, this approach results in less device communication than Intra-Op parallelism, avoiding this pitfall.

1.3.2 Intra-Op Parallelism

As a whole, Intra-Op parallelism splits different parts of an operator amongst different devices. In other words, Intra-Op parallelism "cuts" and splits the operator rather than splitting the computational graph itself. Additionally, this type of parallelism uses Collective Communication where multiple workers are constantly exchanging information in order to incrementally complete the task. This exchange of information between devices (also known as all reduce) gets more expensive as we use more and more devices.

Some benefits of Intra-Op parallelism include:

1. **Parallel Execution Within a Single Operation:** Large operations, such as matrix multiplications or convolution layers, are parallelized across multiple devices.
2. **Focus on Single Task Acceleration:** Optimizes the execution of individual operations to reduce their runtime.
3. **Tensor Partitioning and Sharding:** Splits large tensors across multiple devices for simultaneous computation.
4. **Applicable for Large-Scale Linear Algebra Operations:** Particularly beneficial in deep learning models with large parameter matrices.
5. **Device Utilization:** This approach results in high device utilization as each device will generally have a part of an operator to work on.
6. **Examples:** CUDA-based parallelism for matrix operations, MKL-DNN, and cuDNN optimizations.

Some trade-offs of this approach include:

1. **Collective Communication:** Each device needs to send its partial result to every other device. Following this, each device must also aggregate or reduce this information. This operation has about the number of devices squared complexity. Consequently, this communication is quite expensive.
2. **Device Load:** On the other hand, device utilization is higher than in Inter-Op parallelism.

1.4 Model Flops Utilization (MFU)

1.4.1 How to Measure Efficiency of Parallelism?

As we all know we can compute the efficiency by measuring the arithmetic intensity. Arithmetic intensity helps us to understand how efficiently computations are being performed, relative to the amount of data transfer. However AI is usually an effective measure on a single operator level.

$$\mathbf{AI} = \frac{\#ops}{\#bytes}$$

where

$$\mathbf{\#ops} = Numberofoperations$$

$$\mathbf{\#bytes} = Numberofbytestransferred$$

1.4.2 Model Flops Utilization (MFU)

We introduce MFU (Model Flops Utilization) since we need a more global and holistic definition that can be used for calculations across setups with 1000s of GPUs. Intuitively, it is useful to understand how much percent of peak utilization of GPUs we are able to achieve. The formula for calculating MFU is

$$\mathbf{MFU} = \frac{\#Flops/t}{peakFLOPS}$$

where

$$\mathbf{\#FLOPs} : Totalfloating - pointoperationsperformedbytheMLprogram$$

$$\mathbf{t} : Timetakentocompletetheprogram$$

$$\mathbf{peak\ FLOPS} : ThemaximumtheoreticalFLOPsthehardwarecanperform$$

MFU performs the calculation by considering the workload of the computation as well as the hardware's capabilities. Using MFU we can identify if the hardware is being under utilized and can optimize the system to achieve higher utilization. MFU values close to the peak FLOPS can only be achieved if we use only GEMM operations and almost no MFU-unfriendly operations.

1.4.3 Potential Factors Affecting MFU

Some potential factors that can affect MFU include

- **Op Types in the Computational Graph**: The type and shape of operations in the ML model can affect MFU. For example, Different operations (e.g., matrix multiplication, convolution) have different FLOPs which can affect MFU.
- **Precision, Core, and GPU Type**: The hardware's precision (e.g., FP32, FP16), core count, and GPU type (e.g., V100, A100, H100)influence peak FLOPs and thus MFU.
- **Communication Over Network**: Network communication can introduce delays, reducing MFU.
- **Optimizations**:Poorly optimized code or inefficient use of hardware resources can lower MFU. Techniques like precision reduction, core utilization,memory-efficient kernels, better scheduling and GPU type selection can impact MFU.

MFU-Friendly and MFU-Unfriendly Operations

- **MFU-friendly operations**: Operations that maximize FLOPs utilization, such as matrix multiplications (matmul) with high arithmetic intensity.
- **MFU-unfriendly operations**: Operations that involve more data movement or memory-bound than computation, such as element-wise operations (e.g., ReLU) or data shuffling.

Reducing Communication Overhead
Techniques like overlapping computation with communication (e.g., using asynchronous communication) or optimizing data transfer (e.g., reducing message size) can help reduce or hide communication overhead.

1.4.4 MFU Estimation

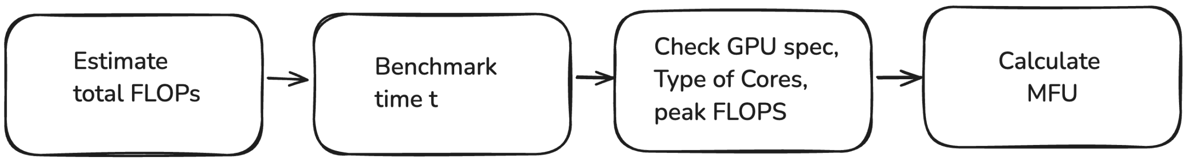


Figure 1: Steps to Calculate MFU

1.4.5 Example MFU Estimation

Suppose:

- $\#FLOPs = 10^{12}$ (1 trillion FLOPs).
- $t = 10$ seconds.
- $peakFLOPS = 10^{13}$ (10 trillion FLOPs per second).

The MFU is calculated as:

$$MFU = \frac{10^{12}/10}{10^{13}} = 0.01 or 1\%$$

This means the hardware is only 1% utilized, indicating it is underutilized and must be optimized further.

1.5 MFU in the LLM Industry

MFU is becoming a widely indexed metric in the LLM industry for benchmarking the efficiency of different hardware and optimization techniques.

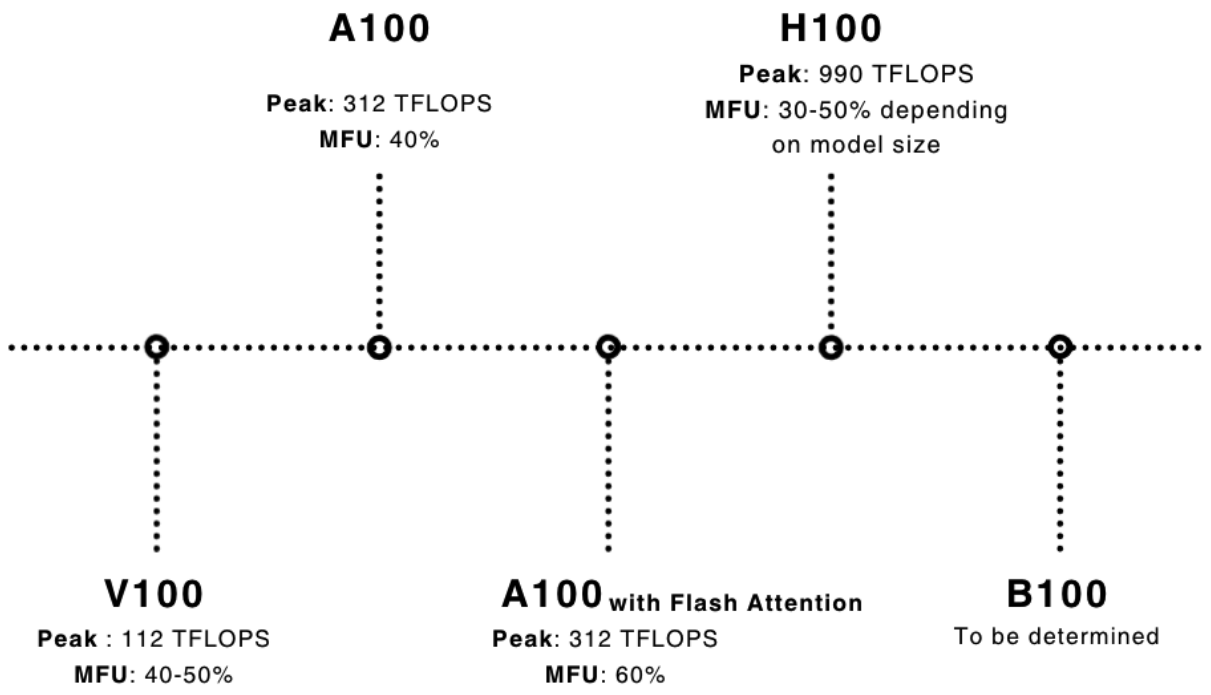


Figure 2: Typical MFU values for different GPUs

1.6 MFU vs. HFU (Hardware Flops Utilization)

Model Flops Utilization (MFU)	**Hardware Flops Utilization (HFU)**
MFU measures how efficiently the model uses the hardware's computational capacity	HFU measures the actual utilization of hardware's peak performance, considering all active processes and overheads.
$MFU = \frac{EstimatedAchievedFLOPs}{PeakTheoreticalFLOPS}$	$HFU = \frac{ActualAchievedFLOPs}{PeakTheoreticalFLOPS}$

1.6.1 When is MFU != HFU?

MFU (Model Flop Utilization) and HFU (Hardware Flop Utilization) are typically assumed to be equal. However, when using certain memory-saving techniques like gradient checkpointing, this assumption no longer holds. There can also be some overhead beyond the pure model computation. Gradient checkpointing reduces memory usage by recomputing intermediate activations during backpropagation. While this saves memory, it introduces additional computations that do not directly contribute to model convergence. As a result, HFU can appear higher than MFU in such scenarios.

The computation calculations are as follows:

Without Checkpointing:

- Forward Pass: 1 pass
- Backward Pass: 1 pass
- Total Equivalent Forward Passes: In most neural network models, a backward pass involves approximately twice the computation of a forward pass because it involves the calculation of gradients with respect to both parameters and activations.

  Therefore, 1 backward pass $\approx$ 2 forward passes.

  Total: 1 (forward) + 2 (backward) = 3 forward passes

With Checkpointing:

- Forward Passes: 2 passes (an extra pass is required for checkpointing to recompute certain activations needed for the backward pass, as not all activations are stored)
- Backward Pass: 1 pass
- Total Equivalent Forward Passes: Using the same equivalence, 1 backward pass $\approx$ 2 forward passes .

  Total: 2 (forward) + 2 (backward) = 4 forward passes

Comparing MFU and HFU:

- Without checkpointing, the total is 3 forward passes .
- With checkpointing, the total increases to 4 forward passes .
- This relationship can be expressed as:

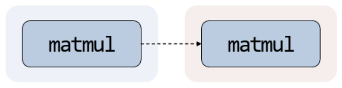
$$MFU = \frac{3}{4}HFU$$

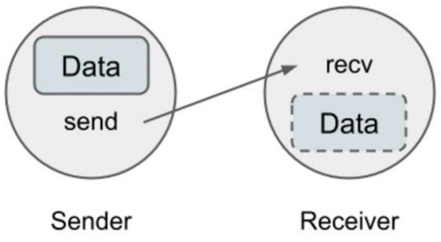
1.7 Maximizing MFU subject to Memory Constraints

Maximizing Model Flops Utilization (MFU) within memory constraints is essential to achieve optimal parallelism efficiency. Effective parallelism strategies ensure high computational throughput while minimizing memory bottlenecks. Optimizations that enhance both inter-op and intra-op parallelism help reduce communication overhead, ensuring full utilization of hardware resources. This can result in substantial gains in MFU and overall performance.

2 Collective Communication

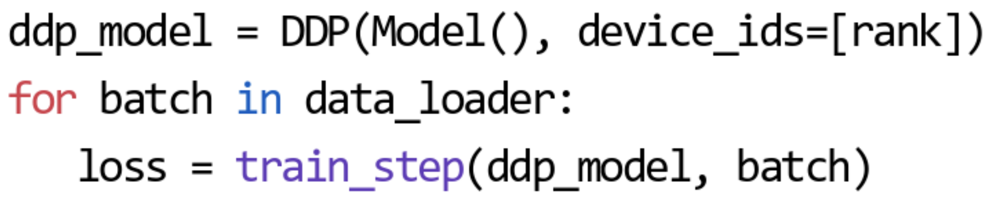
This is the general form of point to point communication:





PyTorch's DDP

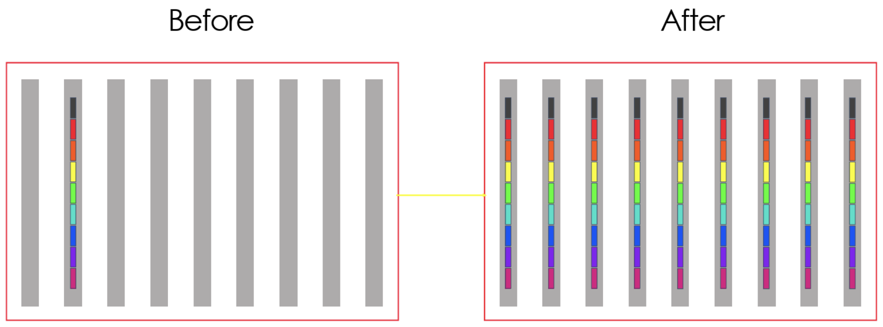
In Torch's DDP there is an implied all-reduce on the train-step in this:



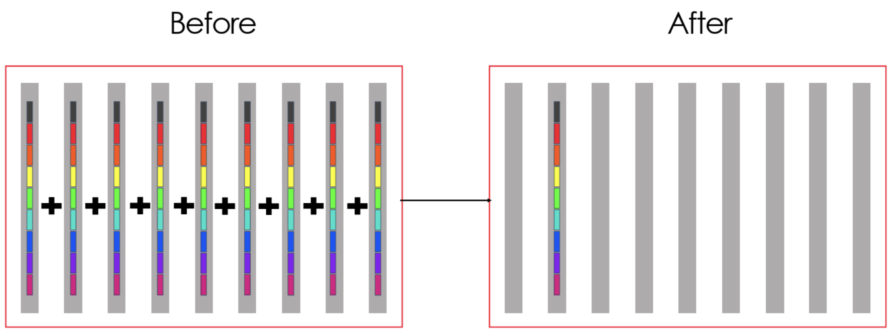
```
ddp_model = DDP(Model(), device_ids=[rank])
for batch in data_loader:
    loss = train_step(ddp_model, batch)
```

2.1 Collective Communications

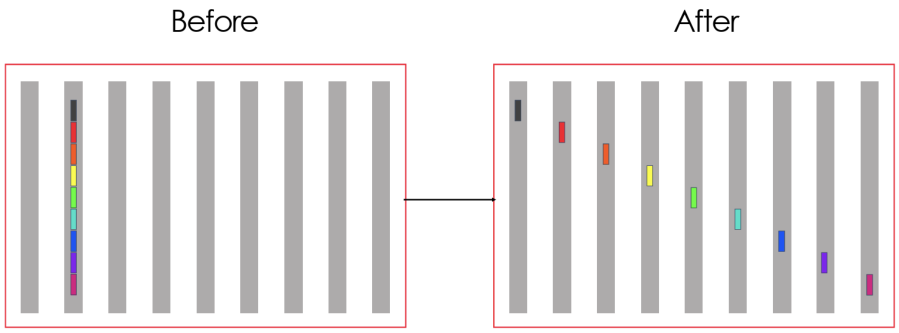
- **Broadcast:** In broadcast, data from a single process/rank is distributed to all other processes in the group.



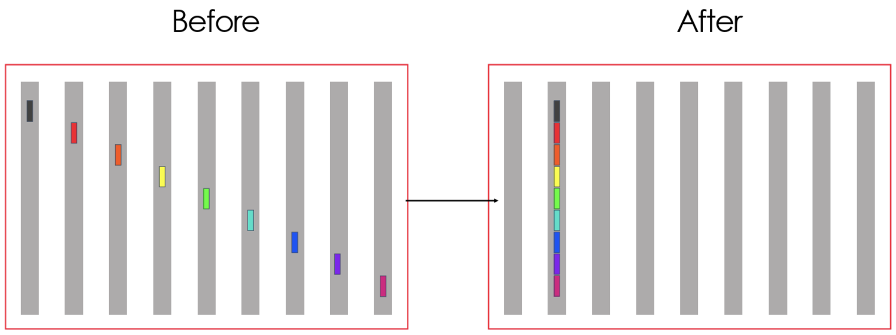
- **Reduce (Reduce-to-One):** Each rank has an array, and we aim to reduce these arrays to one single array. In Reduce-to-One, we combine values from all processes and assign the result to a single process. It's commonly used for summation, product, or finding minimum/maximum. Broadcast and Reduce-to-One can be thought of as inverse operations, although not in a mathematical sense, since the aggregations performed in reduce are often irreversible. But, since broadcast spreads information from one to all and reduce gathers information from all to one, the two processes can be interpreted as opposites.



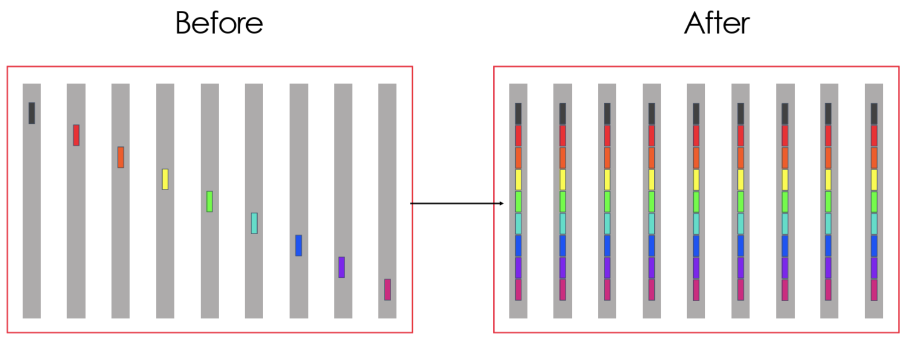
- **Scatter:** Splits data from the array in a single rank into equal-sized chunks, and sends each chunk to a different process. Each process gets a unique portion of the data. It's useful for distributing work across multiple processes, such as partitioning a large dataset across multiple workers for parallel computation.



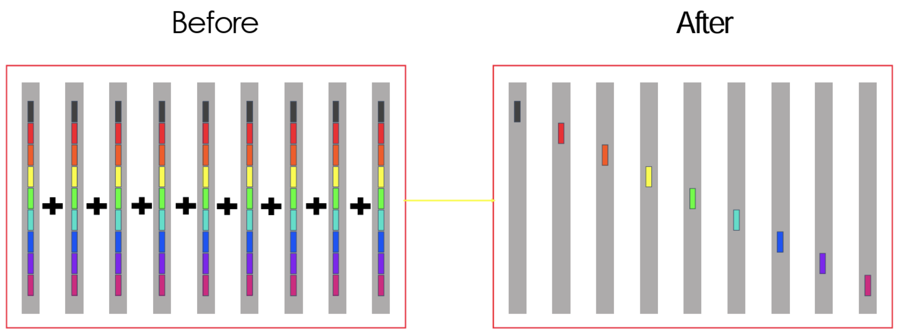
- **Gather:** Gather can be thought of as the inverse of scatter. Each process has a portion of an array, and scatter collects data from multiple processes and combines it into a single process in a structured way.



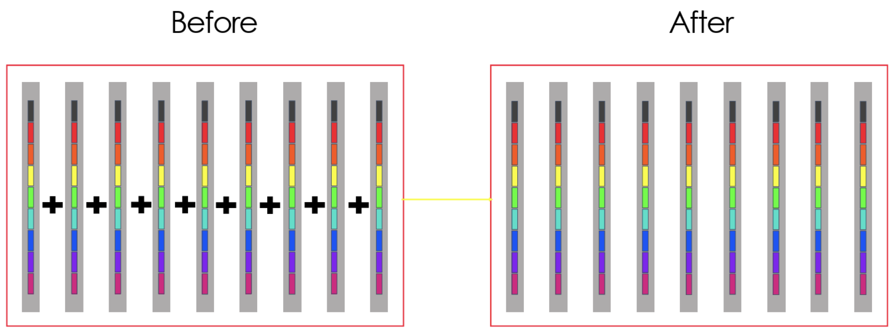
- **Allgather:** An extension of gather, where every process collects and shares data from all other processes. Instead of sending all the data to a single target process like gather does, the collected data is sent to every process. This is useful in distributed training when each worker computes a partial result, and all workers need the complete set of results for further computation.



- **Reduce-Scatter:** Combines reduce and scatter operations. First, each process contributes a data chunk, and an element-wise reduction (e.g., sum, min, max) is applied across corresponding chunks from all processes. Then, instead of keeping the full reduced result, each process receives only a unique portion of the reduced data (the scatter process). This makes it more efficient than performing reduce followed by scatter separately. Reduce-scatter is particularly useful in distributed machine learning when summing gradients across workers and distributing the summed gradients for further computation. In a way (while not according to the mathematical definition), reduce-scatter can be thought of as the reverse of allgather.



- **Allreduce:** Combines reduce and broadcast operations by reducing data to a single result and broadcasting it back to all processes. It's efficient for summation or aggregation tasks. Under the hood, each process starts with a different input array. The operation first performs a reduce step, where data from all processes is aggregated using an operation (e.g., sum, max, min, product). The reduced result is then broadcasted back to all ranks, ensuring that each process gets the same final result.



Facts

- These connective operations originated from HPC (high-performance computing).
- Collective communication is significantly more expensive than point-to-point (P2P) communication. In collective communication, we ask all our ranks to communicate, compared to just two in p2p.
- Collective operations can be constructed using multiple P2P communications.
- Collective communication has been highly optimized over the past 20 years.
- Look out for "X"CCL libraries such as NCCL, MCCL, and OneCCL for optimized implementations. The strength of Nvidia's NCCL library is part of the reason behind their recent success.
- Collective communication is not fault-tolerant, and failure in one process may disrupt the entire operation.

2.2 Communication Model

The latency of communication (between GPUs and/or nodes) can be roughly modeled as

$$\alpha + n\beta, \quad \beta = \frac{1}{B}$$

in which α is the initial cost of setting up connection, β represents the inverse bandwidth, and n is the message size.

- **Small Message Size ($\alpha >> n\beta$):** Latency (α) dominates; the focus is on reducing communication latency.
- **Large Message Size ($\alpha << n\beta$):** Bandwidth utilization ($n\beta$) dominates; the focus is on maximizing bandwidth.

There are two families of algorithms to implement the collective communication primitives.

- **Minimum Spanning Tree Algorithm** for small messages.

  Constructs a spanning tree connectivity, so that the overall cost has the minimum number of α.
- **Ring Algorithm** for large messages.

  Each process communicates with its neighbor in a ring-like structure, ensuring full utilization of device parallel and bandwidth.

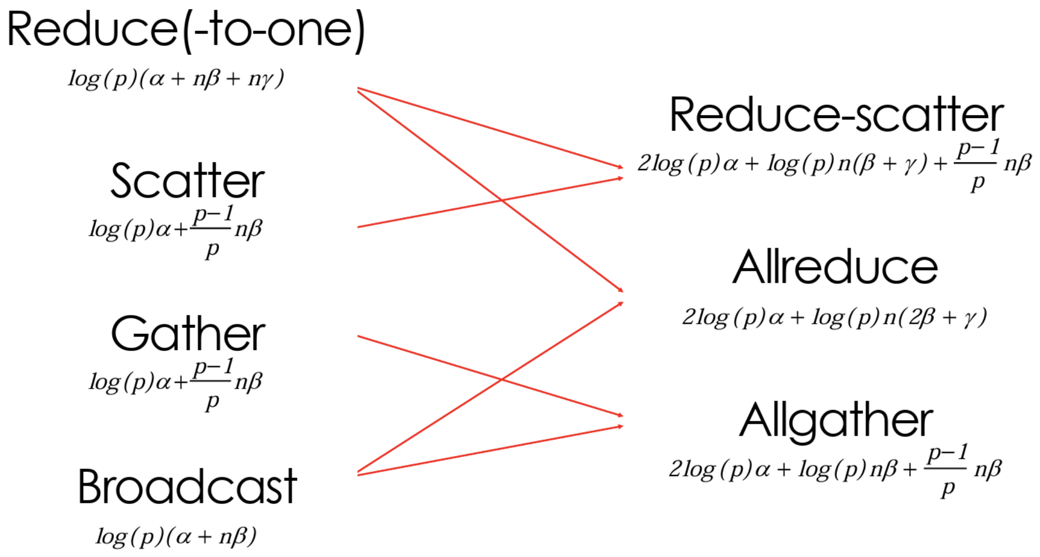
In ML-systems, the messages are usually large data/activation/weight/optimizer-state tensors, i.e. bandwidth utilization is the main concern dominates.

2.3 MST in Operations: Small Model

2.3.1 Overview

The Minimum Spanning Tree (MST) algorithm plays a crucial role in communication models that prioritize low latency. It is particularly well-suited for scenarios where message sizes are small, as it structures communication paths efficiently while ensuring minimal rounds of data transfer. By following a spanning tree structure, MST establishes a hierarchical communication pattern that optimizes latency but does not fully utilize the available bandwidth.

2.3.2 Communication Model and Assumptions


Reduce(-to-one)
$log(p)(\alpha + n\beta + n\gamma)$
Scatter
$log(p)\alpha + \frac{p-1}{p} n\beta$
Gather
$log(p)\alpha + \frac{p-1}{p} n\beta$
Broadcast
$log(p)(\alpha + n\beta)$
Reduce-scatter
$2log(p)\alpha + log(p)n(\beta + \gamma) + \frac{p-1}{p} n\beta$
Allreduce
$2log(p)\alpha + log(p)n(2\beta + \gamma)$
Allgather
$2log(p)\alpha + log(p)n\beta + \frac{p-1}{p} n\beta$

The MST approach assumes a network where each node can communicate with only one other node at a time. This constraint shapes how data is transmitted through the system, forming an organized communication tree. Unlike approaches that aim to maximize throughput by utilizing all available links, MST selectively engages connections to reduce latency. As a result, many network links remain idle, and bandwidth is not fully leveraged. However, this trade-off is intentional, as MST is designed to minimize initial communication delay rather than optimize overall throughput.

2.3.3 Characteristics and Trade-offs

The MST algorithm is particularly effective when dealing with small messages. Its core advantage lies in reducing the initial latency term, often referred to as the alpha (α) term in the latency-bandwidth model. This characteristic makes MST highly suitable for applications where the primary concern is the speed of message propagation rather than the total amount of data transmitted. However, this focus on minimizing latency comes at the cost of bandwidth efficiency. Since each node only communicates with a designated target node, many potential communication links remain unused, leading to a scenario where bandwidth resources are underutilized.

One of the fundamental principles of MST-based communication is its ability to break down complex reduction operations into more manageable components. For example, reduce-scatter can be understood as a combination of scatter and reduce operations, while all-reduce is effectively implemented through a sequence of broadcast and reduce steps. Similarly, all-gather is achieved through a combination of gather and broadcast techniques. These decompositions illustrate how MST efficiently structures message passing while adhering to its latency-driven design.

2.3.4 Implications of MST in Distributed Systems

One of the key insights of MST-based communication is its impact on one-to-one message transmission. In this model, each node sends data only to a specific target node, minimizing communication overhead but also leaving many network links inactive. This results in a system where the initial phase of communication sees a significant amount of idle time across various connections. While this may appear inefficient from a bandwidth utilization perspective, it is an intentional design choice aimed at prioritizing low-latency transmission.

By focusing on latency reduction, the MST approach significantly improves the speed of communication rounds. The structured and hierarchical nature of MST ensures that messages propagate efficiently, avoiding unnecessary delays caused by network congestion. However, this efficiency comes with limitations, particularly in scenarios where large messages need to be transmitted. The selective use of network links means that MST is not an ideal solution for cases where maximizing data throughput is the primary goal.

2.3.5 Conclusion

The MST algorithm serves as an effective approach for optimizing low-latency communication in distributed systems where small message sizes are dominant. While it successfully minimizes communication rounds and ensures efficient data propagation, it does so at the expense of bandwidth utilization. The selective use of network links results in initial idle connections, but this trade-off is justified when minimizing latency is the primary objective. In cases where message sizes are larger and maximizing throughput is essential, alternative strategies are required to complement MST's limitations.

2.4 Ring Algo in Operations: Large Messages

Key Characteristics

- **Uses Links Between Every Two Nodes** $\rightarrow$ Ensures full bandwidth utilization.
- **Higher Latency; Number of Rounds Doesn't Matter** $\rightarrow$ Prioritizes bandwidth over minimizing communication rounds.
- **Logical Ring in a Physical Linear Array** $\rightarrow$ Uses worm-hole routing to avoid conflicts in wrap-around messages.

How It Works

- **Message Splitting** : A large message is divided into equal-sized chunks. For a message of size n, each of the p nodes gets $\frac{n}{p}$ data.
- **Sequential Data Transfer**:
  - Each node sends a chunk to the next node while receiving a chunk from the previous node.
  - This ensures continuous data transfer and prevents idle nodes.
- **Sequential Data Transfer**:
  - Each node sends a chunk to the next node while receiving a chunk from the previous node.
  - This ensures continuous data transfer and prevents idle nodes.
- **Full Utilization of Links** :
  - All available bandwidth is used as each node is actively engaged in sending and receiving.
  - Unlike MST algorithms, no links remain underutilized or idle.

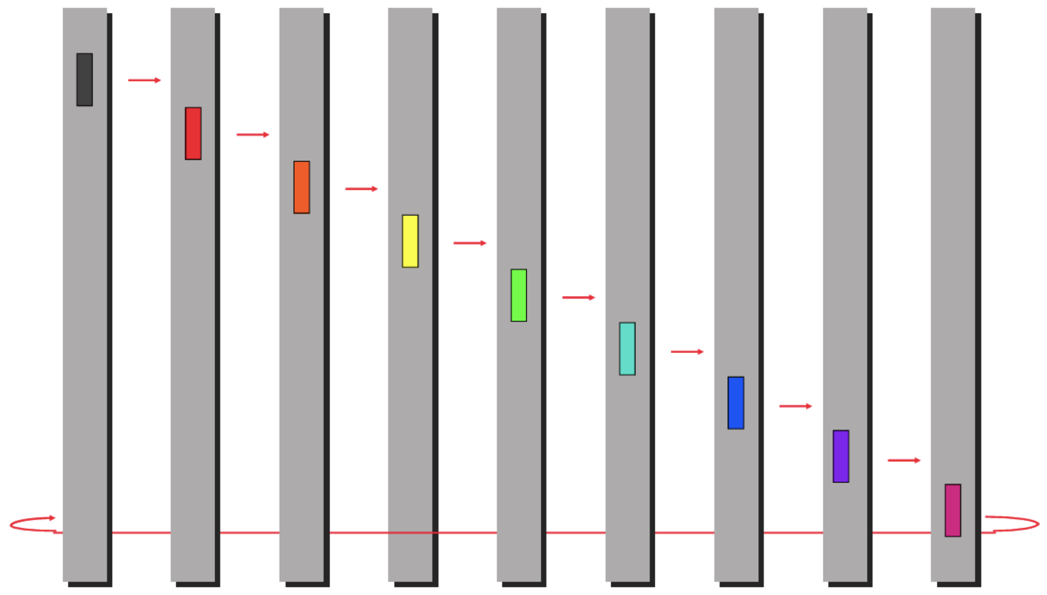
Advantages Over MST

- **Bandwidth Optimality** → Unlike MST, where some links are idle, Ring Algorithm ensures all links are utilized.
- **Scalability** → Works efficiently for arbitrary numbers of nodes.
- **Better Performance for Large Messages** → Since the algorithm focuses on bandwidth rather than latency, it performs better when message sizes are large.

Example: AllReduce Using Ring Algorithm

The Ring Algorithm is often used for AllReduce operations in distributed training:

- Each node contributes to a collective computation while simultaneously transmitting and receiving data.
- This method is used in high-performance computing (HPC) and deep learning frameworks (e.g., Horovod, NCCL, PyTorch DDP).



Reduce-scatter

$(p-1)\alpha+\frac{p-1}{p}n(\beta+\gamma)$

Scatter

$log(p)\alpha+\frac{p-1}{p}n\beta$

Gather

$log(p)\alpha+\frac{p-1}{p}n\beta$

Allgather

$(p-1)\alpha+\frac{p-1}{p}n\beta$

Reduce(-to-one)

Allreduce

Broadcast