# DualPipe

DualPipe is an innovative bidirectional pipeline parallelism algorithm introduced in the DeepSeek-V3 Technical Report. It achieves full overlap of forward and backward computation-communication phases, also reducing pipeline bubbles. For detailed information on computation-communication overlap, please refer to the profile data.

Pipeline Bubbles and Memory Usage Comparison

Method	Bubble	Parameter	Activation
1F1B	(PP-1)(F+B)	1×	PP
ZB1P	(PP-1)(F+B-2W)	1×	PP
DualPipe	(PP/2-1)(F&B+B-3W)	2×	PP+1

F denotes the execution time of a forward chunk, B denotes the execution time of a full backward chunk, W denotes the execution time of a "backward for weights" chunk, and F&B denotes the execution time of two mutually overlapped forward and backward chunks.

### About

A bidirectional pipeline parallelism algorithm for computation-communication overlap in V3/R1 training

` DualPipe was created and developed by Jiashi Li and Chengqi Deng and Wenfeng Liang.`

# Profiling Data in DeepSeek Infra

Here, we publicly share profiling data from our training and inference framework to help the community better understand the communication-computation overlap strategies and low-level implementation details. The profiling data was captured using the PyTorch

Profiler. After downloading, you can visualize it directly by navigating to chrome://tracing in the Chrome browser (or edge://tracing in the Edge browser). Notice that we simulate an absolutely balanced MoE routing strategy for profiling.

## Training

The training profile data demonstrates our overlapping strategy for a pair of individual forward and backward chunks in DualPipe. Each chunk contains 4 MoE (Mixture of Experts) layers. The parallel configuration aligns with DeepSeek-V3 pretraining settings: EP64, TP1 with 4K sequence length. And the PP communication is not included during profilng for simplicity.

## Inference

### Prefilling

For prefilling, the profile employs EP32 and TP1 (in line with DeepSeek V3/R1 's actual online deployment), with a prompt length set to 4K and a batch size of 16K tokens per GPU. In our prefilling stage, we utilize two micro-batches to overlap computation and all-to-all communication, while ensuring that the attention computation load is balanced across the two micro-batches — meaning that the same prompt may be split between them.

### Decoding

For decoding, the profile employs EP128, TP1, and a prompt length of 4K (closely matching the actual online deployment configuration), with a batch size of 128 requests per GPU. Similar to prefilling, decoding also leverages two micro-batches for overlapping computation and all-to-all communication. However, unlike in prefilling, the all-to-all communication during decoding does not occupy GPU SMs: after RDMA messages are issued, all GPU SMs are freed, and the system waits for the all-to-all communication to complete after the computation has finished. For more information about the all-to-all implementation, please refer to DeepEP.

# Expert Parallelism Load Balancer (EPLB)

When using expert parallelism (EP), different experts are assigned to different GPUs. Because the load of different experts may vary depending on the current workload, it is important to keep the load of different GPUs balanced. As described in the DeepSeek-V3 paper, we adopt a redundant experts strategy that duplicates heavy-loaded experts. Then, we heuristically pack the duplicated experts to GPUs to ensure load balancing across different GPUs. Moreover, thanks to the group-limited expert routing used in DeepSeek-V3, we also attempt to place the experts of the same group to the same node to reduce inter-node data traffic, whenever possible.

To facilitate reproduction and deployment, we open-source our deployed EP load balancing algorithm in eplb.py. The algorithm computes a balanced expert replication and placement plan based on the estimated expert loads. Note that the exact method to predict the loads of experts is out of this repo's scope. A common method is to use moving average of historical statistics.

## The Algorithm

The load balancing algorithm comes with two policies used for different cases.

## Hierarchical Load Balancing

When the number of server nodes divides the number of expert groups, we use the hierarchical load balancing policy to harness the group-limited expert routing. We first pack the expert groups to nodes evenly, ensuring the loads of different nodes are balanced. Then, we replicate the experts within each node. Finally, we pack the replicated experts to individual GPUs to ensure different GPUs are load-balanced. The hierarchical load balancing policy can be used in prefilling stage with a smaller expert-parallel size.

### Global Load Balancing

In other cases, we use the global load balancing policy that replicates the experts globally regardless of expert groups, and pack the replicated experts to individual GPUs. This policy can be adopted in decoding stage with a larger expert-parallel size.

# Fire-Flyer File system

The Fire-Flyer File System (3FS) is a high-performance distributed file system designed to address the challenges of AI training and inference workloads. It leverages modern SSDs and RDMA networks to provide a shared storage layer that simplifies development of distributed applications. Key features and benefits of 3FS include:

- Performance and Usability

    - Disaggregated Architecture Combines the throughput of thousands of SSDs and the network bandwidth of hundreds of storage nodes, enabling applications to access storage resource in a locality-oblivious manner.
    - Strong Consistency Implements Chain Replication with Apportioned Queries (CRAQ) for strong consistency, making application code simple and easy to reason about.
    - File Interfaces Develops stateless metadata services backed by a transactional key-value store (e.g., FoundationDB). The file interface is well known and used everywhere. There is no need to learn a new storage API.

- Diverse Workloads

    - Data Preparation Organizes outputs of data analytics pipelines into hierarchical directory structures and manages a large volume of intermediate outputs efficiently.
    - Dataloaders Eliminates the need for prefetching or shuffling datasets by enabling random access to training samples across compute nodes.
    - Checkpointing Supports high-throughput parallel checkpointing for large-scale training.
    - KVCache for Inference Provides a cost-effective alternative to DRAM-based caching, offering high throughput and significantly larger capacity.

## Performance

1. Peak throughput

The following figure demonstrates the throughput of read stress test on a large 3FS cluster. This cluster consists of 180 storage nodes, each equipped with 2×200Gbps InfiniBand NICs and sixteen 14TiB NVMe SSDs. Approximately 500+ client nodes were used for the read stress test, with each client node configured with 1x200Gbps InfiniBand NIC. The final aggregate read throughput reached approximately 6.6 TiB/s with background traffic from training jobs.

2. GraySort

We evaluated smallpond using the GraySort benchmark, which measures sort performance on large-scale datasets. Our implementation adopts a two-phase approach: (1) partitioning data via shuffle using the prefix bits of keys, and (2) in-partition sorting. Both phases read/write data from/to 3FS.

The test cluster comprised 25 storage nodes (2 NUMA domains/node, 1 storage service/NUMA, 2×400Gbps NICs/node) and 50 compute nodes (2 NUMA domains, 192 physical cores, 2.2 TiB RAM, and 1×200 Gbps NIC/node). Sorting 110.5 TiB of data across 8,192 partitions completed in 30 minutes and 14 seconds, achieving an average throughput of 3.66 TiB/min.

3. KVCache

KVCache is a technique used to optimize the LLM inference process. It avoids redundant computations by caching the key and value vectors of previous tokens in the decoder layers. The top figure demonstrates the read throughput of all KVCache clients, highlighting both peak and average values, with peak throughput reaching up to 40 GiB/s. The bottom figure presents the IOPS of removing ops from garbage collection (GC) during the same time period.