PYTORCH CONFERENCE EUROPE 2026 • PARIS

DualPipe from Scratch

Implementing DeepSeek's 5D Parallelism in PyTorch

Devdatta Jadhav
Tech Lead ML Engineer • MLwithDev

Prerequisites: torch.distributed basics

github.com/DevJadhav/deepseek-from-scratch

7-8 April 2026

Three Bottlenecks at 2,048-GPU Scale

Today: We focus on the scheduler — DualPipe — in PyTorch, and found what the paper leaves out.

The Pipeline Bubble Problem

Bubble Overhead

`bubble = (P - 1) / M`

P=8, M=16:
`bubble = 7/16 = 43.75%`

Nearly half the compute wasted on idle GPUs

DualPipe: Bidirectional Scheduling

Stream A →

Microbatches:
Stage 0 → 1 … → P-1

← Stream B

Microbatches:
Stage P-1 … … 1 → 0

Bubble filled from both ends → 47% drops to 3%. 55% throughput gain over GPipe.

3%	55%	4×
Bubble overhead	Throughput gain	MoE dispatch reduction

5D Parallelism: How TP, PP, DP, EP, SP Interact

Tensor (TP=4)
Split weight matrices within node via `torch.distributed`

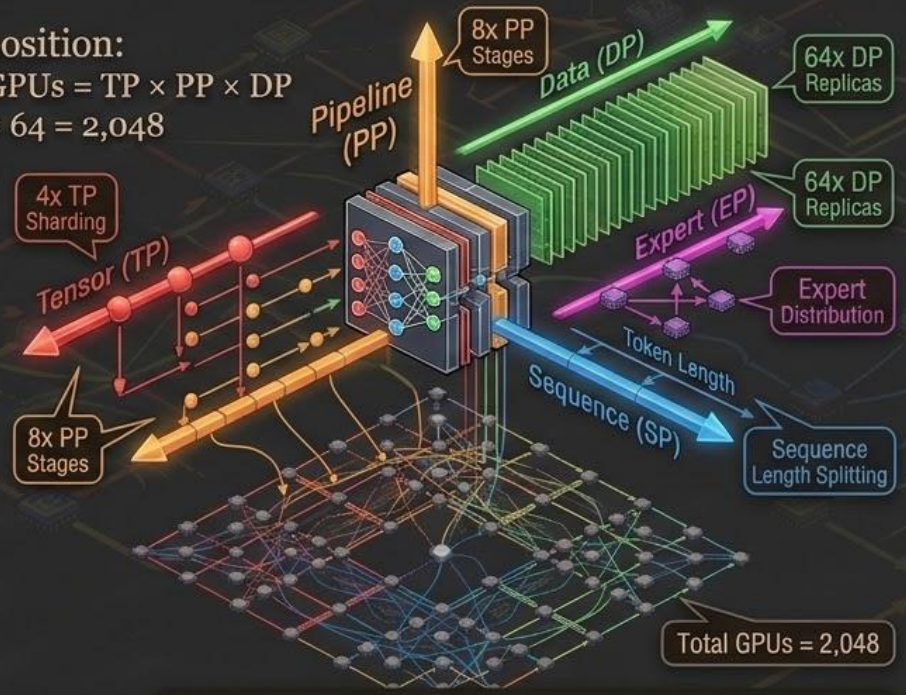
Pipeline (PP=8)
Split layers across stages — DualPipe schedules this

Data (DP=64)
Replicate model, split data — `torchDistributedDataParallel`

Expert (EP=64)
Distribute 256 MoE experts across process groups

Sequence (SP)
Split activations along sequence — reduces memory

Composition:
Total GPUs = TP × PP × DP
4 × 8 × 64 = 2,048




```
# Create process groups for sub-dimensions
tensor_group = dist.new_group(ranks=[0, 1, 2, 3])
pipeline_group = dist.new_group(ranks=[0, 4, 8, ...])
data_group = dist.new_group(ranks=[...])
```

The Warmup Formula (Not in the Paper)

```
warmup_steps = (P // 2) + (M - 1) * 2
```

P = Number of processors or nodes
M = Number of micro-batches

Example: P=8, M=16 → **warmup = 4 + 30 = 34** steps before steady state

Too Few Steps

backward before activations, NaN

Off By One

deadlock, torch.distributed.barrier hangs

Exactly Right

3% bubble, both streams overlap

The DualPipe Scheduler

src/deepseek/torch/training/dualpipe.py

```python
class ScheduleStep(Enum):
    WARMUP_FWD = "warmup_forward"
    STEADY_FWD = "steady_forward"
    STEADY_BWD = "steady_backward"
    COOLDOWN = "cooldown"

class DualPipeScheduler:
    def __init__(self, num_stages, num_microbatches):
        self.P = num_stages
        self.M = num_microbatches
        # The formula NOT in the paper:
        self.warmup = (self.P // 2) + (self.M - 1) * 2

    def generate_schedule(self) -> List[ScheduleStep]:
        steps = []
        for t in range(self.total_steps):
            # Bidirectional: fwd_A + bwd_B overlap
            steps.append(self._classify(t))
        return steps
```

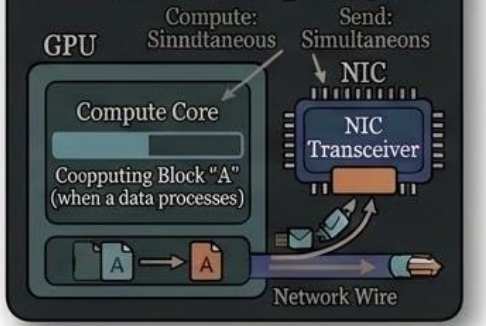
Communication Overlap with torch.distributed

```python
async def overlap_comm_compute(self, mb_fwd, mb_bwd):

    # 1. Start async backward gradient send

    bwd_handle = dist.isend(grad, dst=prev_rank)

    # 2. Compute forward while network sends
    output = self.stage_forward(mb_fwd)

    # 3. Wait for backward send

    bwd_handle.wait()

    # 4. Start async forward output send

    fwd_handle = dist.isend(output, dst=next_rank)

    return output, fwd_handle
```

torch.distributed Behavior

- `dist.isend()` = async (non-blocking) – Returns Work handle.
- `dist.send()` = sync (blocking) – Kills overlap.
- *Near-zero idle time.*

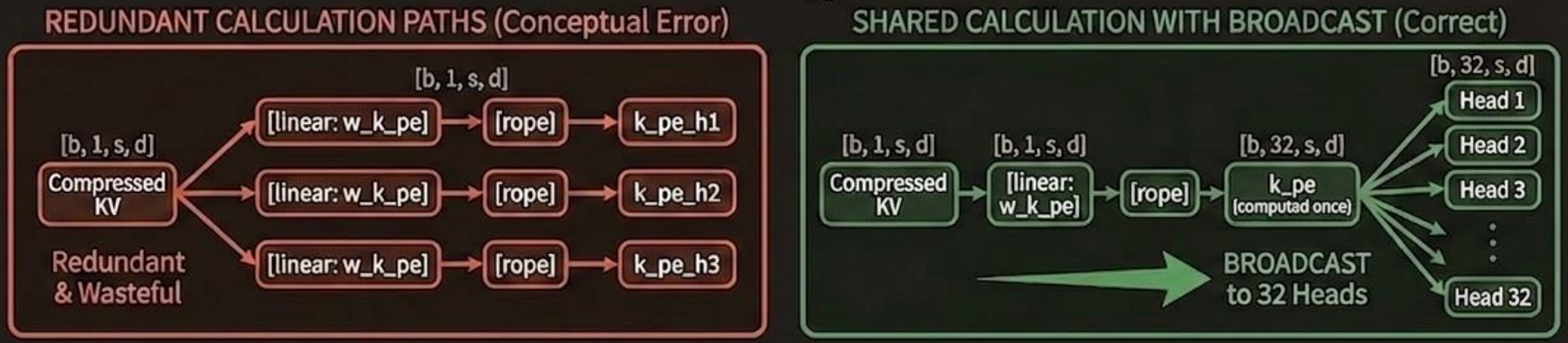
Hardware Overlap Diagram



K_{pe} Shared Across Heads in Decoupled RoPE

Not explicit in paper — src/deepseek/torch/model/mla.py

Approaches to K_{pe} Calculation




```
# From mla.py — correct PyTorch implementation
k_pe = rope(self.w_k_pe(compressed_kv))  # compute once
k_pe = k_pe.unsqueeze(1).expand(-1, n_heads, -1, -1)
# Shape: [batch, n_heads, seq, d_rope] via broadcast
```

<-- Highlighting the correct broadcast operation

Bias Update Timing in Load Balancing

CFP: EMA initialization + auxiliary-loss-free balancing — src/deepseek/torch/model/moe.py

Bugs We Caught

EMA initialized to zeros. First ~100 steps: some experts get 10x more tokens. Capacity dropping priority wrong.

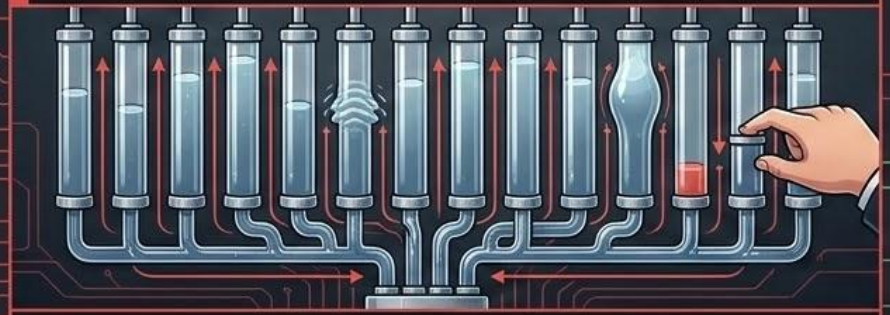
The Fix

Init EMA = 1/num_experts. Update bias BEFORE routing.


```
# Bug: zeros init causes routing chaos
self.expert_ema = torch.zeros(num_experts)    # BAD

# Fix: uniform init from our moe.py
self.expert_ema = torch.full((n,), 1.0 / n)   # GOOD
```

Sigmoid Routing: Selection vs Gating

Softmax (Coupled)



```
scores = F.softmax(logits, dim=-1)
top_k = scores.topk(k=8)
gates = scores[top_k.indices]
```

Sigmoid (Decoupled)

```
scores = torch.sigmoid(logits)
top_k = scores.topk(k=8)
gates = scores[top_k.indices]
gates = gates / gates.sum()
```

Selection (which experts?) and gating (how much weight?) are separate decisions — critical at 256 experts.

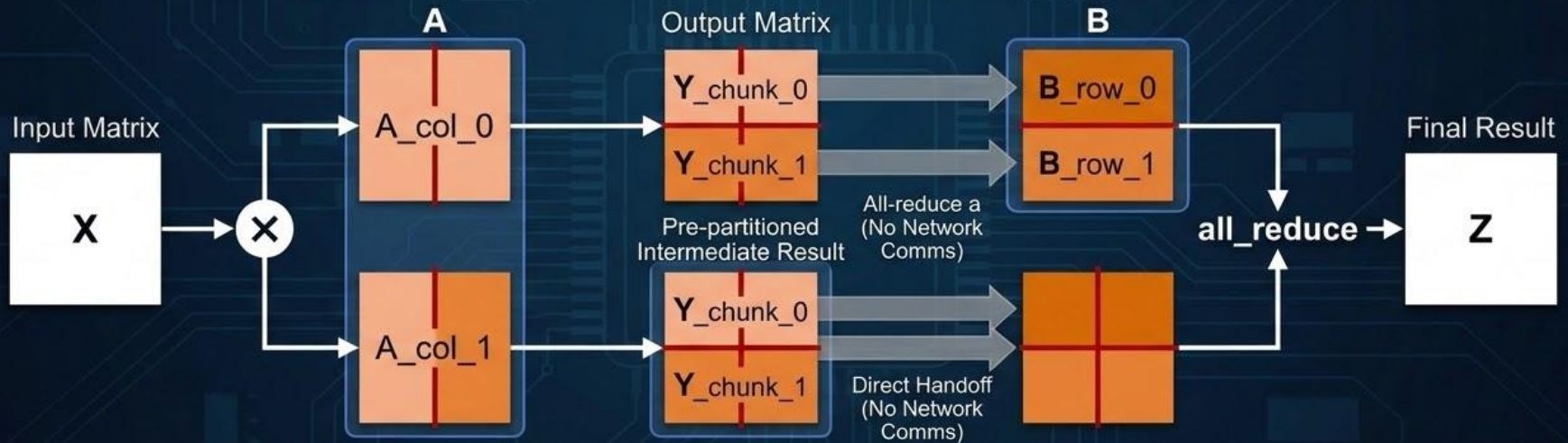
Tensor Parallelism: Column -> Row Composition

COLUMN PARALLEL OPERATION

`torch.mm(X, A_col_chunk)`
- No `dist.all_reduce` needed

ROW PARALLEL OPERATION

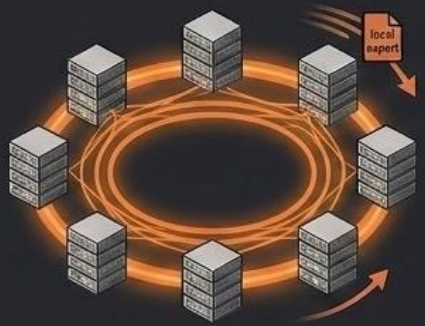
`torch.mm(X_chunk, A_row_chunk)`
- Requires `dist.all_reduce(output)`



Column → Row eliminates one `dist.all_reduce` per transformer block
Saves thousands of communication rounds using `torch.distributed`.

Hierarchical All-to-All

Two-level communication reducing MoE dispatch overhead by 4×



LEVEL 1: INTRA-NODE
(Fast, Dense NVLink)

- NVLink 900 GB/s Ring Bandwidth
- Local all_to_all communication within node group
- Prioritizes local experts

LEVEL 2: INTER-NODE (Slow, Thin InfiniBand Connection)
Dual-level Topology

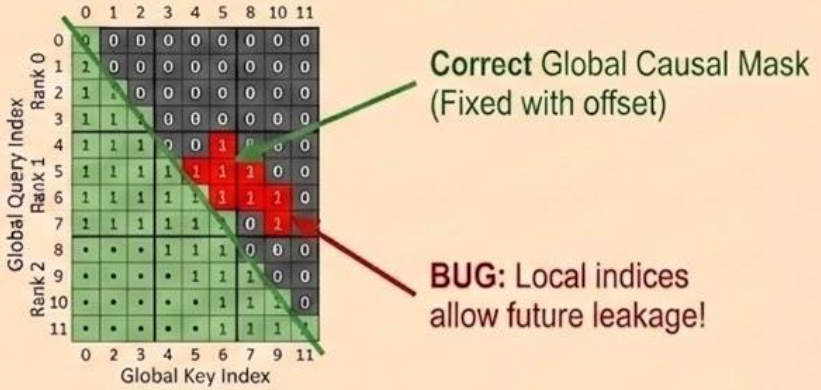
- InfiniBand 50 GB/s Inter-Node Connection
- 18x slower than Level 1
- Cross-node all_to_all across multiple groups
- Only when needed

```
# Two-level process groups in PyTorch
intra_group = dist.new_group(ranks=local_ranks)
inter_group = dist.new_group(ranks=node_leaders)

# Level 1: fast local dispatch
dist.all_to_all(local_out, local_in, group=intra_group)

# Level 2: cross-node only when needed
dist.all_to_all(global_out, global_in, group=inter_group)
```

Causal Mask Position Offsets

The Bug

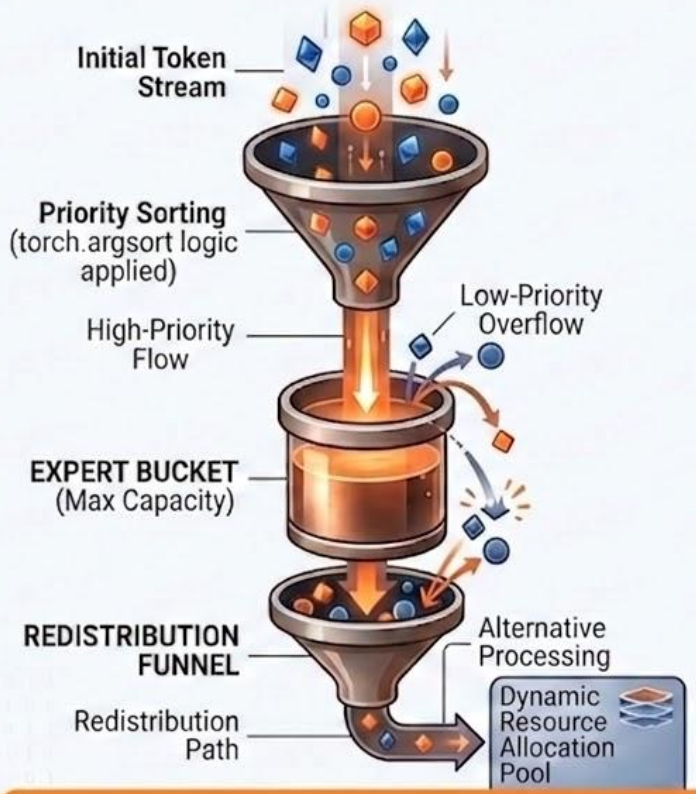
- SP uses local indices for mask
- Future tokens become visible
- Silent information leakage




```
# Bug: local positions → future leakage
mask = torch.tril(torch.ones(local_len, local_len))  # WRONG
# Fix: global positions with SP offset
offset = dist.get_rank(sp_group) * chunk_size
mask = create_causal_mask(local_len, offset=offset)  # CORRECT
```

No error. No NaN. No crash. Just subtly wrong outputs.

CAPACITY DROPPING PRIORITY




```python
@dataclass
class CapacityMetrics:
    expert_id: int
    capacity: int  # max tokens
    load: float    # utilization
                   0.0-1.0+
    overflow: int  # tokens beyond
capacity

def drop_tokens(self, assigned:
Tensor, metrics):
  for expert in overloaded_experts:
     sorted = torch.argsort(self,
part.gate_scores, descending=True)
     keep = sorted[:expert.capacity]
     dropped = sorted[expert.capacity:]
     redistribute(dropped,
underloaded_experts)
```

⚠ TOKENS ARE NEVER DISCARDED ⚠

Overflow tokens are intelligently redistributed, not dropped.

Teachable Abstractions

Three reusable PyTorch patterns from our implementation

ScheduleStep Enum

Unit-test pipeline schedules without GPUs. pytest tests/torch/test_dualpipe.py — no CUDA needed.

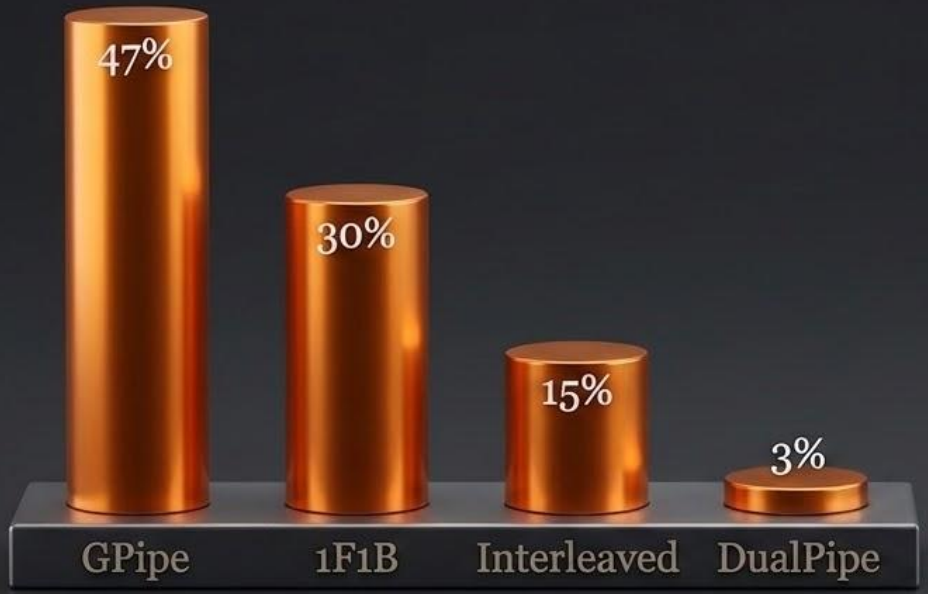
CapacityMetrics Dataclass

Decouple monitoring from routing logic. Drop-in for any torch MoE. Works with TensorBoard.

ExpertSpecializationTracker

Detect expert collapse before loss degrades. EMA-smoothed, W&B integration ready.

Performance: Why DeepSeek, and Does It Generalize?



Why DeepSeek Specifically?

671B + 256 experts demands PP=8+ while smaller models see diminishing returns.

Generalizes To:

Llama-3 405B, Mixtral, Grok

Scheduler is model-agnostic

Existing Research & Applications References: **1F1B** (Narayanan et al., 2021), **ZeRO** (Rajbhandari et al., 2020), and Megatron-LM compatibility

Key Takeaways

- The scheduler is the key leverage point — DualPipe cuts bubble from 47% to 3%
- The warmup formula isn't in the paper — off by one = torch.distributed deadlock
- Five silent bugs: K_pe sharing, bias timing, sigmoid routing, causal mask, capacity drop
- Teachable abstractions: ScheduleStep, CapacityMetrics, Expert SpecializationTracker