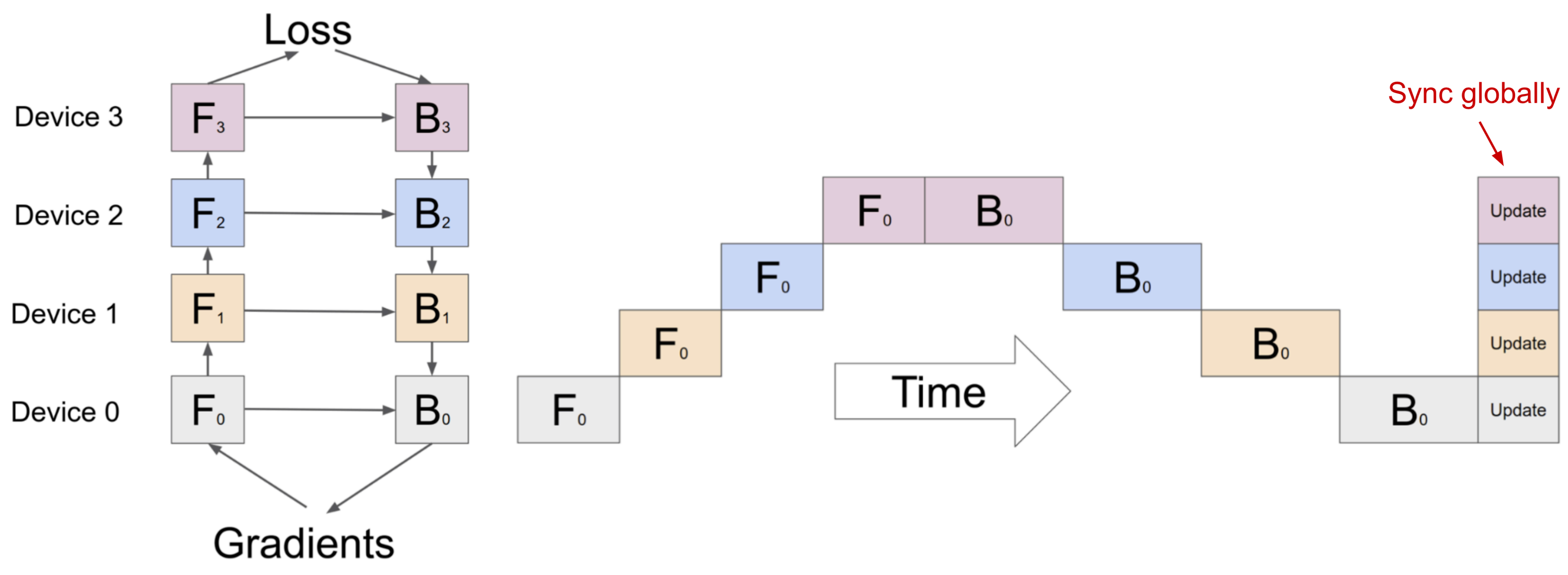
micropp

Naive Model Parallelism

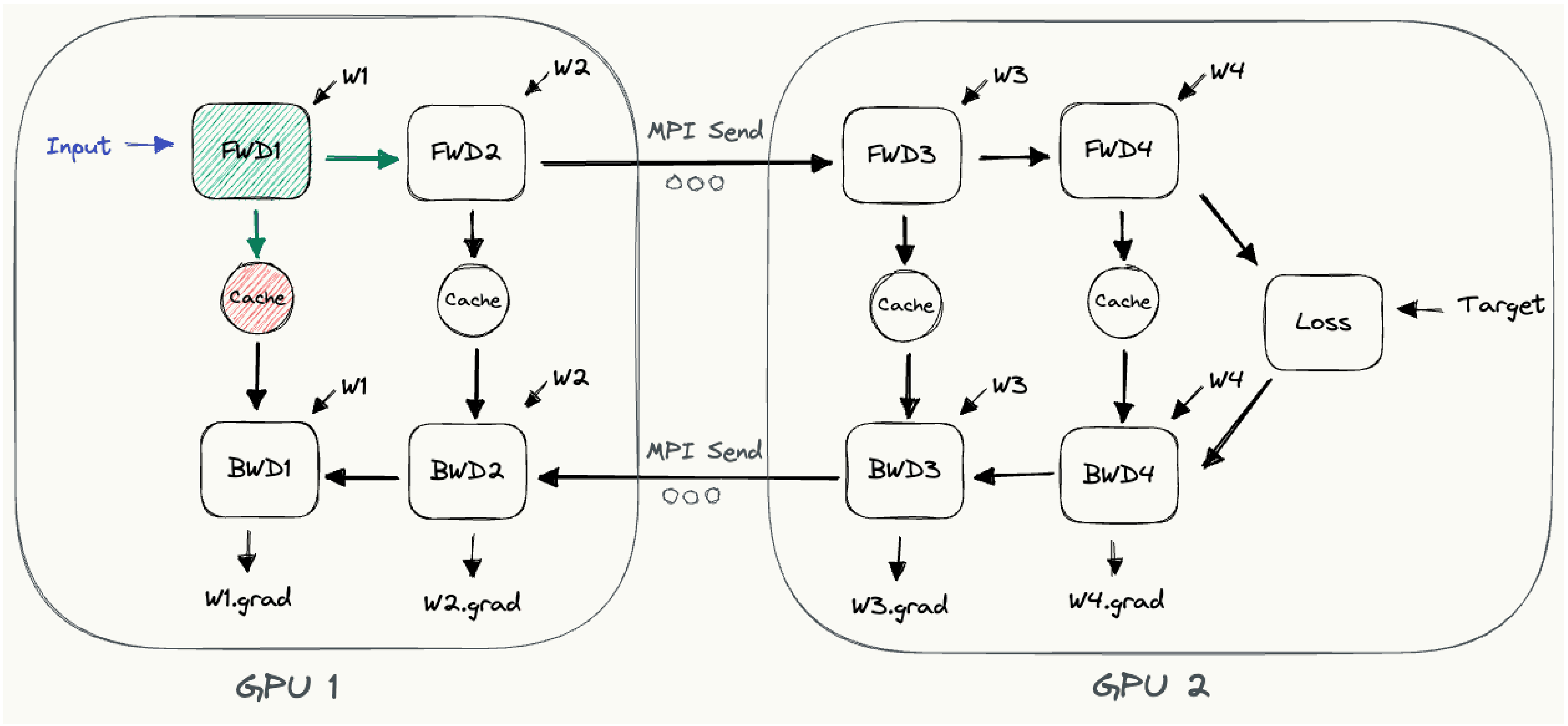
- Naive model parallelism is the most straightforward way of implementing pipeline-parallel training.
- We split our model into multiple parts, and assign each one to a GPU.
- Then we train, inserting communication steps at the boundaries where we've split the model.
- We only use node-to-node communication (MPI.Send and MPI.Recv) and don't need any collective communication primitives.
- 

Forward Pass

1. Compute intermediate on GPU1.
2. Transfer the resulting tensor to GPU2.
3. GPU2 computes the loss of the model.

Backward Pass

1. GPU2 calculates derivative of loss w.r.t its weights and input.
2. Send the gradients w.r.t. intermediate from GPU2 to GPU1.
3. GPU1 then completes the backward pass based on the gradients it was sent.



By looking at the pebble graph, we can observe some inefficiencies of naive model parallelism:

1. Low GPU utilization.

2. No interleaving of communication and computation.
3. High memory demand.

`requires_grad`

Scenario 1: `requires_grad = True`

In this case, we allow the “chain” of math to stay connected across the two devices.

1. **Rank 0** performs a calculation (A×B=C) and sends the result to **Rank 1**.
2. **Rank 1** receives C. We manually set `C.requires_grad = True`.
3. **Rank 1** performs its own calculation (C×D=Output).
4. When we call `.backward()` on Rank 1, PyTorch calculates the gradient for its weights (D) **and** for the input C.
5. **The Result:** Rank 1 now has a value for `C.grad`. It calls `send_backward(C.grad)` to Rank 0. Rank 0 receives this and can now calculate the gradients for its own weights (B).

Scenario 2: `requires_grad = False`

In this case, Rank 1 treats the incoming data as a “constant” rather than a variable.

1. **Rank 0** sends C to **Rank 1**.
2. **Rank 1** receives C. By default, `requires_grad` is **False**.
3. **Rank 1** calculates (Output=C×D).
4. When we call `.backward()` on Rank 1, PyTorch calculates the gradient for the weights (D). However, because C was marked as a constant (no grad required), the engine **stops** there.
5. **The Result:** `C.grad` is `None`. Rank 1 has nothing to send back. Rank 0 never receives a gradient, so its weights (B) never move. **Only half the model learns.**

Essentially, `requires_grad = True` creates a “hook” at the very edge of the device’s memory. Without that hook, the backward pass has nothing to grab onto to pull the information back across the network to the other device.

`detach()`

1. Preventing Memory Leaks

If you do not call `.detach()`, the `output` tensor remains “hooked” to the computational graph of the current GPU. PyTorch will try to keep all the activations of all previous layers in memory because it thinks you might call `.backward()` on that specific `output` variable later. This leads to a massive memory leak.

2. The “Micro” Architecture Logic

By detaching, you are explicitly saying: *“I am done with this forward pass locally. I am handing off a static copy of the data to the next device”*.

- **Forward:** Next device gets a “clean” tensor and restarts the graph using `requires_grad = True`.
- **Backward:** We manually reconnect the chain later when we receive the gradient and call `output.backward(received_grad)`.

3. Summary

You don’t want the next GPU to have “ghost” references to memory that it cannot access. You give it the activations and keep the graph locally for when the backward pass eventually returns.