

零冗余优化器（Zero Redundancy Data Parallelism, ZeRO）的目标是针对模型状态的存储进行去除冗余的优化^[173-175]。ZeRO 使用分区的方法，即将模型状态量分割成多个分区，每个计算设备只保存其中的一部分。这样整个训练系统内只需要维护一份模型状态，减少了内存消耗和通信开销。具体来说，如图4.17所示，ZeRO 包含以下三种方法。

(1) 对 Adam 优化器状态进行分区，图4.17 中的 P_{os} 部分。模型参数和梯度依然是每个计算设备保存一份。此时，每个计算设备所需内存是 $4\phi + \frac{12\phi}{N}$ 字节，其中 N 是计算设备总数。当 N 比较大时，每个计算设备占用内存趋向于 $4\phi B$ ，也就是 $16\phi B$ 的 $\frac{1}{4}$ 。

(2) 对模型梯度进行分区，图4.17 中的 P_{os+g} 部分。模型参数依然是每个计算设备保存一份。此时，每个计算设备所需内存是 $2\phi + \frac{2\phi+12\phi}{N}$ 字节。当 N 比较大时，每个计算设备占用内存趋向于 $2\phi B$ ，也就是 $16\phi B$ 的 $1/8$ 。

(3) 对模型参数进行分区，图4.17 中的 P_{os+g+p} 部分。此时，每个计算设备所需内存是 $\frac{16\phi}{N} B$ 。当 N 比较大时，每个计算设备占用内存趋向于 0。

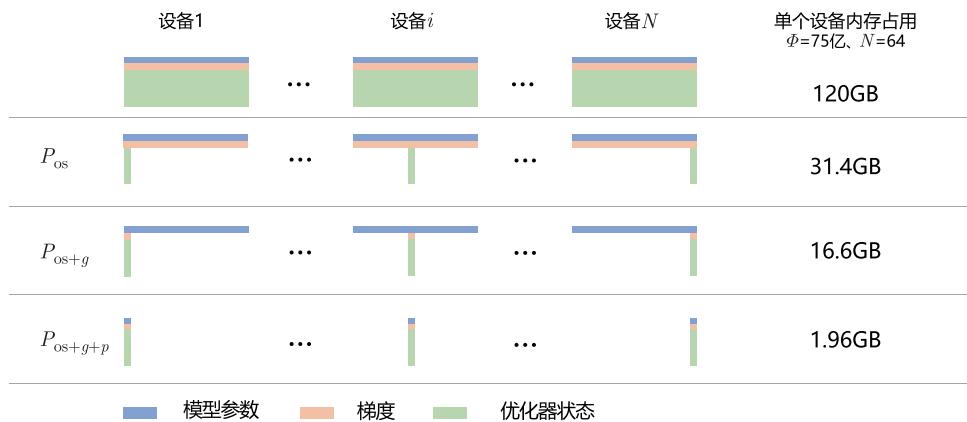


图 4.17 三种 ZeRO 方法的单个设备内存占用

在 DeepSpeed 框架中， P_{os} 对应 Zero-1， P_{os+g} 对应 Zero-2， P_{os+g+p} 对应 Zero-3。文献 [175] 中也对 ZeRO 优化方法所带来的通信量增加的情况进行了分析，Zero-1 和 Zero-2 对整体通信量没有影响，虽然对通信有一定延迟影响，但是整体性能受到的影响很小。Zero-3 所需的通信量则是正常通信量的 1.5 倍。

PyTorch 中也实现了 ZeRO 优化方法，可以使用 ZeroRedundancyOptimizer 调用，也可与“torch.nn.parallel.DistributedDataParallel”结合使用，以减少每个计算设备的内存峰值消耗。使用 ZeroRedundancyOptimizer 的参考代码如下所示：

```

import os
import torch
import torch.distributed as dist
import torch.multiprocessing as mp
import torch.nn as nn
import torch.optim as optim
from torch.distributed.optim import ZeroRedundancyOptimizer
from torch.nn.parallel import DistributedDataParallel as DDP

def print_peak_memory(prefix, device):
    if device == 0:
        print(f"[{prefix}]: {torch.cuda.max_memory_allocated(device) // 1e6}MB")

def example(rank, world_size, use_zero):
    torch.manual_seed(0)
    torch.cuda.manual_seed(0)
    os.environ['MASTER_ADDR'] = 'localhost'
    os.environ['MASTER_PORT'] = '29500'
    # 创建默认进程组
    dist.init_process_group("gloo", rank=rank, world_size=world_size)

    # 创建本地模型
    model = nn.Sequential(*[nn.Linear(2000, 2000).to(rank) for _ in range(20)])
    print_peak_memory("Max memory allocated after creating local model", rank)

    # 构建DDP模型
    ddp_model = DDP(model, device_ids=[rank])
    print_peak_memory("Max memory allocated after creating DDP", rank)

    # 定义损失函数和优化器
    loss_fn = nn.MSELoss()
    if use_zero:
        optimizer = ZeroRedundancyOptimizer( # 这里使用了ZeroRedundancyOptimizer
            ddp_model.parameters(),
            optimizer_class=torch.optim.Adam, # 包装了Adam
            lr=0.01
        )
    else:
        optimizer = torch.optim.Adam(ddp_model.parameters(), lr=0.01)

    # 前向传播
    outputs = ddp_model(torch.randn(20, 2000).to(rank))
    labels = torch.randn(20, 2000).to(rank)
    # 反向传播
    loss_fn(outputs, labels).backward()

    # 更新参数
    print_peak_memory("Max memory allocated before optimizer.step()", rank)

```

执行上述代码，可以得到如下输出：

```
==== Using ZeroRedundancyOptimizer ====
Max memory allocated after creating local model: 335.0MB
Max memory allocated after creating DDP: 656.0MB
Max memory allocated before optimizer step(): 992.0MB
Max memory allocated after optimizer step(): 1361.0MB
params sum is: -3453.6123046875
params sum is: -3453.6123046875
==== Not Using ZeroRedundancyOptimizer ====
Max memory allocated after creating local model: 335.0MB
Max memory allocated after creating DDP: 656.0MB
Max memory allocated before optimizer step(): 992.0MB
Max memory allocated after optimizer step(): 1697.0MB
params sum is: -3453.6123046875
params sum is: -3453.6123046875
```

可以看到，每次迭代之后，无论是否使用 ZeroRedundancyOptimizer，模型参数都使用同样的内存。在启用 ZeroRedundancyOptimizer 封装 Adam 优化器后，优化器的 step() 操作的内存峰值消耗是 Adam 内存消耗的一半。

4.3 分布式训练的集群架构

分布式训练需要使用由多台服务器组成的计算集群（Computing Cluster），而集群的架构也需要根据分布式系统、大语言模型结构、优化算法等综合因素进行设计。分布式训练集群属于高性能计算集群（High Performance Computing Cluster, HPC），其目标是提供海量的计算能力。在由高速网络组成的高性能计算上构建分布式训练系统，主要有两种常见架构：参数服务器架构和去中心化架构。

本章介绍高性能计算集群的典型硬件组成，并在此基础上介绍分布式训练系统所采用的参数服务器架构和去中心化架构。

4.3.1 高性能计算集群的典型硬件组成

典型的用于分布式训练的高性能计算集群的硬件组成如图4.18 所示。整个计算集群包含大量带有计算加速设备的服务器。每个服务器中往往有多个计算加速设备（通常为 2~16 个）。多个服务器会被放置在一个机柜（Rack）中，服务器通过架顶交换机（Top of Rack Switch, ToR）连接网络。在架顶交换机满载的情况下，可以通过在架顶交换机间增加骨干交换机（Spine Switch）接入新的机柜。这种连接服务器的拓扑结构往往是一个多层次树（Multi-Level Tree）。

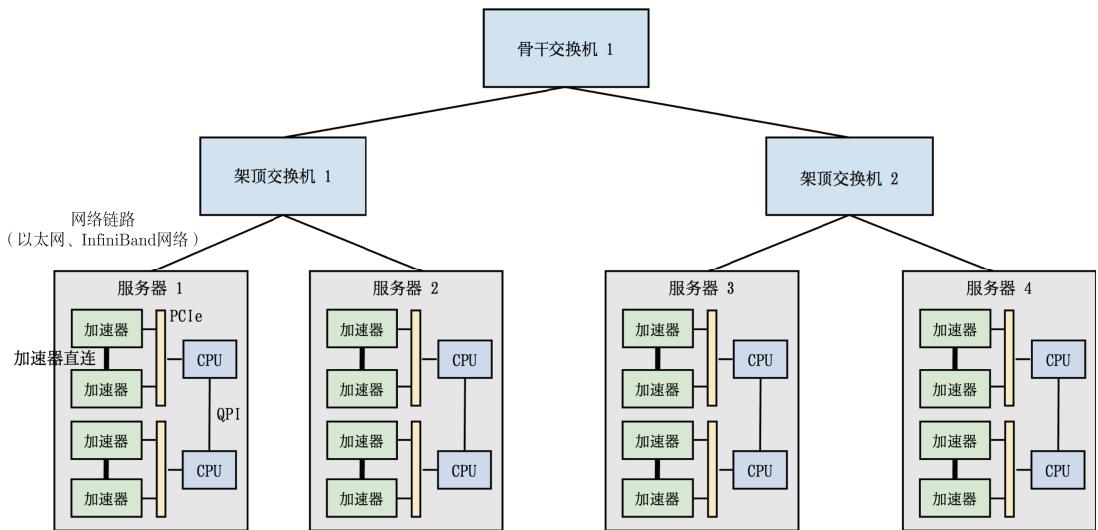
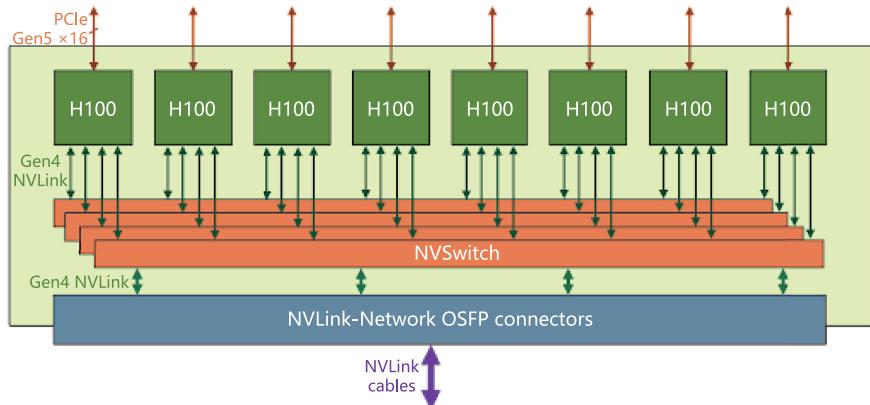


图 4.18 典型的用于分布式训练的高性能计算集群的硬件组成^[167]

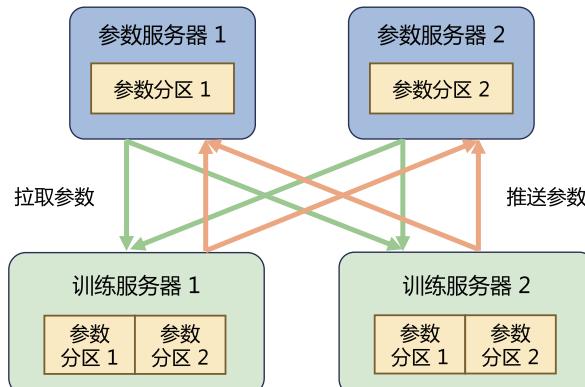
在多层树结构集群中跨机柜通信(Cross-Rack Communication)往往会有网络瓶颈。以包含 1750 亿个参数的 GPT-3 模型为例，每一个参数使用 32 位浮点数表示，在每一轮训练迭代中，每个模型副本会生成 700GB 的本地梯度数据。假如采用包含 1024 卡的计算集群，包含 128 个模型副本，那么至少需要传输 89.6TB ($700\text{GB} \times 128 = 89.6\text{TB}$) 的梯度数据。这会造成严重的网络通信瓶颈。因此，针对大语言模型分布式训练，通常采用**胖树**^[176] (Fat-Tree) 拓扑结构，试图实现网络带宽的无收敛。此外，采用**InfiniBand** (IB) 技术搭建高速网络，单个 InfiniBand 链路可以提供 200Gbps 或者 400Gbps 带宽。NVIDIA 的 DGX 服务器提供单机 1.6Tbps (200Gbps×8) 网络带宽，HGX 服务器网络带宽更是可以达到 3.2Tbps (400Gbps×8)。

单个服务器通常由 2~16 个计算加速设备组成，这些计算加速设备之间的通信带宽也是影响分布式训练的重要因素。如果这些计算加速设备通过服务器 PCIe 总线互联，则会造成服务器内部计算加速设备之间的通信瓶颈。PCIe 5.0 总线也只能提供 128GB/s 的带宽，而 NVIDIA H100 采用的 HBM 可以提供 3350GB/s 的带宽。因此，服务器内部通常采用异构网络架构。NVIDIA HGX H100 8-GPU 服务器采用**NVLink** 和 **NVSwitch** (NVLink 交换机) 技术，如图4.19 所示。每块 H100 GPU 都有多个 NVLink 端口，并连接到所有 (4 个) NVSwitch 上。每个 NVSwitch 都是一个完全无阻塞的交换机，完全连接所有 (8 块) H100 计算加速卡。NVSwitch 的这种完全连接的拓扑结构，使得服务器内任何 H100 加速卡之间都可以达到 900GB/s 的双向通信速度。

图 4.19 NVIDIA HGX H100 8-GPU NVLink 和 NVSwitch 连接框图^[167]

4.3.2 参数服务器架构

参数服务器（Parameter Server, PS）架构的分布式训练系统中有两种服务器角色：训练服务器和参数服务器。参数服务器需要提供充足的内存资源和通信资源，训练服务器需要提供大量的计算资源。图4.20 为参数服务器的分布式训练集群的示意图。该集群包括两个训练服务器和两个参数服务器。假设有一个可分为两个参数分区的模型，每个分区由一个参数服务器负责参数同步。在训练过程中，每个训练服务器都拥有完整的模型，将分配到此服务器的训练数据集切片（Dataset Shard）并进行计算，将得到的梯度推送到相应的参数服务器。参数服务器会等待两个训练服务器都完成梯度推送，再计算平均梯度并更新参数。之后，参数服务器会通知训练服务器拉取最新的参数，并开始下一轮训练迭代。

图 4.20 参数服务器的分布式训练集群的示意图^[167]

参数服务器架构的分布式训练过程可以细分为同步训练和异步训练两种模式。

- 同步训练：训练服务器在完成一个小批次的训练后，将梯度推送给参数服务器。参数服务器在收到所有训练服务器的梯度后，进行梯度聚合和参数更新。
- 异步训练：训练服务器在完成一个小批次的训练后，将梯度推送给参数服务器。参数服务器不再等待接收所有训练服务器的梯度，而是直接基于已收到的梯度进行参数更新。

在同步训练的过程中，参数服务器会等待所有训练服务器完成当前小批次的训练，有诸多的等待或同步机制，导致整个训练速度较慢。异步训练去除了训练过程中的等待机制，训练服务器可以独立进行参数更新，极大地加快了训练速度。引入异步更新的机制会导致训练效果有所波动。应根据具体情况和需求选择适合的训练模式。

4.3.3 去中心化架构

去中心化（Decentralized Network）架构采用集合通信实现分布式训练系统。在去中心化架构中，没有中央服务器或控制节点，而是由节点之间进行直接通信和协调。这种架构的好处是可以减少通信瓶颈，提高系统的可扩展性。由于节点之间可以并行地训练和通信，去中心化架构可以显著降低通信开销，并减少通信墙的影响。在分布式训练过程中，节点之间需要周期性地交换参数更新和梯度信息。可以通过集合通信（Collective Communication, CC）技术实现分布式训练，常用通信原语包括 Broadcast、Scatter、Reduce、All Reduce、Gather、All Gather、Reduce Scatter、All to All 等。4.2 节介绍的大语言模型训练所使用的分布式训练并行策略，大多使用去中心化架构，并利用集合通信实现。

下面介绍一些常见的集合通信原语。

(1) **Broadcast**: 主节点把自身的数据发送到集群中的其他节点。Broadcast 在分布式训练系统中常用于网络参数的初始化。如图4.21 所示，计算设备 1 对大小为 $1 \times N$ 的张量进行广播，最终每张卡输出均为 $[1 \times N]$ 的矩阵。

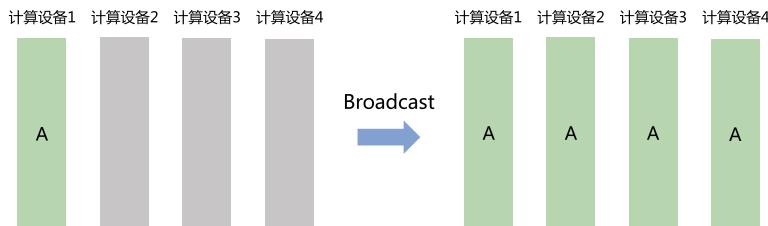


图 4.21 集合通信 Broadcast 原语示例

(2) **Scatter**: 主节点对数据进行划分并散布至其他指定的节点。Scatter 与 Broadcast 非常相似，不同的是，Scatter 是将数据的不同部分按需发送给所有的进程。如图4.22 所示，计算设备 1 将大小为 $1 \times N$ 的张量分为 4 份后发送到不同节点。

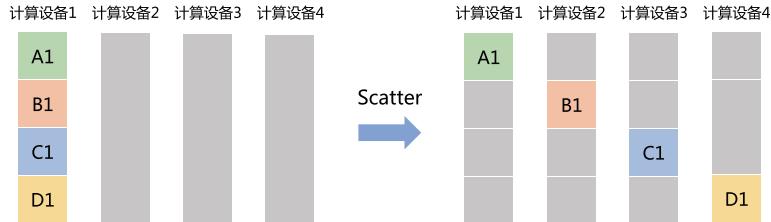


图 4.22 集合通信 Scatter 原语示例

(3) **Reduce**: 是一系列简单运算操作的统称, 将不同节点上的计算结果进行聚合 (Aggregation), 可以细分为 Sum、Min、Max、Prod、Lor 等类型的归约操作。如图4.23 所示, Reduce Sum 操作将所有计算设备上的数据汇聚到计算设备 1, 并执行求和操作。

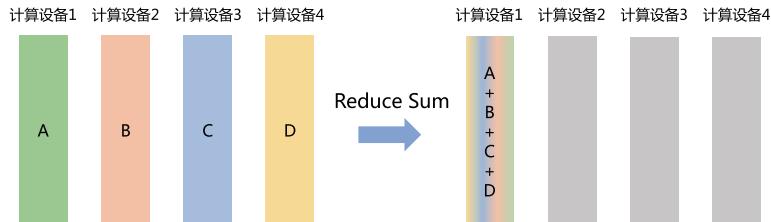


图 4.23 集合通信 Reduce Sum 原语示例

(4) **All Reduce**: 在所有的节点上都应用同样的 Reduce 操作。可以细分为 Sum、Min、Max、Prod、Lor 等类型的归约操作。All Reduce 操作可通过单节点上的“Reduce + Broadcast”操作完成。如图4.24 所示, All Reduce Sum 操作将所有计算设备上的数据汇聚到各个计算设备中, 并执行求和操作。

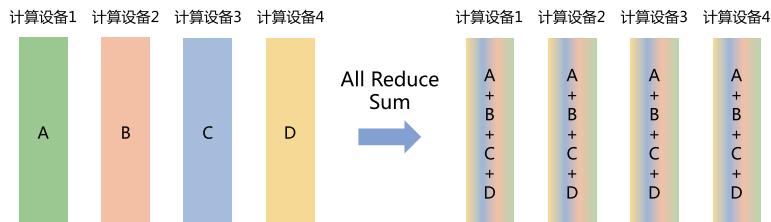


图 4.24 集合通信 All Reduce Sum 原语示例

(5) **Gather**: 将多个节点上的数据收集到单个节点上, 可以将 Gather 理解为反向的 Scatter。如图4.25 所示, Gather 操作将所有计算设备上的数据收集到计算设备 1 中。

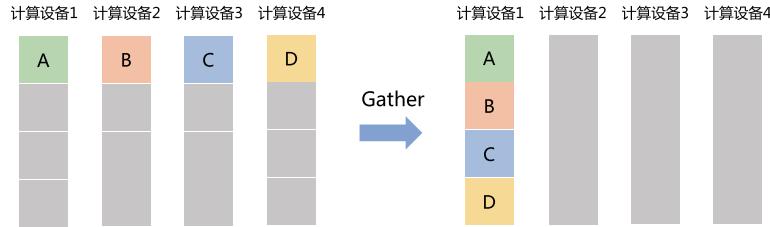


图 4.25 集合通信 Gather 原语示例

(6) **All Gather**: 每个节点都收集所有其他节点上的数据, All Gather 相当于一个 Gather 操作之后跟着一个 Broadcast 操作。如图4.26 所示, All Gather 操作将所有计算设备上的数据收集到每个计算设备中。

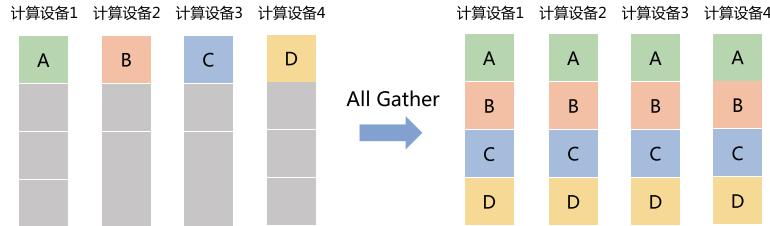


图 4.26 集合通信 All Gather 原语示例

(7) **Reduce Scatter**: 将每个节点中的张量切分为多个块, 每个块被分配给不同的节点。接收到的块会在每个节点上进行特定的操作, 例如求和、取平均值等。如图4.27 所示, 每个计算设备都将其中的张量切分为 4 块, 并分发到 4 个不同的计算设备中, 每个计算设备分别对接收的分块进行特定操作。

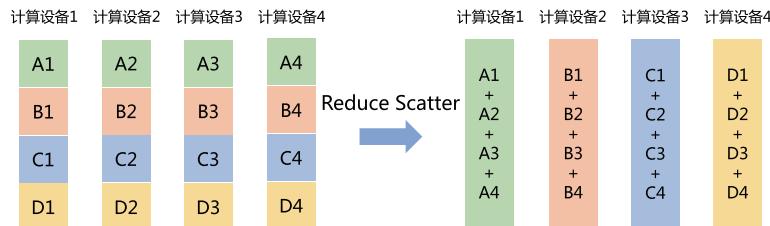


图 4.27 集合通信 Reduce Scatter 原语示例

(8) **All to All**: 将每个节点的张量切分为多个块, 每个块分别发送给不同的节点。如图4.28 所示, 每个计算设备都将其中的张量切分为 4 块, 并分发到 4 个不同的计算设备中。

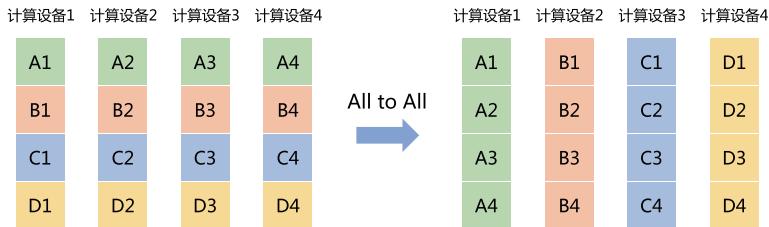


图 4.28 集合通信 All to All 原语示例

分布式集群中的网络硬件多种多样，包括以太网、InfiniBand 网络等。PyTorch 等深度学习框架通常不直接操作硬件，而是使用通信库。常用的通信库包括 MPI、GLOO、NCCL 等，可以根据具体情况进行选择和配置。MPI（Message Passing Interface）是一种广泛使用的并行计算通信库，常用于在多个进程之间进行通信和协调。GLOO 是 Facebook 推出的一个类似 MPI 的集合通信库（Collective Communications Library），也大体遵照 MPI 提供的接口规定，实现了包括点对点通信、集合通信等相关接口，支持在 CPU 和 GPU 上的分布式训练。NCCL（NVIDIA Collective Communications Library）是 NVIDIA 开发的高性能 GPU 间通信库，专门用于在多个 GPU 之间进行快速通信和同步，因为 NCCL 是 NVIDIA 基于自身硬件定制的，能做到更有针对性且更便于优化，故在 NVIDIA 硬件上，NCCL 的效果往往比其他通信库更好。GLOO、MPI 和 NCCL 在 CPU 和 GPU 环境下对通信原语的支持情况如表4.1 所示。在进行分布式训练时，根据所使用的硬件环境和需求，选择适当的通信库可以充分发挥硬件的优势并提高分布式训练的性能和效率。一般而言，如果在 CPU 集群上进行训练，则可选择使用 MPI 或 GLOO 作为通信库；而如果在 GPU 集群上进行训练，则可选择 NCCL 作为通信库。

表 4.1 GLOO、MPI 和 NCCL 在 CPU 和 GPU 环境下对通信原语的支持情况

通信原语	GLOO		MPI		NCCL	
	CPU	GPU	CPU	GPU	CPU	GPU
Send	✓	✗	✓	?	✗	✓
Receive	✓	✗	✓	?	✗	✓
Broadcast	✓	✓	✓	?	✗	✓
Scatter	✓	✗	✓	?	✗	✓
Reduce	✓	✗	✓	?	✗	✓
All Reduce	✓	✓	✓	?	✗	✓
Gather	✓	✗	✓	?	✗	✓
All Gather	✓	✗	✓	?	✗	✓
Reduce Scatter	✗	✗	✗	✗	✗	✓
All To All	✗	✗	✓	?	✗	✓
Barrier	✓	✗	✓	?	✗	✓

以 PyTorch 为例，介绍如何使用上述通信原语完成多计算设备间通信。先使用 “torch.distributed” 初始化分布式环境：

```
import os
from typing import Callable

import torch
import torch.distributed as dist

def init_process(rank: int, size: int, fn: Callable[[int, int], None], backend="gloo"):
    """ 初始化分布式环境 """
    os.environ["MASTER_ADDR"] = "127.0.0.1"
    os.environ["MASTER_PORT"] = "29500"
    dist.init_process_group(backend, rank=rank, world_size=size)
    fn(rank, size)
```

接下来使用 “torch.multiprocessing” 开启多个进程，本例中共开启了 4 个进程：

```

...
import torch.multiprocessing as mp

def func(rank: int, size: int):
    # 每个进程都将调用此函数
    continue

if __name__ == "__main__":
    size = 4
    processes = []
    mp.set_start_method("spawn")
    for rank in range(size):
        p = mp.Process(target=init_process, args=(rank, size, func))
        p.start()
        processes.append(p)

    for p in processes:
        p.join()

```

每个新开启的进程都会调用“init_process”，接下来调用用户指定的函数“func”。这里以 All Reduce 为例：

```

def do_all_reduce(rank: int, size: int):
    # 创建包含所有处理器的群组
    group = dist.new_group(list(range(size)))
    tensor = torch.ones(1)
    dist.all_reduce(tensor, op=dist.ReduceOp.SUM, group=group)
    # 可以是dist.ReduceOp.PRODUCT, dist.ReduceOp.MAX, dist.ReduceOp.MIN
    # 将输出所有秩为4的结果
    print(f"[{rank}] data = {tensor[0]}")

    ...

for rank in range(size):
    # 传递`hello_world`
    p = mp.Process(target=init_process, args=(rank, size, do_all_reduce))

    ...

```

根据 All Reduce 通信原语，在所有的节点上都应用同样的 Reduce 操作，可以得到如下输出：

```
[3] data = 4.0
[0] data = 4.0
[1] data = 4.0
[2] data = 4.0
```

4.4 DeepSpeed 实践

DeepSpeed^[172] 是一个由 Microsoft 公司开发的开源深度学习优化库，旨在提高大语言模型训练的效率和可扩展性，使研究人员和工程师能够更快地迭代和探索新的深度学习模型和算法。它采用了多种技术手段来加速训练，包括模型并行化、梯度累积、动态精度缩放、本地模式混合精度等。此外，DeepSpeed 还提供了一些辅助工具，例如分布式训练管理、内存优化和模型压缩，以帮助开发者更好地管理和优化大规模深度学习训练任务。DeepSpeed 是基于 PyTorch 构建的，因此将现有的 PyTorch 训练代码迁移到 DeepSpeed 上通常只需要进行简单的修改。这使得开发者可以快速利用 DeepSpeed 的优化功能来加速训练任务。DeepSpeed 已经在许多大规模深度学习项目中得到了应用，包括语言模型、图像分类、目标检测等领域。大语言模型 BLOOM^[31]（1750 亿个参数）和 MT-NLG^[134]（5400 亿个参数）都采用 DeepSpeed 框架完成训练。

DeepSpeed 的主要优势在于支持大规模神经网络模型、提供了更多的优化策略和工具。**DeepSpeed** 通过实现三种并行方法的灵活组合，即 ZeRO 支持的数据并行、流水线并行和张量并行，可以应对不同工作负载的需求。特别是通过 3D 并行性的支持，DeepSpeed 可以处理具有万亿个参数的超大规模模型。DeepSpeed 还引入了 **ZeRO-Offload**，使单个 GPU 能够训练比其显存容量大 10 倍的模型。为了充分利用 CPU 和 GPU 的内存来训练大语言模型，DeepSpeed 还扩展了 ZeRO-2。此外，DeepSpeed 还提供了稀疏注意力核（Sparse Attention Kernel），支持处理包括文本、图像和语音等长序列输入的模型。DeepSpeed 还集成了 1 比特 Adam 算法（1-bit Adam），该算法可以只使用原始 Adam 算法 1/5 的通信量，达到与 Adam 类似的收敛率，显著提高分布式训练的效率，降低通信开销。

DeepSpeed 的 3D 并行充分利用硬件架构特性，综合考虑了显存效率和计算效率。4.3 节介绍了分布式集群的硬件架构，截至 2023 年 9 月，分布式训练集群通常采用 NVIDIA DGX/HGX 节点，利用**胖树网络拓扑结构**构建计算集群。因此，每个节点内部 8 个计算加速设备之间具有非常高的通信带宽，节点之间的通信带宽则相对较低。由于张量并行是分布式训练策略中通信开销最大的，因此优先考虑将张量并行计算组放置在节点内以利用更大的节点内带宽。当张量并行组不能占满节点内的所有计算节点时，选择将数据并行组放置在节点内，否则就使用跨节点进行数据并行。流水线并行的通信量最低，因此可以使用跨节点的方式调度流水线的各个阶段，降低通信带宽的要求。每个数据并行组需要通信的梯度量随着流水线和模型并行的规模线性减小，因此总通信量少于单纯使用数据并行。此外，每个数据并行组会在局部的一小部分计算节点内部独立通信，组间

通信可以并行。通过减少通信量和增加局部性与并行性，数据并行通信的有效带宽有效增大。

图4.29 给出了 DeepSpeed 3D 并行策略示意图。图中给出了 32 个计算设备进行 3D 并行的例子。神经网络的各层分为 4 个流水线阶段。每个流水线阶段中的层在 4 个张量并行计算设备之间进一步划分。最后，每个流水线阶段有两个数据并行实例，使用 ZeRO 内存优化在这 2 个副本之间划分优化器状态量。

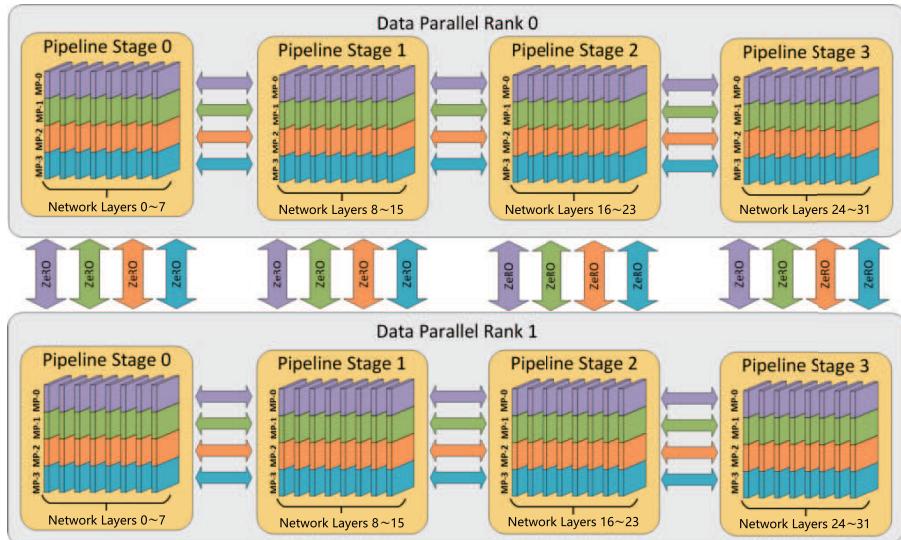


图 4.29 DeepSpeed 3D 并行策略示意图^[177]

DeepSpeed 软件架构如图4.30 所示，主要包含以下三部分。

(1) API: DeepSpeed 提供了易于使用的 API 接口，简化了训练模型和推断的过程。用户只需调用几个 API 接口即可完成任务。通过“initialize”接口可以初始化引擎，并在参数中配置训练参数、优化技术等。这些配置参数通常保存在名为“ds_config.json”的文件中。

(2) RunTime: RunTime 是 DeepSpeed 的核心运行时组件，使用 Python 语言实现，负责管理、执行和优化性能。它承担了将训练任务部署到分布式设备的功能，包括数据分区、模型分区、系统优化、微调、故障检测及检查点的保存和加载等任务。

(3) Ops: Ops 是 DeepSpeed 的底层内核组件，使用 C++ 和 CUDA 实现。它优化计算和通信过程，提供了一系列底层操作，包括 Ultrafast Transformer Kernels、Fuse LAN Kernels、Customary Deals 等。Ops 的目标是通过高效的计算和通信加速深度学习训练过程。

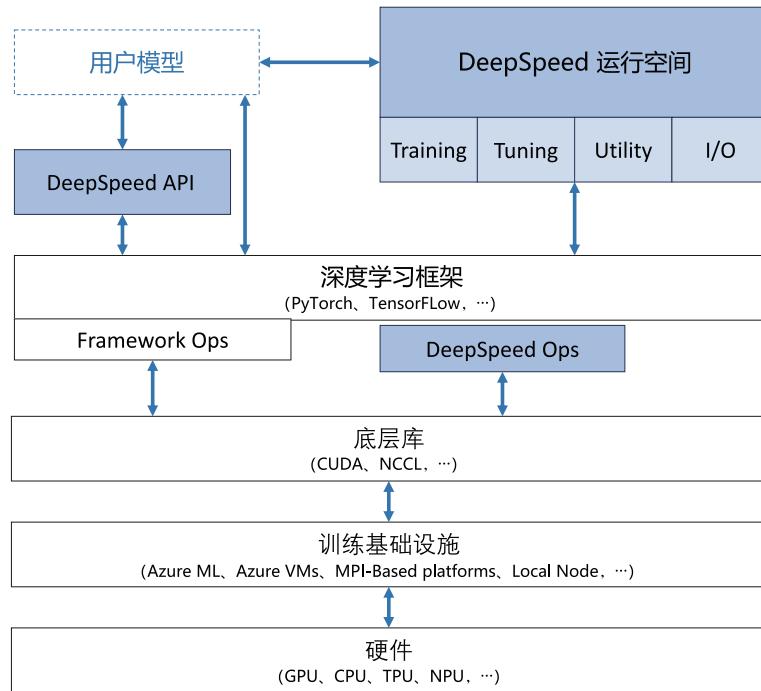


图 4.30 DeepSpeed 软件架构

4.4.1 基础概念

DeepSpeed 提供了分布式计算框架，首先需要明确几个重要的基础概念：主节点、节点编号、全局进程编号、局部进程编号和全局总进程数。DeepSpeed 主节点（`master_ip+master_port`）负责协调所有其他节点和进程的工作，由主节点所在服务器的 IP 地址和主节点进程的端口号来确定主节点。主节点还负责监控系统状态、处理任务分配、结果汇总等任务，因此是整个系统的关键部分。节点编号（`node_rank`）是系统中每个节点的唯一标识符，用于区分不同计算机之间的通信。全局进程编号（`rank`）是整个系统中的每个进程的唯一标识符，用于区分不同进程之间的通信。局部进程编号（`local_rank`）是单个节点内的每个进程的唯一标识符，用于区分同一节点内的不同进程之间的通信。全局总进程数（`world_size`）是整个系统中运行的所有进程的总数，用于确定可以并行完成多少工作及完成任务所需的资源数量。

在网络通信策略方面，DeepSpeed 提供了 MPI、GLOO、NCCL 等选项，可以根据具体情况选择和配置。在 DeepSpeed 配置文件中，在 optimizer 部分配置通信策略，以下是使用 1-Bit Adam 优化器的配置样例，配置中使用了 NCCL 通信库：

```
{  
    "optimizer": {  
        "type": "OneBitAdam",  
        "params": {  
            "lr": 0.001,  
            "betas": [  
                0.8,  
                0.999  
            ],  
            "eps": 1e-8,  
            "weight_decay": 3e-7,  
            "freeze_step": 400,  
            "cuda_aware": false,  
            "comm_backend_name": "nccl"  
        }  
    }  
}  
...  
}
```

DeepSpeed 中也支持多种类型 ZeRO 的分片机制，包括 ZeRO-0、ZeRO-1、ZeRO-2、ZeRO-3 以及 ZeRO-Infinity。ZeRO-0 禁用所有类型的分片，仅将 DeepSpeed 当作分布式数据并行使用；ZeRO-1 对优化器状态进行分片，占用内存为原始的 1/4，通信容量与数据并行性相同；ZeRO-2 对优化器状态和梯度进行分片，占用内存为原始的 1/8，通信容量与数据并行性相同；ZeRO-3 对优化器状态、梯度及模型参数进行分片，内存减少与数据并行度和复杂度成线性关系，同时通信容量是数据并行性的 1.5 倍；ZeRO-Infinity 是 ZeRO-3 的拓展，允许通过使用 NVMe 固态硬盘扩展 GPU 和 CPU 内存来训练大语言模型。

以下是 DeepSpeed 使用 ZeRO-3 配置参数的样例：

```
{
    "zero_optimization": {
        "stage": 3,
    },
    "fp16": {
        "enabled": true
    },
    "optimizer": {
        "type": "AdamW",
        "params": {
            "lr": 0.001,
            "betas": [
                0.8,
                0.999
            ],
            "eps": 1e-8,
            "weight_decay": 3e-7
        }
    },
    ...
}
```

如果希望在 ZeRO-3 的基础上继续使用 ZeRO-Infinity 将优化器状态和计算转移到 CPU 中，则可以在配置文件中按照如下方式配置：

```
{
    "zero_optimization": {
        "stage": 3,
        "offload_optimizer": {
            "device": "cpu"
        }
    },
    ...
}
```

甚至可以进一步将模型参数也装载到 CPU 内存中，在配置文件中按照如下方式配置：

```
{
    "zero_optimization": {
        "stage": 3,
        "offload_optimizer": {
            "device": "cpu"
        }
        "offload_param": {
            "device": "cpu"
        }
    },
    ...
}
```

如果希望将更多的内存装载到 NVMe 中，则可以在配置文件中按照如下方式配置：

```
{
    "zero_optimization": {
        "stage": 3,
        "offload_optimizer": {
            "device": "nvme",
            "nvme_path": "/nvme_data"
        }
        "offload_param": {
            "device": "nvme",
            "nvme_path": "/nvme_data"
        }
    },
    ...
}
```

4.4.2 LLaMA 分布式训练实践

LLaMA 模型是目前最流行、性能最强大的开源模型之一，基于 LLaMA 构造的模型生态可以覆盖绝大部分模型使用场景。在设置完必要的数据和环境配置后，本节将逐步演示如何使用 DeepSpeed 框架训练 LLaMA 模型。

DeepSpeed 可以很好地兼容 PyTorch 和 CUDA 的大多数版本，其安装过程通常无须指定特殊配置选项，直接通过 pip 命令完成。

```
pip install deepspeed
```

1. 训练数据配置

使用 PyTorch 和 transformers 库来设置预训练模型的数据加载器，以实现在单机或多机分布式训练环境中对数据的加载和采样。需要导入的模块如下。

- DataLoader 是 PyTorch 提供的工具，用于从数据集加载数据到模型进行训练或评估。
- RandomSampler 和 SequentialSampler 是 PyTorch 提供的两种采样器。RandomSampler 随机采样数据，而 SequentialSampler 顺序采样数据。
- DistributedSampler 是用于分布式训练的数据采样器。
- default_data_collator 是 transformers 库提供的默认数据收集器，用于将多个样本整合为一个批量数据。
- create_pretrain_dataset 是一个自定义函数，用于创建预训练数据集。

通过检查 args.local_rank 是否为 -1，代码会选择使用普通的采样器（单机）还是分布式采样器（多机）。DistributedSampler 确保在分布式训练环境中，每个进程或节点都能获得数据的一个不重复的子集，这使得分布式训练变为可能。而在单机环境中，使用常规的随机或顺序采样器即可。具体代码如下所示：

```

from torch.utils.data import DataLoader, RandomSampler, SequentialSampler
from torch.utils.data.distributed import DistributedSampler
from transformers import default_data_collator
from utils.data.data_utils import create_pretrain_dataset

# 数据准备
train_dataset, eval_dataset = create_pretrain_dataset(
    args.local_rank,
    args.data_path,
    args.data_split,
    args.data_output_path,
    args.seed,
    tokenizer,
    args.max_seq_len)

# DataLoader创建
if args.local_rank == -1:
    train_sampler = RandomSampler(train_dataset)
    eval_sampler = SequentialSampler(eval_dataset)
else:
    train_sampler = DistributedSampler(train_dataset)
    eval_sampler = DistributedSampler(eval_dataset)
train_dataloader = DataLoader(train_dataset,
                             collate_fn=default_data_collator,
                             sampler=train_sampler,
                             batch_size=args.per_device_train_batch_size)
eval_dataloader = DataLoader(eval_dataset,
                            collate_fn=default_data_collator,
                            sampler=eval_sampler,
                            batch_size=args.per_device_eval_batch_size)

```

2. 模型载入

使用 transformers 库加载和配置 LLaMA 模型及其相关的词元分析器。从 transformers 库中导入 LLaMA 模型、相应的词元分析器和模型配置后，使用 from_pretrained 方法加载预训练的 LLaMA 模型、词元分析器和配置。为了确保词元分析器可以处理各种文本的长度，还需要进行填充设置。如果词元分析器还没有指定填充符号，则将其设置为 [PAD]，并确定填充行为发生在句子的右侧。此外，为了保证模型能够正确地处理句子结束和填充，还为模型配置设置了结束符号和填充符号的 ID。最后，为了优化模型在硬件上的性能，还需要调整模型的词汇表嵌入大小，使其成为 8 的倍数。通过这些步骤，可以成功地加载并配置 LLaMA 模型，为后续的训练任务做好准备。具体代码如下：

```

from transformers import LlamaForCausalLM, LlamaTokenizer, LlamaConfig

# 载入词元分析器：将获得正确的词元分析器并根据模型设置填充词元
tokenizer = LlamaTokenizer.from_pretrained(
    model_name_or_path, fast_tokenizer=True)
if tokenizer.pad_token is None:
    # 判断tokenizer.eos_token不为None
    # 往词元分析器中加入特殊词元
    tokenizer.add_special_tokens({'pad_token': tokenizer.eos_token})
    tokenizer.add_special_tokens({'pad_token': '[PAD]'})
    tokenizer.padding_side = 'right'

model_config = LlamaConfig.from_pretrained(model_name_or_path)
model = LlamaForCausalLM.from_pretrained(model_name_or_path, config=model_config)

model.config.end_token_id = tokenizer.eos_token_id
model.config.pad_token_id = model.config.eos_token_id
model.resize_token_embeddings(int(
    8 *
    math.ceil(len(tokenizer) / 8.0))) # 设置词表大小为8的倍数

```

3. 优化器设置

DeepSpeed 库提供了高效的优化器算法，如 DeepSpeedCPUAdam 和 FusedAdam，这些算法经过特殊优化以提高在大规模数据和模型上的训练速度。优化器配置主要包含以下几个方面。

(1) 参数分组：通过 `get_optimizer_grouped_parameters` 函数将模型参数分为两组，一组使用权重衰减，另一组则不使用。这种参数分组有助于正则化模型，防止过拟合，并允许对特定参数应用不同的学习设置。

(2) 优化器选择：根据训练设置（如是否在 CPU 上进行模型参数卸载），可以选择使用 DeepSpeedCPUAdam 或 FusedAdam 优化器。这两种优化器都是对经典的 Adam 优化器进行优化和改进的版本，为大规模训练提供了高效性能。

(3) 学习率调度：不同于固定的学习率，学习率调度器在训练过程中动态调整学习率。例如，在训练初期快速提高学习率以加速收敛，在训练中后期逐渐降低学习率以获得更精细的优化。我们的配置考虑了预热步骤、训练的总步数及其他关键因素。

具体代码如下所示：

```

from transformers import get_scheduler
from deepspeed.ops.adam import DeepSpeedCPUAdam, FusedAdam

# 设置需要优化的模型参数及优化器
optimizer_grouped_parameters = get_optimizer_grouped_parameters(
    model, args.weight_decay, args.learning_rate)

AdamOptimizer = DeepSpeedCPUAdam if args.offload else FusedAdam
optimizer = AdamOptimizer(optimizer_grouped_parameters,
                         lr=args.learning_rate,
                         betas=(0.9, 0.95))

num_update_steps_per_epoch = math.ceil(
    len(train_dataloader) / args.gradient_accumulation_steps)
lr_scheduler = get_scheduler(
    name=args.lr_scheduler_type,
    optimizer=optimizer,
    num_warmup_steps=args.num_warmup_steps,
    num_training_steps=args.num_train_epochs * num_update_steps_per_epoch,
)

def get_optimizer_grouped_parameters(model,
                                      weight_decay,
                                      no_decay_name_list=[
                                          "bias", "LayerNorm.weight"
                                      ]):
    # 将权重分为两组，一组有权重衰减，另一组没有
    optimizer_grouped_parameters = [
        {
            "params": [
                p for n, p in model.named_parameters()
                if (not any(nd in n
                            for nd in no_decay_name_list) and p.requires_grad)
            ],
            "weight_decay": weight_decay,
        },
        {
            "params": [
                p for n, p in model.named_parameters()
                if (any(nd in n
                        for nd in no_decay_name_list) and p.requires_grad)
            ],
            "weight_decay": 0.0,
        },
    ]
    return optimizer_grouped_parameters

```

4. DeepSpeed 设置

在配置代码的开始，定义了两个关键参数 GLOBAL_BATCH_SIZE 和 MICRO_BATCH_SIZE。GLOBAL_BATCH_SIZE 定义了全局的批次大小。这通常是所有 GPU 加起来的总批次大小。MICRO_BATCH_SIZE 定义了每块 GPU 上的微批次大小。因为微批次处理每次只加载并处理一小部分数据，所以可以帮助大语言模型在有限的 GPU 内存中运行。训练配置函数 get_train_ds_config 主要包括以下内容。

- (1) ZeRO 优化配置：ZeRO 是 DeepSpeed 提供的一种优化策略，旨在减少训练中的冗余并加速模型的训练。其中的参数，如 offload_param 和 offload_optimizer，允许用户选择是否将模型参数或优化器状态卸载到 CPU。
- (2) 混合精度训练：通过设置 FP16 字段，使模型可以使用 16 位浮点数进行训练，加速训练过程并减少内存使用。
- (3) 梯度裁剪：通过 gradient_clipping 字段，可以防止训练过程中出现梯度爆炸问题。
- (4) 混合引擎配置：hybrid_engine 部分允许用户配置更高级的优化选项，如输出分词的最大数量和推理张量的大小。
- (5) TensorBoard 配置：使用 DeepSpeed 时，可以通过配置选项直接集成 TensorBoard，从而更方便地跟踪训练过程。

(6) 验证集配置函数 `get_eval_ds_config`: 此函数提供了 DeepSpeed 的验证集。与训练配置相比，验证集配置更为简洁，只需要关注模型推理阶段。

具体代码如下所示：

```
import torch
import deepspeed.comm as dist

GLOBAL_BATCH_SIZE = 32
MICRO_BATCH_SIZE = 4

def get_train_ds_config(offload,
                       stage=2,
                       enable_hybrid_engine=False,
                       inference_tp_size=1,
                       release_inference_cache=False,
                       pin_parameters=True,
                       tp_gather_partition_size=8,
                       max_out_tokens=512,
                       enable_tensorboard=False,
                       tb_path="",
                       tb_name ""):

    # 设置训练过程的DeepSpeed配置
    device = "cpu" if offload else "none"
    zero_opt_dict = {
        "stage": stage,
        "offload_param": {
            "device": device
        },
        "offload_optimizer": {
            "device": device
        },
        "stage3_param_persistence_threshold": 1e4,
        "stage3_max_live_parameters": 3e7,
        "stage3_prefetch_bucket_size": 3e7,
        "memory_efficient_linear": False
    }

    return {
        "train_batch_size": GLOBAL_BATCH_SIZE,
        "train_micro_batch_size_per_gpu": MICRO_BATCH_SIZE,
        "steps_per_print": 10,
        "zero_optimization": zero_opt_dict,
        "fp16": {
            "enabled": True,
            "loss_scale_window": 100
        },
        "gradient_clipping": 1.0,
        "prescale_gradients": False,
        "wall_clock_breakdown": False,
```