```
python dedup/generate_duplicate_pairs.py \
       --input_dir <prefix_path>/RedPajama_minhash/ \
       --out_file <prefix_path>/redpj_duplicates/duplicate_pairs.txt \
       --range <range> --bands <bands> --processes <n_processes>
```

（c）重复图构建及连通域查找（Duplicate Graph Construction & Search for Connected Components）：确定了重复的文档对之后，需要找到包含重复文档的连通域。例如，根据以下文档对：(A, B)、(A, C)、(A, E)，可以形成一个 (A, B, C, E) 的组，并仅保留该组中的一个文档。可以使用如下命令构建重复图：

```
python dedup/generate_connected_components.py \
      --input_dir <prefix_path>/redpj_duplicates \
      --out_file <prefix_path>/redpj_duplicates/connected_components.pickle
```

（d）生成最终重复列表（Generate Final List of Duplicates）：根据连通域构建一个查找表，以便稍后过滤重复项。生成最终重复列表的命令示例如下：

```
python preprocessing/shuffle_holdout.py pass1 \
       --input_dir <prefix_path>/RedPajama_norm/ \
       --duplicates <prefix_path>/redpj_duplicates/duplicates.pickle \
       --short_docs <prefix_path>/RedPajama_filtered.pickle\
       --out_dir <prefix_path>/SlimPajama/pass1
```

（4）文档交错与文档重排（Interleave & Shuffle）：大语言模型训练大多是在多源数据集上进行的，需要使用指定的权重混合这些数据源。SlimPajama 数据集默认从每个数据库中采样 1 轮，可以通过修改 preprocessing/datasets.py 参数更新采样权重。除了混合数据源，还要执行随机重排操作以避免任何顺序偏差。文档交错和文档重排的命令示例如下：

```
python preprocessing/shuffle_holdout.py pass1 \
      --input_dir <prefix_path>/RedPajama_norm/ \
      --duplicates <prefix_path>/redpj_duplicates/duplicates.pickle \
      --short_docs <prefix_path>/RedPajama_filtered.pickle \
      --out_dir <prefix_path>/SlimPajama/pass1
```

（5）训练集和保留集拆分（Split Dataset into Train and Holdout）：这一步主要是完成第二次随

机重排并创建保留集。为了加快处理速度，将源数据分成块并行处理。以下是命令示例：

```
for j in {1..20}
  do
      python preprocessing/shuffle_holdout.py pass2 "$((j-1))" "$j" "$j" \
             --input_dir <prefix_path>/SlimPajama/pass1 \
             --train_dir <prefix_path>/SlimPajama/train \
             --holdout_dir <prefix_path>/SlimPajama/holdout > $j.log 2>&1 &
  done
```

（6）训练集与保留集中相似数据去重（Deduplicate Train against Holdout）：最后一步是确保训练集和保留集之间没有重叠。为了去除训练集的污染，用 SHA256 哈希算法查找训练集和保留集之间的精确匹配项。然后，从训练集中过滤这些精确匹配项。以下是命令示例：

```
python dedup/dedup_train.py 1 \
       --src_dir <prefix_path>/SlimPajama/train \
       --tgt_dir <prefix_path>/SlimPajama/holdout \
       --out_dir <prefix_path>/SlimPajama/train_deduped
for j in {2..20}
do
    python dedup/dedup_train.py "$j" \
           --src_dir <prefix_path>/SlimPajama/train \
           --tgt_dir <prefix_path>/SlimPajama/holdout \
           --out_dir <prefix_path>/SlimPajama/train_deduped > $j.log 2>&1 &
done
```

4. 分布式训练

随着大语言模型参数量和所需训练数据量的急速增长，单个机器上有限的资源已无法满足其训练的要求。需要设计**分布式训练**系统来解决海量的计算和内存资源需求问题。在分布式训练系统环境下，需要将一个模型训练任务拆分成多个子任务，并将子任务分发给多个计算设备，从而解决资源瓶颈。如何才能利用数万个计算加速芯片的集群，训练千亿甚至万亿参数规模的大语言模型？这其中涉及集群架构、并行策略、模型架构、内存优化、计算优化等一系列的技术。

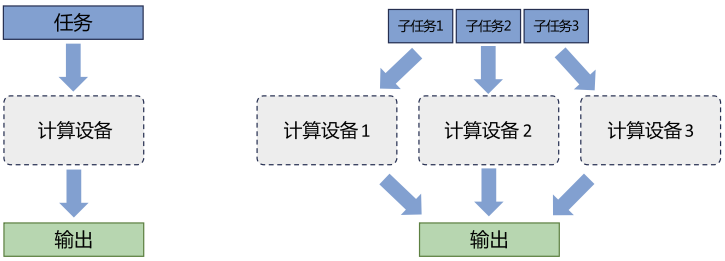
本章将介绍分布式机器学习系统的基础概念、分布式训练的并行策略、分布式训练的集群架构，并以 DeepSpeed 为例，介绍如何在集群上训练大语言模型。

4.1 分布式训练概述

分布式训练（Distributed Training）是指将机器学习或深度学习模型训练任务分解成多个子任务，并在多个计算设备上并行训练。图4.1 给出了单个计算设备和多个计算设备的示例，这里计算设备可以是中央处理器（Central Processing Unit，CPU）、图形处理器（Graphics Processing Unit，GPU）、张量处理器（Tensor Processing Unit，TPU），也可以是神经网络处理器（Neural network Processing Unit，NPU）。由于同一个服务器内部的多个计算设备之间可能并不共享内存，因此无论这些计算设备是处于一个服务器还是多个服务器中，其系统架构都属于分布式系统范畴。一个模型训练任务往往会有大量的训练样本作为输入，可以利用一个计算设备完成，也可以将整个模型的训练任务拆分成多个子任务，分发给不同的计算设备，实现并行计算。此后，还需要对每个计算设备的输出进行合并，最终得到与单个计算设备等价的计算结果。由于每个计算设备只需要负责子任务，并且多个计算设备可以并行执行，因此其可以更快速地完成整体计算，并最终实现对整个计算过程的加速。

促使人们设计分布式训练系统的一个最重要的原因是单个计算设备的算力已经不足以支撑模型训练。图4.2 给出了机器学习模型对于算力的需求以及同期单个计算设备能够提供的算力。机器学习模型快速发展，从 2013 年 AlexNet 被提出开始，到 2022 年拥有 5400 亿个参数的 PaLM 模型被提出，再到 2024 年拥有 6710 亿个参数的 DeepSeek-V2 发布，机器学习模型以每 18 个月增长 56

倍的速度发展。模型参数规模增大的同时，对训练数据量的要求也呈指数级增长，这更加剧了对算力的需求。然而，近几年，CPU 的算力增加已经远低于摩尔定律（Moore's Law），虽然计算加速设备（如 GPU、TPU 等）为机器学习模型提供了大量的算力，但是其增长速度仍然没有突破每 18 个月翻倍的摩尔定律。只有通过分布式训练系统才可以匹配模型不断增长的算力需求，满足机器学习模型的发展需要。



(a) 单个计算设备　　(b) 多个计算设备

图 4.1　单个计算设备和多个计算设备的示例

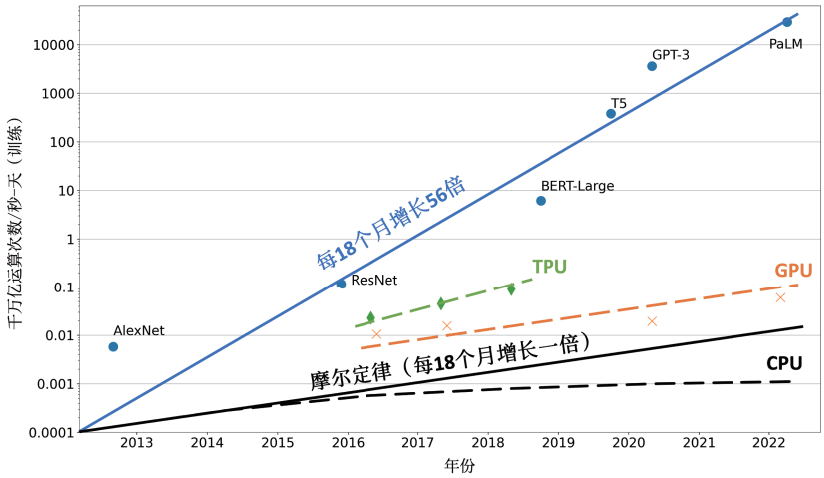


图 4.2　机器学习模型参数量增长和计算硬件的算力增长对比[167]

分布式训练的总体目标就是加快总的训练速度，减少模型训练的总体时间。总训练速度可以用式（4.1）简略估计：

$$\text{总训练速度} \propto \text{单设备计算速度} \times \text{计算设备总量} \times \text{多设备加速比} \tag{4.1}$$

其中，单设备计算速度主要由单块计算加速芯片的运算速度和数据 I/O 能力决定，对单设备训练

效率进行优化，主要的技术手段有混合精度训练、算子融合、梯度累加等；在分布式训练系统中，随着计算设备数量的增加，理论上峰值计算速度会增加，然而受通信效率的影响，计算设备数量增多会造成加速比急速降低；多设备加速比是由计算和通信效率决定的，需要结合算法和网络拓扑结构进行优化，分布式训练并行策略的主要目标就是提升分布式训练系统中的多设备加速比。

大语言模型的参数量和所使用的数据量都非常大，因此都采用了分布式训练架构完成训练。文献 [13] 仅在 GPT-3 的训练过程中提到全部使用 NVIDIA V100 GPU，文献 [29] 介绍了 OPT 使用 992 块 NVIDIA A100 80GB GPU，采用全分片数据并行（Fully Sharded Data Parallel）[168] 以及 Megatron-LM 张量并行（Tensor Parallelism）[169]，整体训练时间近两个月。BLOOM[31] 模型的研究人员则公开了更多在硬件和所采用的系统架构方面的细节。该模型的训练一共花费了 3.5 个月，使用 48 个计算节点。每个计算节点包含 8 块 NVIDIA A100 80GB GPU（总计 384 块 GPU），并且使用 4×NVLink 用于节点内部 GPU 之间的通信。节点之间采用 4 个 Omni-Path 100 Gbps 网卡构建的增强 8 维超立方体全局拓扑网络进行通信。文献 [34] 并没有给出 LLaMA 模型训练中所使用的集群的具体配置和网络拓扑结构，但是给出了不同参数规模的总 GPU 小时数。LLaMA 模型训练使用 NVIDIA A100 80GB GPU，LLaMA-7B 模型训练需要 82432 GPU 小时，LLaMA-13B 模型训练需要 135168 GPU 小时，LLaMA-33B 模型训练需要 530432 GPU 小时，而 LLaMA-65B 模型训练需要高达 1022362 GPU 小时。LLaMA 使用的训练数据量远超 OPT 和 BLOOM 模型，虽然模型参数量远小于上述两个模型，但是其所需计算量非常惊人。

通过使用分布式训练系统，大语言模型的训练周期可以从单计算设备花费几十年，缩短到使用数千个计算设备花费几十天。分布式训练系统需要克服计算墙、显存墙、通信墙等挑战，以确保集群内的所有资源得到充分利用，从而加速训练过程并缩短训练周期。

- **计算墙**：单个计算设备所能提供的计算能力与大语言模型所需的总计算量之间存在巨大差异。2022 年 3 月发布的 NVIDIA H100 SXM 的单卡 FP16 算力只有 2000 TFLOPS（Floating Point Operations Per Second），而 GPT-3 需要 314 ZFLOPS 的总计算量，两者相差了 8 个数量级。
- **显存墙**：单个计算设备无法完整存储一个大语言模型的参数。GPT-3 包含 1750 亿个参数，如果在推理阶段采用 FP32 格式进行存储，则需要 700GB 的计算设备内存空间，而 NVIDIA H100 GPU 只有 80GB 显存。
- **通信墙**：分布式训练系统中各计算设备之间需要频繁地进行参数传输和同步。由于通信的延迟和带宽限制，这可能成为训练的瓶颈。在 GPT-3 的训练过程中，如果分布式系统中存在 128 个模型副本，那么在每次迭代过程中至少需要传输 89.6TB 的梯度数据。截至 2023 年 8 月，单个 InfiniBand 链路仅能提供不超过 800Gbps 的带宽。

计算墙和显存墙源于单计算设备的计算和存储能力有限，与模型所需庞大计算和存储需求存在矛盾。这个问题可以通过采用分布式训练的方法解决，但分布式训练又会面临通信墙的挑战。在多机多卡的训练中，这些问题逐渐显现。随着大语言模型参数的增大，对应的集群规模也随之增

加，这些问题变得更加突出。同时，当大型集群进行长时间训练时，设备故障可能会影响或中断训练，对分布式系统的问题处理也提出了很高的要求。

4.2 分布式训练的并行策略

分布式训练系统的目标是将单节点模型训练转换成等价的分布式并行模型训练。对于大语言模型来说，训练过程就是根据数据和损失函数，利用优化算法对神经网络模型参数进行更新的过程。单个计算设备模型训练系统的结构如图4.3 所示，其主要由数据和模型两个部分组成。训练过程由多个数据**小批次**（Mini-batch）完成。图中数据表示一个数据小批次。训练系统会利用数据小批次根据损失函数和优化算法计算梯度，从而对模型参数进行修正。针对大语言模型多层神经网络的执行过程，可以由一个**计算图**（Computational Graph）表示。这个图有多个相互连接的算子（Operator），每个算子实现一个神经网络层（Neural Network Layer），而参数则代表了这个层在训练中所更新的权重。

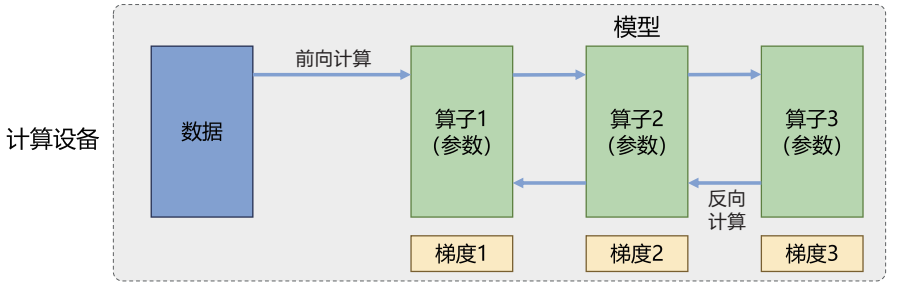


图 4.3　单个计算设备模型训练系统的结构

计算图的执行过程可以分为前向计算和反向计算两个阶段。**前向计算**的过程是将数据读入第一个算子，计算出相应的输出结构，然后重复这个前向计算过程，直到最后一个算子结束处理。**反向计算**的过程是根据损失函数和优化算法，对每个算子依次计算梯度，并利用梯度更新本地的参数。在反向计算结束后，该数据小批次的计算完成，系统就会读取下一个数据小批次，继续下一轮的模型参数更新。

根据单个计算设备模型训练系统的流程，可以看到，如果进行并行加速，可以从数据和模型两个维度进行考虑。可以对数据进行切分（Partition），并将同一个模型复制到多个设备上，并行执行不同的数据分片，这种方式通常被称为**数据并行**（Data Parallelism，DP）。还可以对模型进行划分，将模型中的算子分发到多个设备上分别完成处理，这种方式通常被称为**模型并行**（Model Parallelism，MP）。训练大语言模型时，往往需要同时对数据和模型进行切分，从而实现更高程度的并行，这种方式通常被称为**混合并行**（Hybrid Parallelism，HP）。

4.2.1 数据并行

在数据并行系统中，每个计算设备都有整个神经网络模型的模型副本（Model Replica），进行迭代时，每个计算设备只分配一个批次数据样本的子集，并根据该批次样本子集的数据进行网络模型的前向计算。假设一个批次的训练样本数为 N，使用 M 个计算设备并行计算，每个计算设备会分配到 N/M 个样本。前向计算完成后，每个计算设备都会根据本地样本计算损失误差，得到梯度 $\boldsymbol{G}_i$（i 为加速卡编号），并将本地梯度 $\boldsymbol{G}_i$ 进行广播。所有计算设备需要聚合其他加速卡给出的梯度值，然后使用平均梯度 $(\sum_{i=1}^{N} \boldsymbol{G}_i)/N$ 对模型进行更新，完成该批次训练。图4.4 给出了由两个计算设备组成的数据并行训练系统样例。

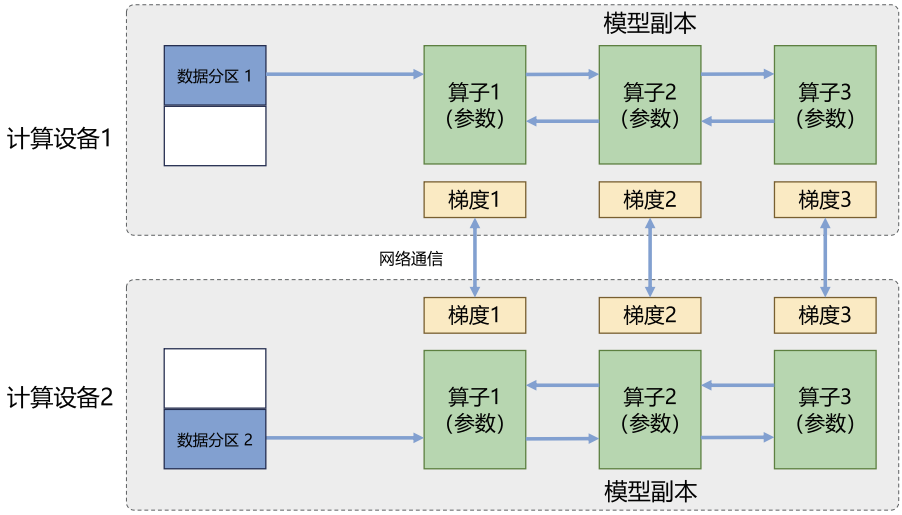


图 4.4　由两个计算设备组成的数据并行训练系统样例

数据并行训练系统可以通过增加计算设备，有效提升整体训练吞吐量，即**每秒全局批次数**（Global Batch Size Per Second）。与单个计算设备训练相比，其最主要的区别在于反向计算中的梯度需要在所有计算设备中进行同步，以保证每个计算设备上最终得到的是所有进程上梯度的平均值。常见的神经网络框架中都有数据并行方式的具体实现，包括 TensorFlow DistributedStrategy、PyTorch Distributed、Horovod DistributedOptimizer 等。由于基于 Transformer 结构的大语言模型中每个算子都依赖单个数据而非批次数据，因此数据并行并不会影响其计算逻辑。一般情况下，各训练设备中前向计算是独立的，不涉及同步问题。数据并行训练加速比最高，但要求每个设备上都备份一份模型，显存占用比较高。

使用 PyTorch DistributedDataParallel 实现单个服务器多加速卡训练的代码如下。首先，构造 DistributedSampler 类，将数据集的样本随机打乱并分配到不同计算设备上：

```python
class DistributedSampler(Sampler):
    def __init__(self, dataset, num_replicas=None, rank=None, shuffle=True, seed=0):
        if num_replicas is None:
            if not dist.is_available():
                raise RuntimeError("Requires distributed package to be available")
            num_replicas = dist.get_world_size()
        if rank is None:
            if not dist.is_available():
                raise RuntimeError("Requires distributed package to be available")
            rank = dist.get_rank()
        self.dataset = dataset  # 数据集
        self.num_replicas = num_replicas  # 进程个数，默认等于world_size(GPU块数)
        self.rank = rank   # 当前属于哪个进程/哪块GPU
        self.epoch = 0
        self.num_samples = int(math.ceil(len(self.dataset) * 1.0 / self.num_replicas))
                                            # 每个进程的样本个数
        self.total_size = self.num_samples * self.num_replicas  # 数据集总样本的个数
        self.shuffle = shuffle  # 是否要打乱数据集
        self.seed = seed

    def __iter__(self):
        # 1. Shuffle处理：打乱数据集顺序
        if self.shuffle:
            # 根据训练轮数和种子数进行混淆
            g = torch.Generator()
            # 这里self.seed是一个定值，通过set_epoch改变self.epoch可以改变我们的初始化种子
            # 这就可以让每一轮训练中数据集的打乱顺序不同
            # 使每一轮训练中每一块GPU得到的数据都不一样，这有利于更好的训练
            g.manual_seed(self.seed + self.epoch)
            indices = torch.randperm(len(self.dataset), generator=g).tolist()
        else:
            indices = list(range(len(self.dataset)))

        # 数据补充
        indices += indices[:(self.total_size - len(indices))]
        assert len(indices) == self.total_size

        # 分配数据
        indices = indices[self.rank:self.total_size:self.num_replicas]
        assert len(indices) == self.num_samples
```

```
        return iter(indices)

    def __len__(self):
        return self.num_samples

    def set_epoch(self, epoch):
        r"""
        设置此采样器的训练轮数
        当:attr:`shuffle=True`时，确保所有副本在每个轮数使用不同的随机顺序
        否则，此采样器的下一次迭代将产生相同的顺序

        Arguments:
            epoch (int): 训练轮数
        """
        self.epoch = epoch
```

利用 DistributedSampler 类构造的完整的训练程序样例 main.py 如下：

```
import argparse
import os
import shutil
import time
import warnings
import numpy as np

warnings.filterwarnings('ignore')

import torch
import torch.nn as nn
import torch.nn.parallel
import torch.backends.cudnn as cudnn
import torch.distributed as dist
import torch.optim
import torch.utils.data
import torch.utils.data.distributed
from torch.utils.data.distributed import DistributedSampler

from models import DeepLab
from dataset import Cityscaples

# 参数设置
parser = argparse.ArgumentParser(description='DeepLab')

parser.add_argument('-j', '--workers', default=4, type=int, metavar='N',
                    help='number of data loading workers (default: 4)')
parser.add_argument('--epochs', default=100, type=int, metavar='N',
                    help='number of total epochs to run')
parser.add_argument('--start-epoch', default=0, type=int, metavar='N',
                    help='manual epoch number (useful on restarts)')
parser.add_argument('-b', '--batch-size', default=3, type=int,
                    metavar='N')
parser.add_argument('--local_rank', default=0, type=int,
                    help='node rank for distributed training')


args = parser.parse_args()
torch.distributed.init_process_group(backend="nccl") # 初始化


print("Use GPU: {} for training".format(args.local_rank))

# 创建模型
model = DeepLab()

torch.cuda.set_device(args.local_rank) # 当前显卡
model = model.cuda()
model = torch.nn.parallel.DistributedDataParallel(model, device_ids=[args.local_rank],
    output_device=args.local_rank, find_unused_parameters=True) # 数据并行

criterion = nn.CrossEntropyLoss().cuda()

optimizer = torch.optim.SGD(model.parameters(), args.lr,
```

通过以下命令行启动上述程序：

```
CUDA_VISIBLE_DEVICES=0,1 python -m torch.distributed.launch --nproc_per_node=2 main.py
```

4.2.2 模型并行

模型并行往往用于解决单节点内存不足的问题。以包含 1750 亿个参数的 GPT-3 模型为例，如果模型中每一个参数都使用 32 位浮点数表示，那么模型需要占用 700GB 内存。如果使用 16 位浮点数表示，那么每个模型副本需要占用 350GB 内存。2022 年 3 月 NVIDIA 发布的 H100 加速卡仅支持 80GB 显存，无法将整个模型完整放入其中。模型并行可以从计算图角度，用以下两种形式进行切分。

（1）按模型的层切分到不同设备，即**层间并行**或**算子间并行**（Inter-operator Parallelism），也称之为**流水线并行**（Pipeline Parallelism，PP）。

（2）将计算图层内的参数切分到不同设备，即**层内并行**或**算子内并行**（Intra-operator Parallelism），也称之为**张量并行**（Tensor Parallelism，TP）。两节点模型并行训练系统样例如图4.5 所示，图4.5(a) 为流水线并行，模型的不同层被切分到不同的设备中；图4.5(b) 为张量并行，同一层中的不同参数被切分到不同的设备中进行计算。

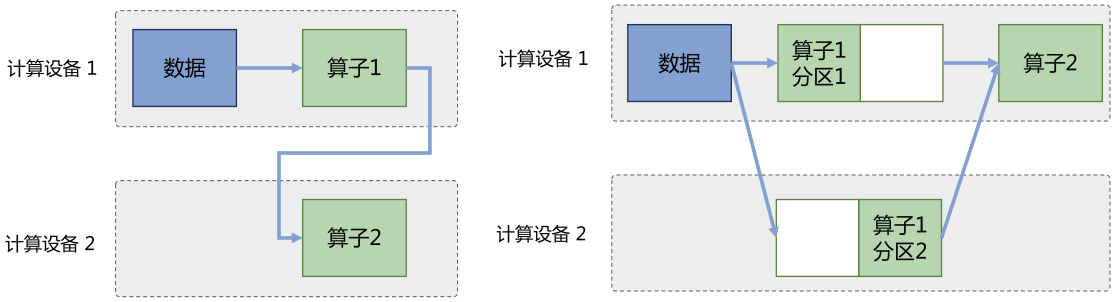


图 4.5　两节点模型并行训练系统样例

1. 流水线并行

流水线并行是一种并行计算策略，将模型的各个层分段处理，并将每个段分布在不同的计算设备上，使得前后阶段能够流水式、分批工作。流水线并行通常应用于大语言模型的并行系统中，以有效解决单个计算设备内存不足的问题。图4.6 给出了一个由四个计算设备组成的流水线并行系统，包含前向计算和后向计算。其中 F_1、F_2、F_3、F_4 分别代表四个前向路径，位于不同的设备上；而 B_4、B_3、B_2、B_1 则代表逆序的后向路径，也分别位于四个不同的设备上。从图 4.6 中可以看出，计算图中

的下游设备（Downstream Device）需要长时间持续处于空闲状态，等待上游设备（Upstream Device）计算完成，才能开始计算自身的任务。这种情况导致设备的平均使用率大幅降低，形成了**模型并行气泡**（Model Parallelism Bubble），也称为**流水线气泡**（Pipeline Bubble）。

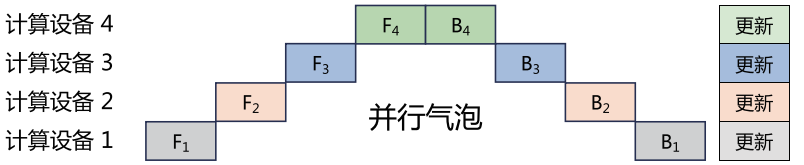


图 4.6　流水线并行样例

朴素流水线策略所产生的并行气泡，使得系统无法充分利用计算资源，降低了系统整体的计算效率。为了减少并行气泡，文献 [170] 提出了 GPipe 方法，将小批次（Mini-batch）进一步划分成更小的**微批次**（Micro-batch），利用流水线并行方法，每次处理一个微批次的数据。在当前阶段计算完成得到结果后，将该微批次的结果发送给下游设备，同时开始处理后一个微批次的数据，这样可以在一定程度上减少并行气泡。图4.7 给出了 GPipe 策略流水线并行样例。前向 F_1 计算被拆解为 F_{11}、F_{12}、F_{13}、F_{14}，在计算设备 1 中计算完成 F_{11} 后，会在计算设备 2 中进行 F_{21} 计算，同时在计算设备 1 中并行计算 F_{12}。相比于最原始的流水线并行方法，GPipe 流水线方法可以有效减少并行气泡。

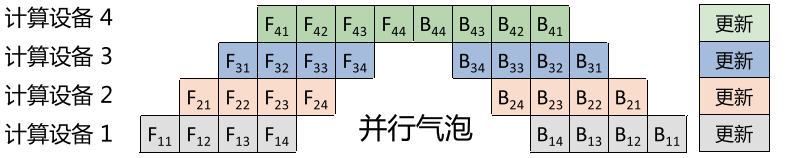


图 4.7　GPipe 策略流水线并行样例[170]

虽然 GPipe 策略可以减少一定的并行气泡，但是只有当一个小批次中所有的前向计算都完成时，才能执行后向计算。因此，还是会产生很多并行气泡，从而降低系统的并行效率。Megatron-LM[171] 采用了 1F1B 流水线并行策略，即一个前向通道和一个后向通道。1F1B 流水线并行策略引入了任务调度机制，使得下游设备能够在等待上游计算的同时执行其他可并行的任务，从而提高设备的利用率。1F1B 给出了非交错式和交错式两种调度模式，如图4.8 所示。

1F1B 非交错式调度模式可分为三个阶段。首先是热身阶段，在计算设备中进行不同数量的前向计算。接下来的阶段是前向-后向阶段，计算设备按顺序执行一次前向计算，然后进行一次后向计算。最后一个阶段是后向阶段，计算设备完成最后一次后向计算。相比于 GPipe 策略，1F1B 非交错式调度模式在节省内存方面表现得更好。然而，它需要与 GPipe 策略一样的时间来完成一轮计算。

1F1B 交错式调度模式要求微批次的数量是流水线阶段的整数倍。每个设备不仅负责连续多个层的计算，还可以处理多个层的子集，这些子集被称为模型块。具体而言，在之前的模式中，设备 1 可能负责层 1~4，设备 2 负责层 5~8，依此类推。在新的模式下，设备 1 可以处理层 1、2、9、10，设备 2 处理层 3、4、11、12，依此类推。在这种模式下，每个设备在流水线中被分配到多个阶段。例如，设备 1 可能参与热身阶段、前向计算阶段和后向计算阶段的某些子集任务。每个设备可以并行执行不同阶段的计算任务，从而更好地利用流水线并行的优势。这种模式不仅在内存消耗方面表现出色，还能提高计算效率，使大型模型的并行系统能够更高效地完成计算任务。

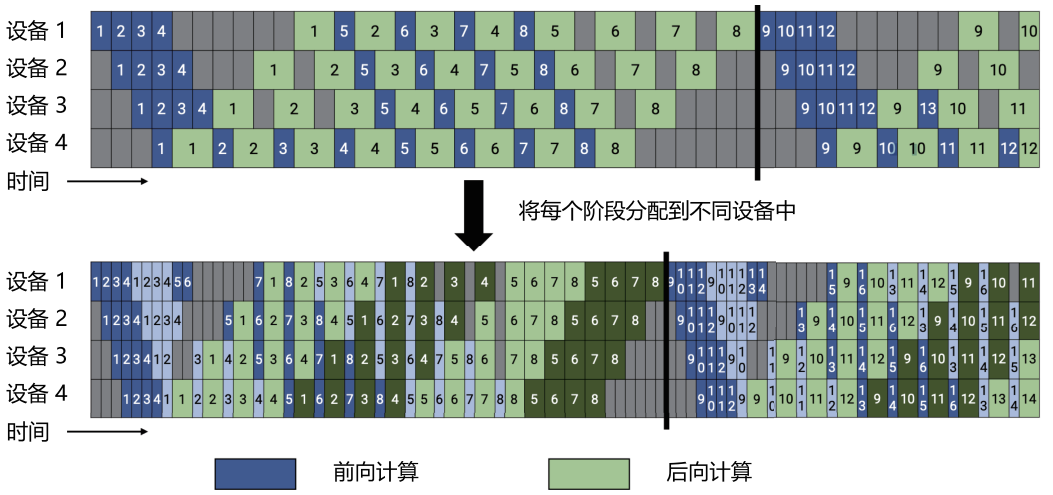


图 4.8　1F1B 流水线并行策略样例[171]

PyTorch 中也包含了实现流水线的 API 函数 Pipe，具体实现参考“torch.distributed.pipeline.sync.Pipe”类。可以使用这个 API 构造一个模型，其包含两个线性层，分别放置在两个计算设备中的样例如下：

```
{
# 步骤 0: 先初始化远程过程调用 (RPC) 框架
os.environ['MASTER_ADDR'] = 'localhost'
os.environ['MASTER_PORT'] = '29500'
torch.distributed.rpc.init_rpc('worker', rank=0, world_size=1)

# 步骤 1: 构建一个模型，包括两个线性层
fc1 = nn.Linear(16, 8).cuda(0)
fc2 = nn.Linear(8, 4).cuda(1)


# 步骤 2: 使用nn.Sequential包装这两个层
model = nn.Sequential(fc1, fc2)


# 步骤 3: 构建流水线 (torch.distributed.pipeline.sync.Pipe)
model = Pipe(model, chunks=8)


# 进行训练/推断
input = torch.rand(16, 16).cuda(0)
output_rref = model(input)
}
```

2. 张量并行

张量并行需要根据模型的具体结构和算子类型，解决如何将参数切分到不同设备，以及如何保证切分后的数学一致性这两个问题。大语言模型都是以 Transformer 结构为基础，Transformer 结构主要由嵌入式表示（Embedding）、矩阵乘（MatMul）和交叉熵损失（Cross Entropy Loss）计算构成。这三种类型的算子有较大的差异，需要设计对应的张量并行策略[169] 才可以实现将参数切分到不同的设备。

对于嵌入式表示算子，如果总的词表数非常大，会导致单计算设备显存无法容纳 Embedding 层参数。举例来说，如果词表数量是 64000，嵌入式表示维度为 5120，类型采用 32 位精度浮点数，那么整层参数需要的显存大约为 $64000 \times 5120 \times 4/1024/1024 = 1250$MB，反向梯度同样需要 1250MB 显存，仅仅存储就需要将近 2.5GB。对于嵌入表示层的参数，可以按照词维度切分，每个计算设备只存储部分词向量，然后通过汇总各个设备上的部分词向量，得到完整的词向量。图4.9 给出了单节点 Embedding 和两节点 Embedding 张量并行的示意图。在单节点上，执行 Embedding 操作，bz 是批次大小（batch size），Embedding 的参数大小为 [word_size, hidden_size]，计算得到 [bz, hidden_size] 张量。图4.9 中 Embedding 张量并行示例将 Embedding 参数沿 word_size 维度切分为两块，每块大小为 [word_size/2, hidden_size]，分别存储在两个设备上。当每个节点查询各自的词表时，如果无法查到，则该词的表示为 0，各设备查询后得到 [bz, hidden_size] 结果张量，最后

通过 AllReduce_Sum 通信[①]，跨设备求和，得到完整的全量结果。可以看出，这里的输出结果和单计算设备执行的结果一致。

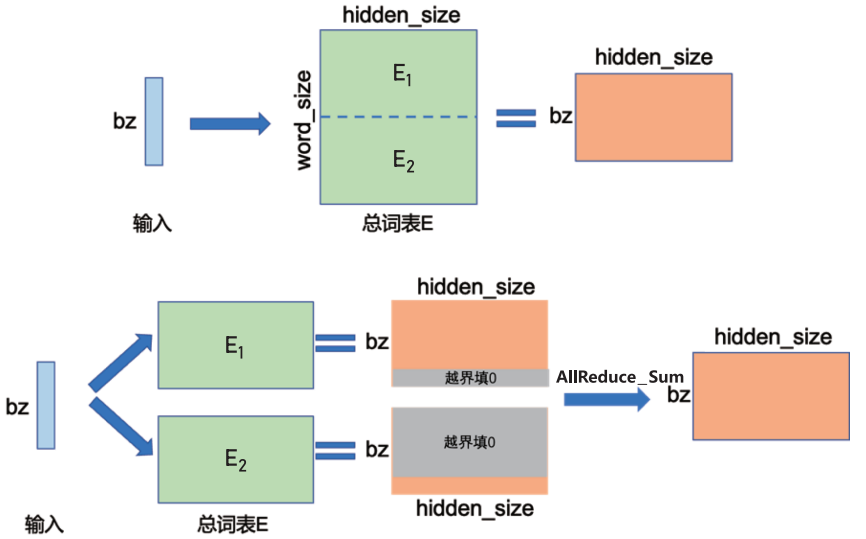


图 4.9　单节点 Embedding 和两节点 Embedding 张量并行的示意图

矩阵乘的张量并行要充分利用矩阵的分块乘法原理。举例来说，要实现如下矩阵乘法 $\boldsymbol{Y} = \boldsymbol{X}\boldsymbol{A}$，其中 $\boldsymbol{X}$ 是维度为 $M \times N$ 的输入矩阵，$\boldsymbol{A}$ 是维度为 $N \times K$ 的参数矩阵，$\boldsymbol{Y}$ 是结果矩阵，维度为 $M \times K$。如果参数矩阵 $\boldsymbol{A}$ 非常大，甚至超出单张卡的显存容量，那么可以把参数矩阵 $\boldsymbol{A}$ 切分到多张卡上，并通过集合通信汇集结果，保证最终结果在数学计算上等价于单计算设备的计算结果。参数矩阵 $\boldsymbol{A}$ 存在以下两种切分方式。

（1）参数矩阵 $\boldsymbol{A}$ 按列切块，将矩阵 $\boldsymbol{A}$ 按列切成

$$\boldsymbol{A} = [\boldsymbol{A}_1, \boldsymbol{A}_2] \tag{4.2}$$

（2）参数矩阵 $\boldsymbol{A}$ 按行切块，将矩阵 $\boldsymbol{A}$ 按行切成

$$\boldsymbol{A} = \begin{vmatrix} \boldsymbol{A}_1 \\ \boldsymbol{A}_2 \end{vmatrix} \tag{4.3}$$

图4.10 给出了参数矩阵按列切分的示例，参数矩阵 $\boldsymbol{A}$ 分别将 $\boldsymbol{A}_1, \boldsymbol{A}_2$ 放置在两个计算设备上。两个计算设备分别计算 $\boldsymbol{Y}_1 = \boldsymbol{X}\boldsymbol{A}_1$ 和 $\boldsymbol{Y}_2 = \boldsymbol{X}\boldsymbol{A}_2$。计算完成后，多计算设备间进行通信，从而获

① 在 4.3.3 节进行介绍。

取其他计算设备上的计算结果，并拼接在一起得到最终的结果矩阵 $\boldsymbol{Y}$，该结果在数学上与单计算设备在计算结果上完全等价。

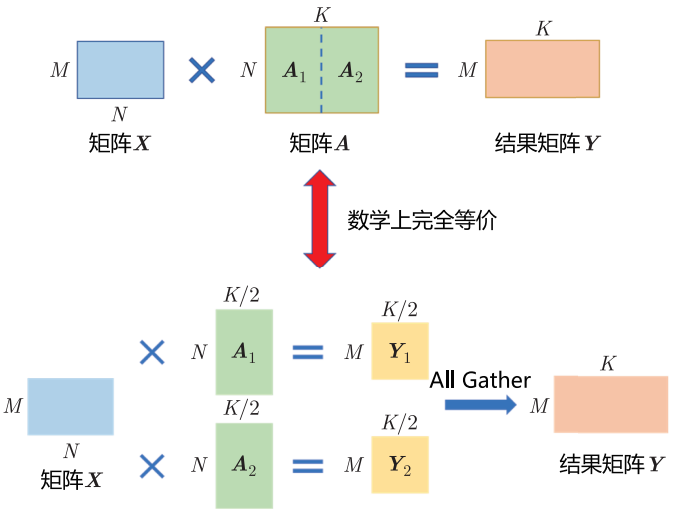


图 4.10 参数矩阵按列切分的示例

图4.11 给出了参数矩阵按行切分的示例，为了满足矩阵乘法规则，输入矩阵 $\boldsymbol{X}$ 需要按列切分 $\boldsymbol{X} = [\boldsymbol{X}_1|\boldsymbol{X}_2]$。同时，将矩阵分块，分别放置在两个计算设备上，每个计算设备分别计算 $\boldsymbol{Y}_1 = \boldsymbol{X}_1\boldsymbol{A}_1$ 和 $\boldsymbol{Y}_2 = \boldsymbol{X}_2\boldsymbol{A}_2$。计算完成后，多个计算设备间通信获取其他卡上的计算结果，可以得到最终的结果矩阵 $\boldsymbol{Y}$。同样，这种切分方式，既可以保证数学上的计算等价性，解决单计算设备显存无法容纳的问题，又可以保证单计算设备通过拆分的方式装下参数 $\boldsymbol{A}$。

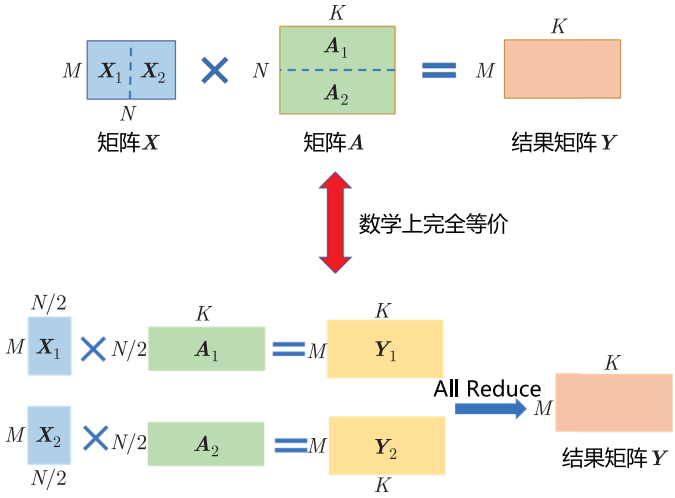


图 4.11　参数矩阵按行切分的示例

Transformer 中的 FFN 结构均包含两层全连接（Fully Connected，FC）层，即存在两个矩阵乘，这两个矩阵乘分别采用上述两种切分方式，如图4.12 所示。对第一个 FC 层的参数矩阵按列切块，对第二个 FC 层的参数矩阵按行切块。这样，第一个 FC 层的输出恰好满足第二个 FC 层的数据输入要求（按列切分），因此可以省去第一个 FC 层后的汇总通信操作。多头自注意力机制的张量并行与 FFN 类似，因为具有多个独立的头，所以相较于 FFN 更容易实现并行，其矩阵切分方式如图4.13 所示。具体可以参考文献 [169]。

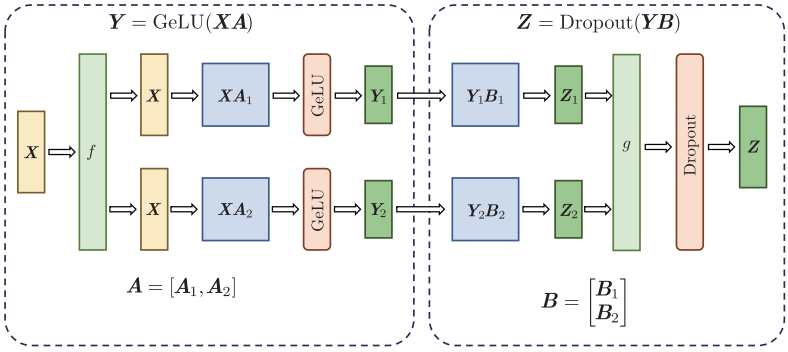


图 4.12　FNN 结构的张量并行示意图[169]

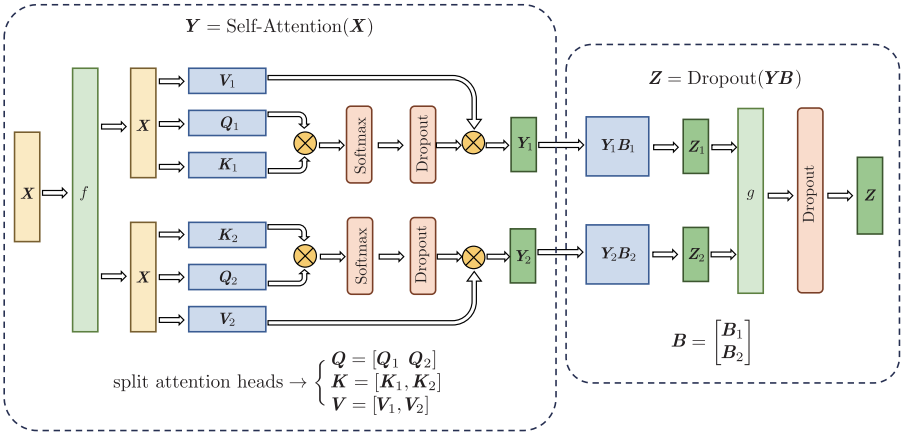


图 4.13 多头自注意力机制的张量并行示意图[169]

分类网络最后一层一般会选用 Softmax 和 Cross_entropy 算子来计算交叉熵损失。如果类别数量非常大，则会导致单计算设备内存无法存储和计算 logit 矩阵。针对这一类算子，可以按照类别维度切分，同时通过中间结果通信，得到最终的全局交叉熵损失。首先计算的是 Softmax 值，公式如下：

$$\text{Softmax}(x_i) = \frac{\mathrm{e}^{x_i}}{\sum\limits_j \mathrm{e}^{x_j}} = \frac{\mathrm{e}^{x_i - x_{\max}}}{\sum\limits_j \mathrm{e}^{x_j - x_{\max}}} = \frac{\mathrm{e}^{x_i - x_{\max}}}{\sum\limits_N \sum\limits_j \mathrm{e}^{x_j - x_{\max}}} \tag{4.4}$$

$$x_{\max} = \max_p(\max_k(x_k)) \tag{4.5}$$

其中，p 表示张量并行的设备号。得到 Softmax 计算结果之后，同时对标签 Target 按类别切分，每个设备得到部分损失，最后进行一次通信，得到所有类别的损失。整个过程，只需要进行三次小量的通信，就可以完成交叉熵损失的计算。

PyTorch 提供了细粒度张量级别的并行 API——DistributedTensor。也提供了粗粒度模型层面的 API 对“nn.Module”进行张量并行。通过以下几行代码就可以实现对一个大的张量进行分片：

```
import torch
from torch.distributed._tensor import DTensor, DeviceMesh, Shard, distribute_tensor

# 使用可用设备构建设备网格（多主机或单主机）
device_mesh = DeviceMesh("cuda", [0, 1, 2, 3])
# 如果想要进行逐行分片
rowwise_placement=[Shard(0)]
# 如果想要进行逐列分片
colwise_placement=[Shard(1)]

big_tensor = torch.randn(888, 12)
# 分布式张量返回将根据指定的放置维度进行分片
rowwise_tensor = distribute_tensor(big_tensor, device_mesh=device_mesh,
                                   placements=rowwise_placement)
```

对于像“nn.Linear”这样已经有“torch.Tensor”作为参数的模块，也提供了模块级 API “distribute_module”在模型层面进行张量并行，参考代码如下：

```
import torch
from torch.distributed._tensor import DeviceMesh, Shard, distribute_tensor, distribute_module

class MyModule(nn.Module):
    def __init__(self):
        super().__init__()
        self.fc1 = nn.Linear(8, 8)
        self.fc2 = nn.Linear(8, 8)
        self.relu = nn.ReLU()

    def forward(self, input):
        return self.relu(self.fc1(input) + self.fc2(input))

mesh = DeviceMesh(device_type="cuda", mesh=[[0, 1], [2, 3]])

def shard_params(mod_name, mod, mesh):
    rowwise_placement = [Shard(0),Replicate()]
    def to_dist_tensor(t): return distribute_tensor(t, mesh, rowwise_placement)
    mod._apply(to_dist_tensor)

sharded_module = distribute_module(MyModule(), mesh, partition_fn=shard_params)

def shard_fc(mod_name, mod, mesh):
    rowwise_placement = [Shard(0),Replicate()]
    if mod_name == "fc1":
        mod.weight = torch.nn.Parameter(distribute_tensor(mod.weight, mesh, rowwise_placement))

# 在使用时与前面 shard_params 两者间仅可以选择其一
sharded_module = distribute_module(MyModule(), mesh, partition_fn=shard_fc)
```

4.2.3 混合并行

混合并行将多种并行策略如数据并行、流水线并行和张量并行等混合使用。通过结合不同的并行策略，混合并行可以充分发挥各种并行策略的优点，最大限度地提高计算性能和效率。针对千亿规模的大语言模型，通常，在每个服务器内部使用张量并行策略，由于该策略涉及的网络通信量较大，因此需要利用服务器内部的不同计算设备之间的高速通信带宽。通过流水线并行，将模型的不同层划分为多个阶段，每个阶段由不同的机器负责计算。这样可以充分利用多台机器的计算能力，并通过机器之间的高速通信传递计算结果和中间数据，以提高整体的计算速度和效率。最后，在外层叠加数据并行策略，以增加并发数量，加快整体训练速度。通过数据并行，将训练数据分发到多组服务器上进行并行处理，每组服务器处理不同的数据批次。这样可以充分利用多台服务器的计算资源，并增加训练的并发度，从而加快整体训练速度。

BLOOM 使用 Megatron-DeepSpeed[134] 框架进行训练，主要包含两个部分：Megatron-LM 提

供张量并行能力和数据加载原语；DeepSpeed[172] 提供 ZeRO 优化器、模型流水线及常规的分布式训练组件。通过这种方式可以实现数据、张量和流水线三维并行，BLOOM 模型训练时采用的并行计算结构如图4.14 所示。BLOOM 模型训练使用由 48 个 NVIDIA DGX-A100 服务器组成的集群，每个 DGX-A100 服务器包含 8 块 NVIDIA A100 80GB GPU，总计包含 384 块。BLOOM 训练采用的策略是先将集群分为 48 个一组，进行数据并行。接下来，模型整体被分为 12 个阶段，进行流水线并行。每个阶段的模型被划分到 4 块 GPU 中，进行张量并行。同时，BLOOM

使用了 ZeRO（零冗余优化器）[173] 进一步降低模型对显存的占用。通过上述步骤可以实现数百个 GPU 的高效并行计算。

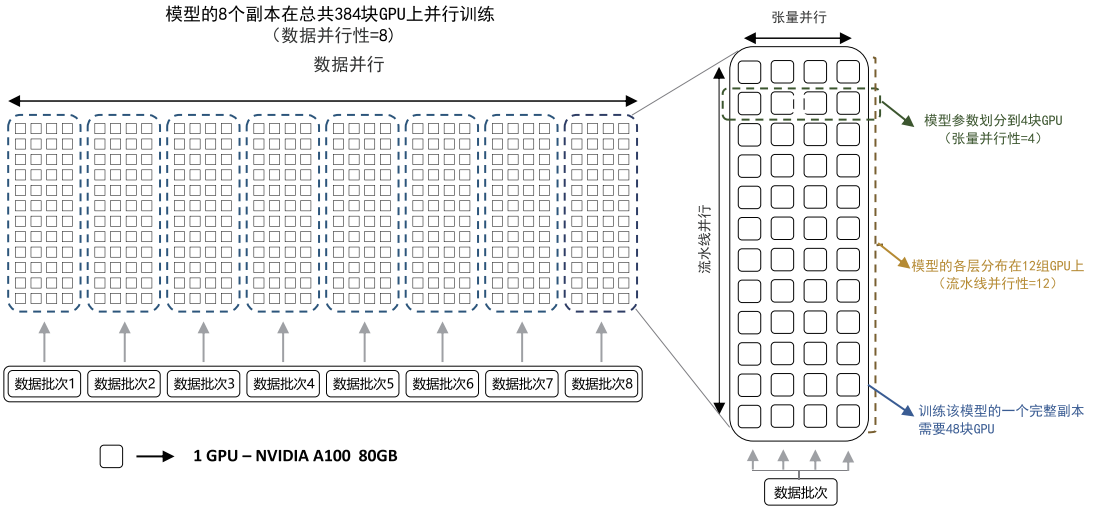


图 4.14 BLOOM 模型训练时采用的并行计算结构[31]

4.2.4 计算设备内存优化

当前，大语言模型训练通常采用 Adam 优化算法，除了需要每个参数梯度，还需要一阶动量（Momentum）和二阶动量（Variance）。虽然 Adam 优化算法相较 SGD 算法效果更好也更稳定，但是对计算设备内存的占用显著增大。为了降低内存占用，大多数系统采用混合精度训练（Mixed Precision Training）方式，即同时存在 **FP32**（32 位浮点数）与**FP16**（16 位浮点数）或者 **BF16**（BFloat16）格式的数值。FP32、FP16 和 BF16 的表示如图4.15 所示。FP32 中第 31 位为符号位，第 30 位～第 23 位用于表示指数，第 22 位～第 0 位用于表示尾数。FP16 中第 15 位为符号位，第 14 位～第 10 位用于表示指数，第 9 位～第 0 位用于表示尾数。BF16 中第 15 位为符号位，第 14 位～第 7 位用于表示指数，第 6 位～第 0 位用于表示尾数。由于 FP16 的值区间比 FP32 的值区间小很多，所以在计算过程中很容易出现上溢出和下溢出。BF16 相较于 FP16 以精度换取更大的值区间范围。由于 FP16 和 BF16 相较 FP32 精度低，训练过程中可能会出现梯度消失和模型不稳定的问题，因此，需要使用一些技术解决这些问题，例如**动态损失缩放**（Dynamic Loss Scaling）和**混合精度优化器**（Mixed Precision Optimizer）等。

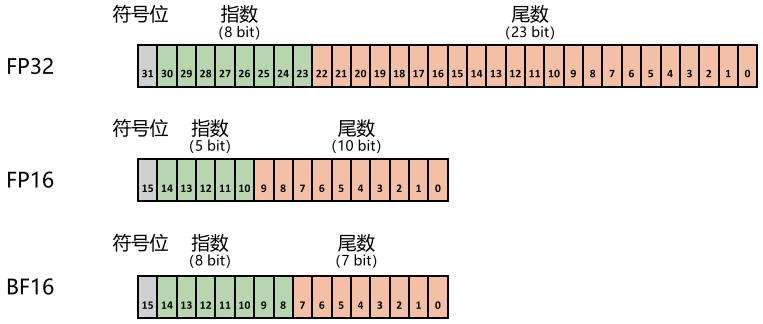


图 4.15　FP32、FP16 和 BF16 的表示

混合精度优化的过程如图4.16 所示。Adam 优化器状态包括采用 FP32 保存的模型参数备份，一阶动量和二阶动量也都采用 FP32 格式存储。假设模型参数量为 Φ，模型参数和梯度都是用 FP16 格式存储，则共需要 $2\Phi + 2\Phi + (4\Phi + 4\Phi + 4\Phi) = 16\Phi$ 字节存储。其中，Adam 状态占比 75%。动态损失缩放在反向传播前，将损失变化（dLoss）手动增大 2^K 倍，因此反向传播时得到的激活函数梯度不会溢出；反向传播后，将权重梯度缩小 2^K 倍，恢复正常值。举例来说，有 75 亿个参数的模型，如果用 FP16 格式，只需要 15GB 计算设备内存，但是在训练阶段，模型状态实际上需要耗费 120GB 内存。计算卡内存占用中除了模型状态，还有剩余状态（Residual States），包括激活值（Activation）、各种临时缓冲区（Buffer）及无法使用的显存碎片（Fragmentation）等。可以使用激活值检查点（Activation Checkpointing）方式使激活值内存占用大幅减少，因此如何减少模型状态尤其是 Adam 优化器状态是解决内存占用问题的关键。

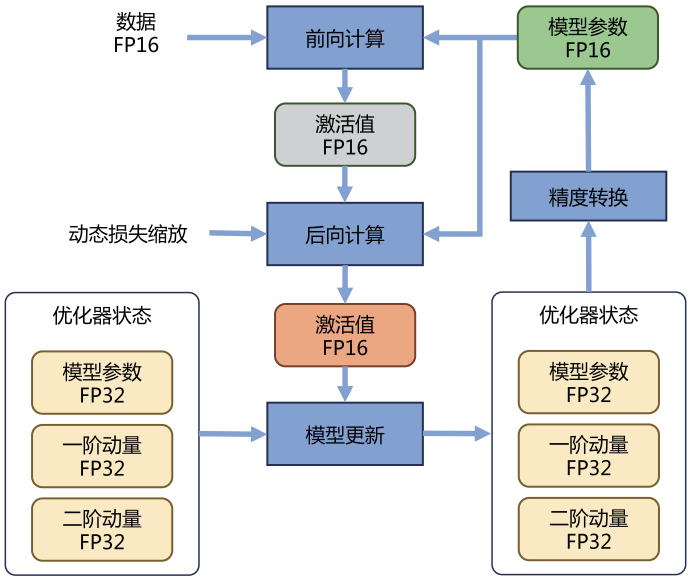


图 4.16　混合精度优化的过程