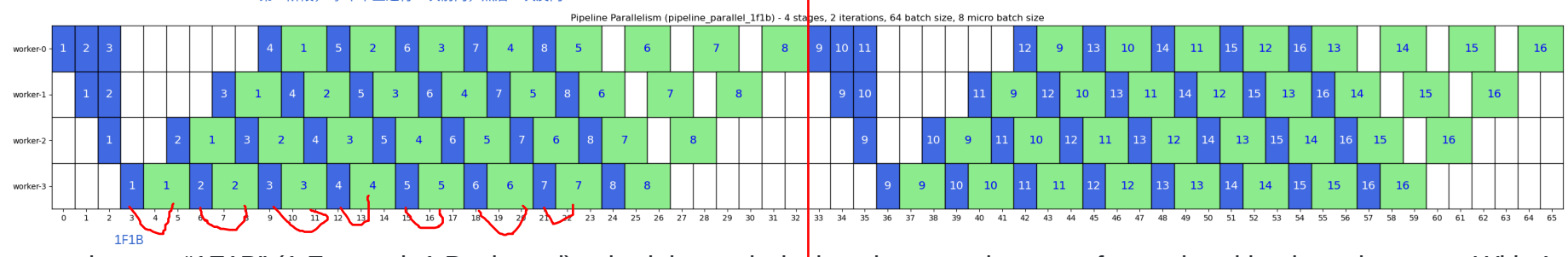
micropp

PipeDream: 1F1B Scheduling

- PipeDream accelerates pipeline parallelism by starting the backward pass for a microbatch as soon as its forward pass completes the final stage, enabling earlier disposal of cached activations.

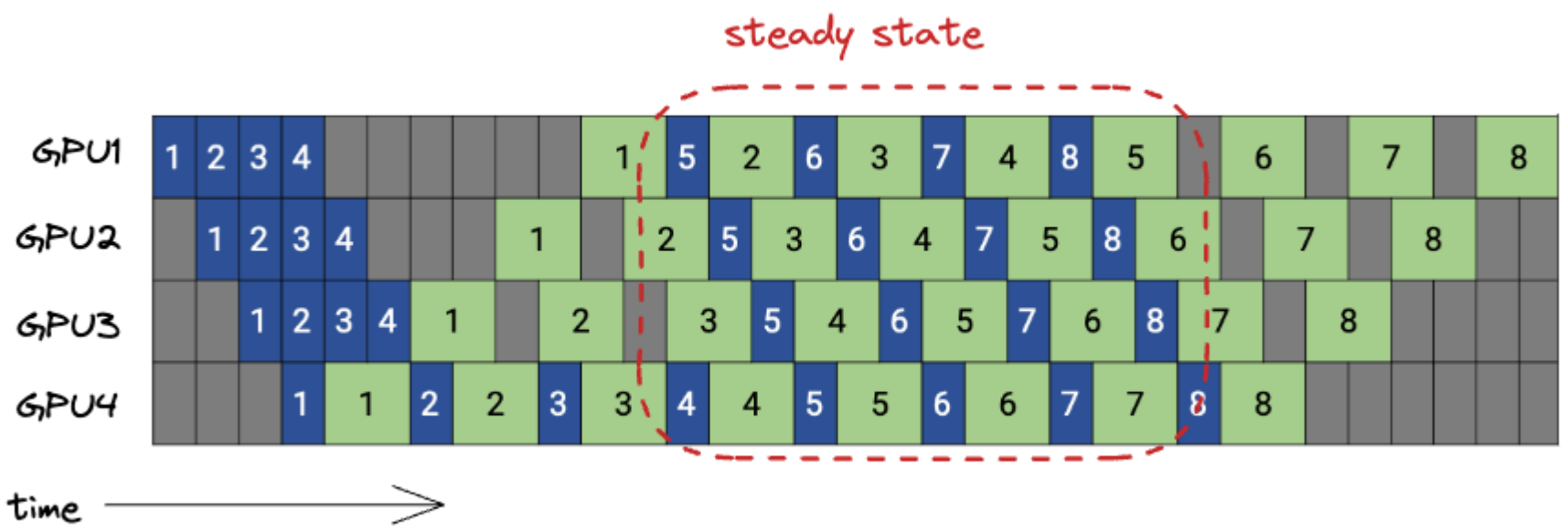
  - 第一阶段，只进行前向
  - 第二阶段，每 个卡上进行一次前向，然后一次反向
  - 第一批数据训练完毕
  - 第二批数据训练开始
  - 第三阶段，每 个卡上只进行反向



- In the steady-state “1F1B” (1 Forward, 1 Backward) schedule, each device alternates between forward and backward passes. With 4 GPUs and 8 microbatches, at most 4 microbatches are in flight at any time, halving the same peak activation memory as GPipe.

$$O(\#\text{max microbatches in flight} \cdot \text{microbatch-size} \cdot \frac{\#\text{total layers}}{\#\text{GPUs}})$$

  - A microbatch is in-flight if we performed >1 forward pass for it, but haven’t completed all the backward passes yet.



- Despite this memory advantage, the pipeline “bubble fraction”—idle time due to dependencies—is the same for GPipe and PipeDream because of the sequential dependency structure.
  - Visually, looking at the above PipeDream plot if you shift the blue forward passes left and the green backward passes right, you get GPipe. This explains why the bubble fraction is the same.

Communication Volume

- Both GPipe and PipeDream require each GPU to send/receive `batch_size × N` data per forward and backward, totaling approximately `2 × (num_GPUs – 1) × batch_size × N` floats per batch for a model of hidden size `N`.
  - The -1 terms comes from the initial GPU not having to receive and the last GPU not having to send anything.

Hypothetical Scenario

- If we had 100 micro-batches, GPipe would need 100x memory, but 1F1B would still only need about 4x. This is why we go through the trouble of the warm-up and cooldown!

Why 1F1B Needs Async Forward + Request Tracking

1. **Async forward sends** (`isend_forward`) - prevents deadlock 导步发送，可以防止死锁
2. **Save request handles** (`async_requests.append(req)`) - prevents buffer deallocation 将异步函数的引用加入列表，防止缓存释放

Case 1: Blocking `send_forward` (DEADLOCK)

Rank 2:

rank=2上，数据m1被 rank=2 send()阻塞,等待rank3进行receive()，所以数据m0不能调用receive()，因为网卡被占用
rank=3上，数据m0正在发送到rank=2,等待rank2进行receive(),不会对m1进行receive()

- Blocked on `send_forward()` for microbatch 1, waiting for Rank 3 to call `recv_forward()`
- Can't call `recv_backward()` for microbatch 0 because it's stuck in the blocking send

Rank 3:

- Blocked on `send_backward()` for microbatch 0, waiting for Rank 2 to call `recv_backward()`
- Won't call `recv_forward()` for microbatch 1 because it already finished forward(0) and moved to backward(0)

Result: Both are blocked sending, and neither can receive because:

- Rank 2 is stuck sending forward(mb1) and can't receive backward(mb0)
- Rank 3 is stuck sending backward(mb0) and won't receive forward(mb1)

This is why async `isend_forward()` is needed: Rank 2 can start the send and immediately proceed to `run_backward(0)` → `recv_backward()`, which unblocks Rank 3, allowing Rank 3 to eventually receive the forward(mb1) that Rank 2 started.

Case 2: Async `isend_forward` WITHOUT saving request (CRASH)

```
# Rank 0:
def run_forward(micro_batch_idx):
    output = model(input_data)
    req = comms.isend_forward(output.detach())  # Start async send
    # req is NOT saved anywhere
    # Function returns, req goes out of scope
    # Python GC may collect req object

# What happens inside Gloo:
# 1. isend() creates internal buffer reference to output.detach()
# 2. Send happens in background thread
# 3. req object is GC'd → Gloo loses reference
# 4. Gloo tries to access buffer → "Cannot lock pointer to unbound buffer"
# 5. CRASH
```

Why saving the request fixes it:

```
async_requests.append(req)  # Keep reference alive!
# Now req stays in scope until function returns
# Gloo can safely access buffer during async send
```

Case 3: Async `isend_forward` WITH saving request (WORKS)

```
# Rank 0:
def run_forward(micro_batch_idx):
    output = model(input_data)
    req = comms.isend_forward(output.detach())
    async_requests.append(req)  # ✅ Keep alive
    # req stays in async_requests list
    # Buffer reference stays valid
    # Gloo can complete async send safely

# Function returns, but async_requests is still in scope
# (it's a closure variable, lives until onef_oneb_pipeline_step returns)
# All sends complete before function exits
```

Key Insight

The request object returned by `isend()` contains a reference to the tensor buffer. If the request is garbage collected, Gloo loses that reference and crashes. Saving it in `async_requests` keeps the reference chain alive:

```
output.detach() → req object → Gloo internal buffer reference
     ↑                ↑
     |                |
  output_buffers  async_requests (both keep things alive)
```

Both references are needed:

- `output_buffers` keeps the original `output` tensor alive (needed for backward)
- `async_requests` keeps the `req` object alive (needed for Gloo's buffer reference)