# 第一章　分布式通信机制详解

参考链接：算能 SCCL

## 第1条　同步原语

1) **Barrier/Fence（栅障同步）**

**定义：**所有节点的同步点

**语义：**阻塞所有调用者，直到所有组内成员都到达该点

**应用场景：**

a) 确保所有节点完成某阶段计算后再进入下一阶段

b) 性能测试中的时间同步点

2) **Signal & Wait（信号等待）**

**定义：**节点间的事件同步机制

**语义：**

a) Signal：发送确认信号

b) Wait：等待确认信号

**应用场景：**

c) 节点间数据同步确认

d) 异步通信的完成检测

## 第2条　通信原语（Communication Primitives）

通信原语是分布式训练中最基础的抽象通信操作，定义了**做什么**（What）。在大模型训练中，不同的并行策略会使用不同的通信原语来实现节点间的数据同步。

通信原语是一组标准化的集合通信操作模板，由最基础的操作（发送 send、接收 receive、复制 copy、同步 barrier）组合而成。每个原语定义了数据在多个节点间的传输模式和规约方式。

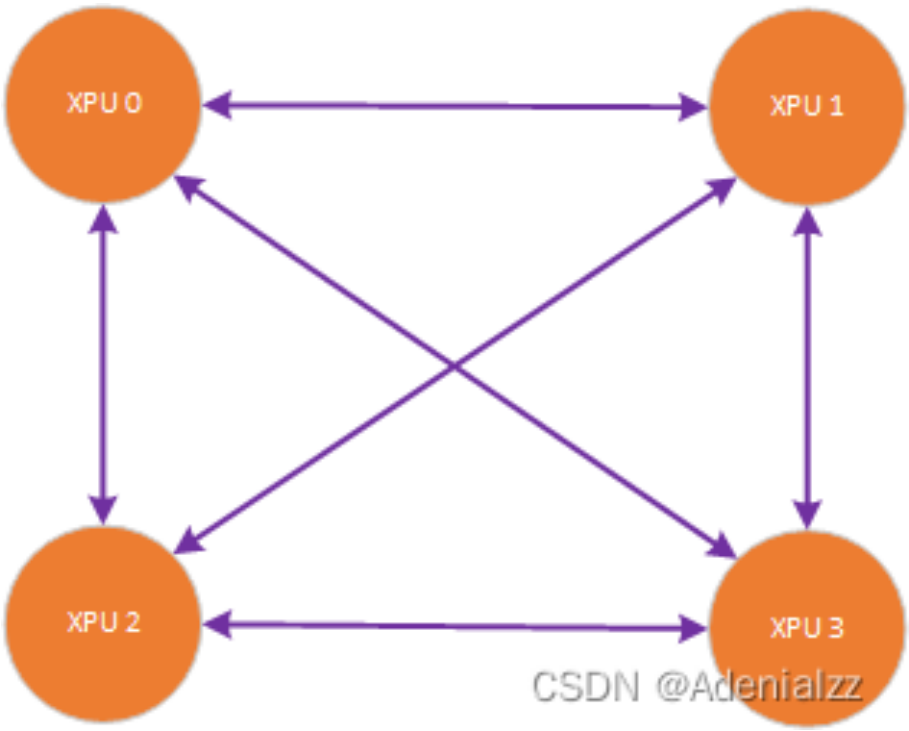



图 4.1 4 个节点的全连接拓扑示意图

1) Broadcast（广播）

**定义**：1对多通信，一个发送者，多个接收者

**语义**：将一个节点的数据复制并发送到所有其他节点

**特点**：

a) 所有节点接收到**相同**的数据

b) 主节点（通常是节点0）作为数据源

**应用场景**：

c) 数据并行的参数初始化，确保每张卡初始参数一致

d) AllReduce的Reduce + Broadcast组合中的Broadcast操作

e) Parameter Server架构中master节点向worker节点广播参数

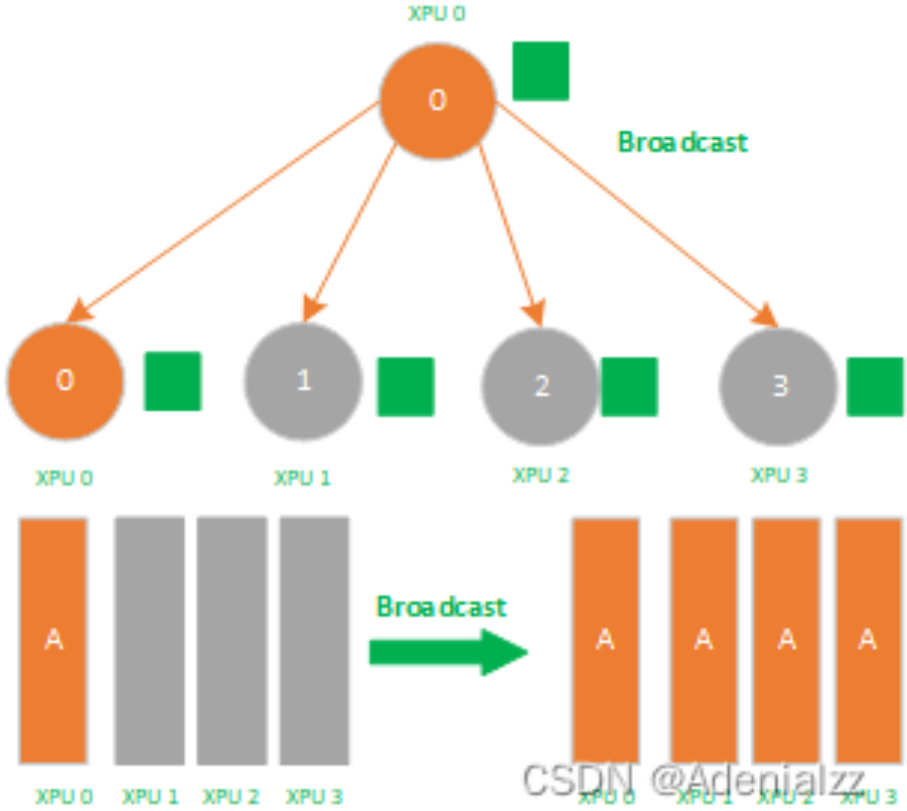



图 4.2 Broadcast 示意图

2) Scatter（分发）

**定义**：1对多通信，数据分片分发

**语义**：将一个节点的数据切分成N份，分发到N个节点（包括自己）

**特点**：

a) 每个节点接收**不同**的数据片段

b) 是Gather的反向操作

**应用场景**：

c) ReduceScatter组合中的Scatter操作

d) 模型并行初始化时将模型scatter到不同节点

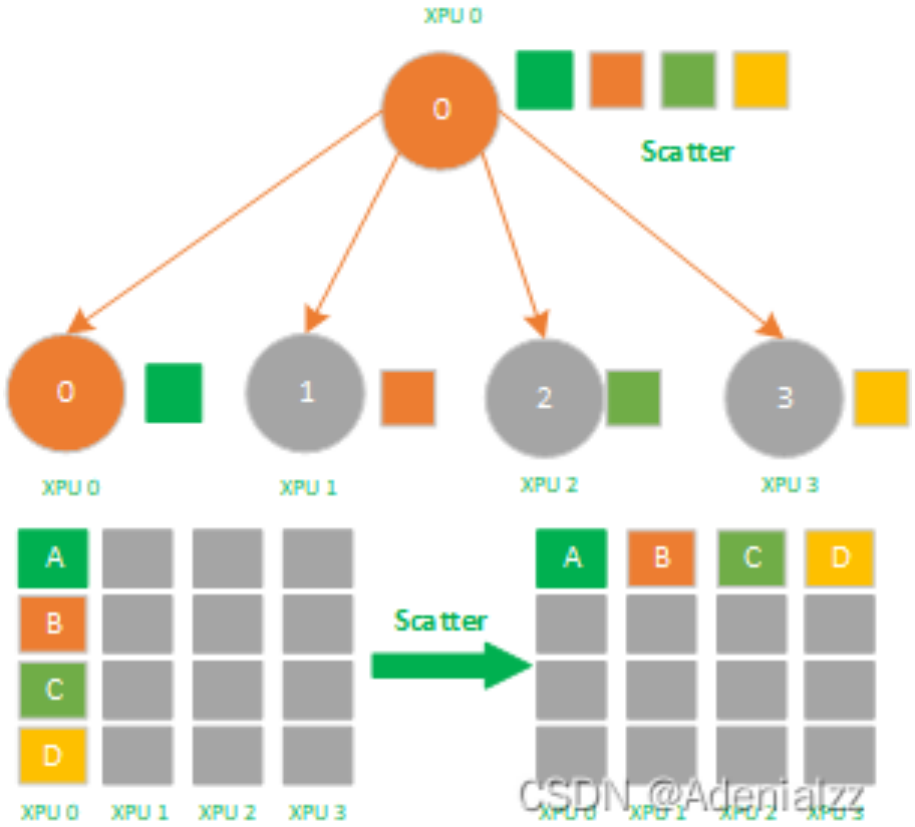



图 4.3 Scatter 示意图

3） Gather（收集）

**定义：**多对1通信，多个发送者，一个接收者

**语义：**将多个节点的数据收集到一个主节点

**特点：**

a） 主节点收集所有数据（按节点顺序拼接）

b） 是Scatter的反向操作

**应用场景：**

c） 模型并行中收集各节点的部分结果

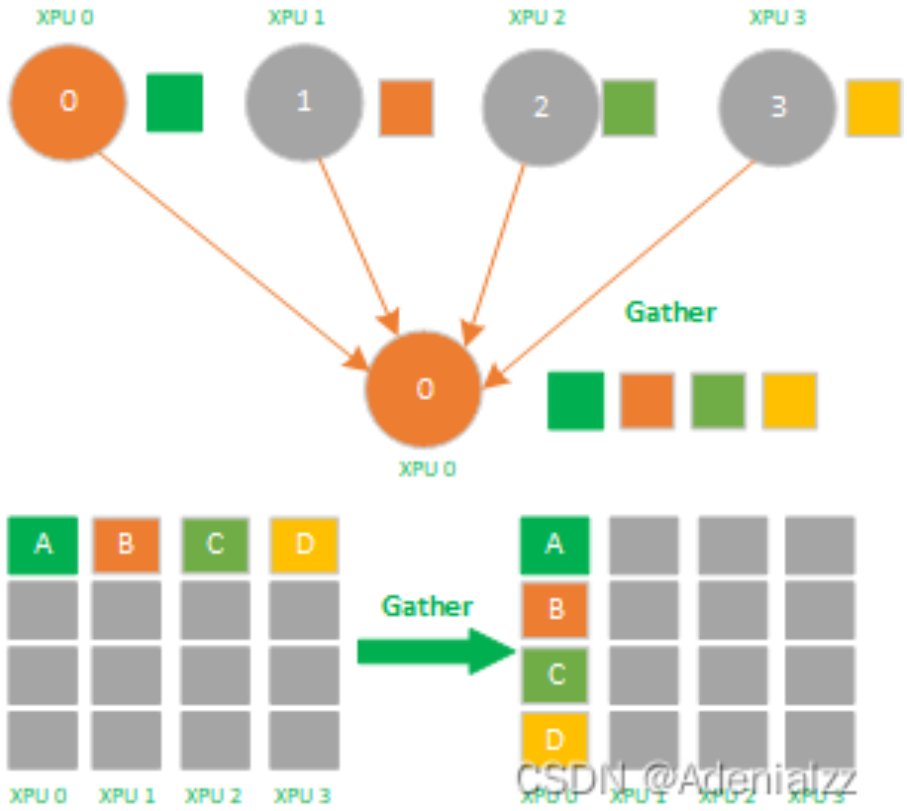



图 4.4 Gather 示意图

4） AllGather（全收集）

**定义**：多对多通信，所有节点互相收集数据

**语义**：每个节点的数据被收集到所有节点 = Gather + Broadcast

**特点**：

a） 所有节点最终拥有相同的完整数据集

b） 是ReduceScatter的反向操作

c） 没有规约计算，只有数据传输

**通信量分析**：

d） 每个节点发送：(N-1) × datasize

e） 每个节点接收：(N-1) × datasize

f） 每个节点总通信量：2 × (N-1) × datasize

**应用场景**：

g） 模型并行前向计算时的参数全同步

h） 需要将切分到不同节点的参数同步到每张卡

i） ZeRO优化中的参数收集阶段

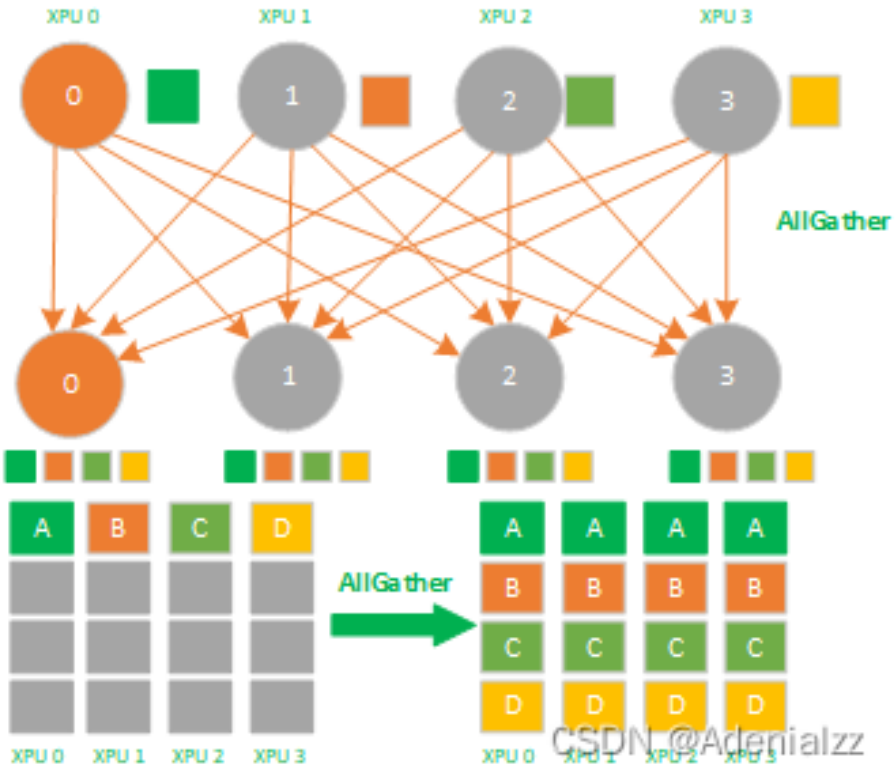



图 4.5 AllGather 示意图

5） Reduce（规约）

**定义**：多对1通信，多个发送者，一个接收者，包含规约运算

**语义**：将多个节点的数据通过规约运算（如SUM、MAX）聚合到主节点

**规约操作符**：

a） 数值运算：SUM（求和）、PROD（求积）、MAX（最大值）、MIN（最小值）

b） 逻辑运算：LAND（逻辑与）、LOR（逻辑或）、LXOR（逻辑异或）

c） 位运算：BAND（按位与）、BOR（按位或）、BXOR（按位异或）

d） 位置运算：MAXLOC（最大值位置）、MINLOC（最小值位置）

**应用场景**：

e） AllReduce的Reduce + Broadcast组合中的Reduce操作

f） 梯度聚合的第一步

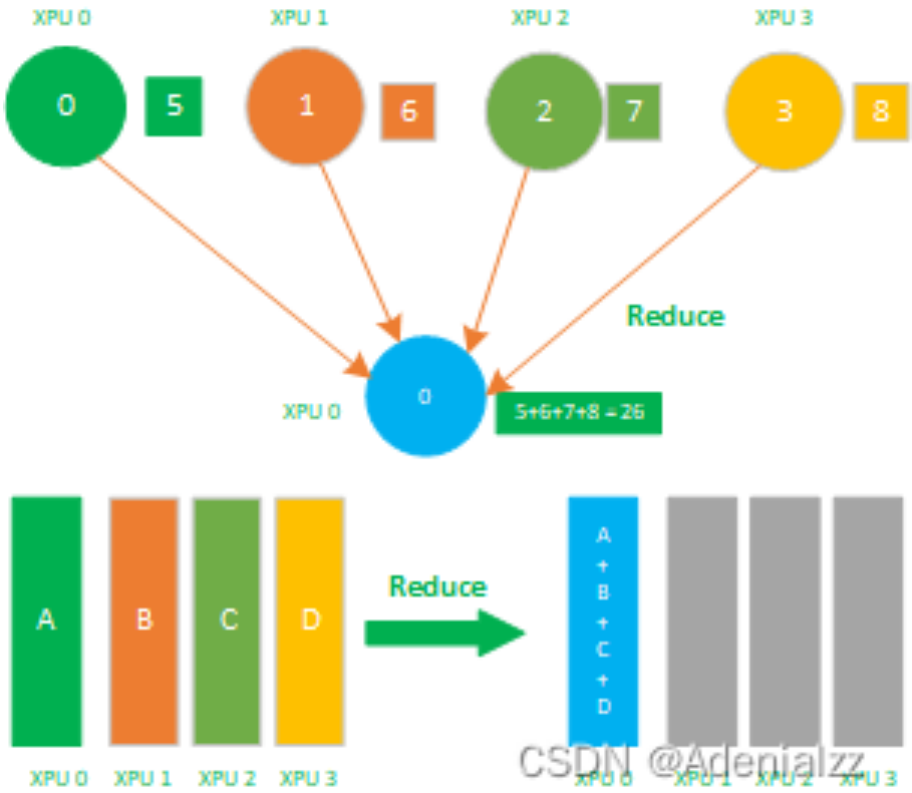



图 4.6 Reduce 示意图

6） ReduceScatter（规约分发）

**定义**：多对多通信，规约后分发

**语义**：对所有节点的数据进行Reduce规约，然后将结果Scatter分发到各节点 = Reduce + Scatter

**特点**：

a） 是AllGather的反向操作

b） 每个节点最终得到不同的规约结果片段

**通信量分析**：

c） 每个节点的通信量：2 × (N-1) × datasize / N

d） 总通信量比AllReduce少，因为结果是分散的

**应用场景**：

e） 数据并行AllReduce的第一阶段（ReduceScatter + AllGather）

f） 模型并行反向传播中的梯度分散

g） ZeRO优化中的梯度规约和分散

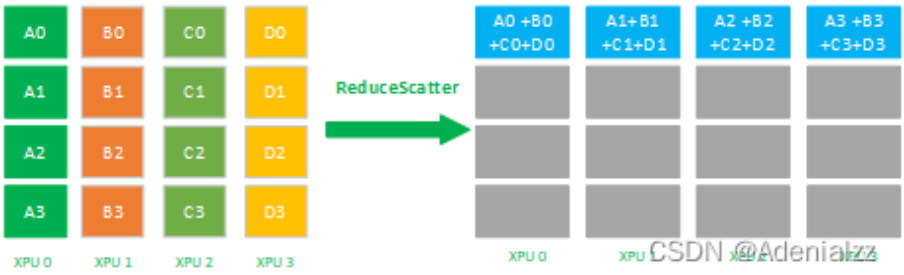



图 4.7 ReduceScatter 示意图

7） AllReduce（全规约）

**定义：**多对多通信，所有节点执行规约并获得相同结果

**语义：**对所有节点的数据进行Reduce规约，并将结果发送到所有节点

**等价实现：**

a） 方式1：Reduce + Broadcast

b） 方式2：ReduceScatter + AllGather

**特点：**

c） 所有节点最终拥有相同的规约结果

d） 是数据并行中最常用的通信原语

e） 不同拓扑和算法实现性能差异显著

**应用场景：**

f） 数据并行的梯度同步（最主要应用）

g） 张量并行中的矩阵乘法结果同步

h） 全局统计信息（如BatchNorm）的计算

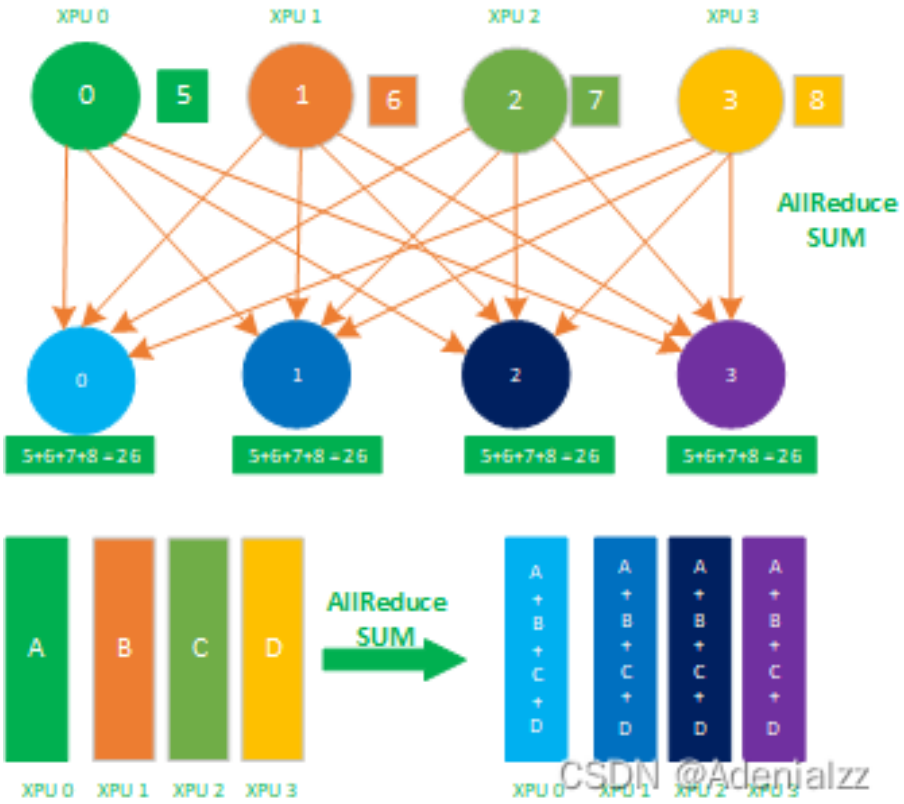



图 4.8 AllReduce 示意图

8) All-to-All（全交换）

**定义**：多对多通信，全局数据转置

**语义**：每个节点将数据切分成N份，第i份发送给节点i；同时接收来自所有节点的第j份数据（j为当前节点编号）

**特点**：

a) 是AllGather的扩展

b) AllGather中所有节点接收相同数据，All-to-All中每个节点接收不同数据

c) 实现全局数据重排（转置）

**通信量分析**：

d) 每个节点发送：(N-1) × datasize / N（发送给其他N-1个节点）

e) 每个节点接收：(N-1) × datasize / N

f) 每个节点总通信量：2 × (N-1) × datasize / N

**应用场景**：

g) 专家并行（MoE）的token分发和收集（Dispatch/Combine）

h) 模型并行中的矩阵转置

i) 数据并行到模型并行的转换

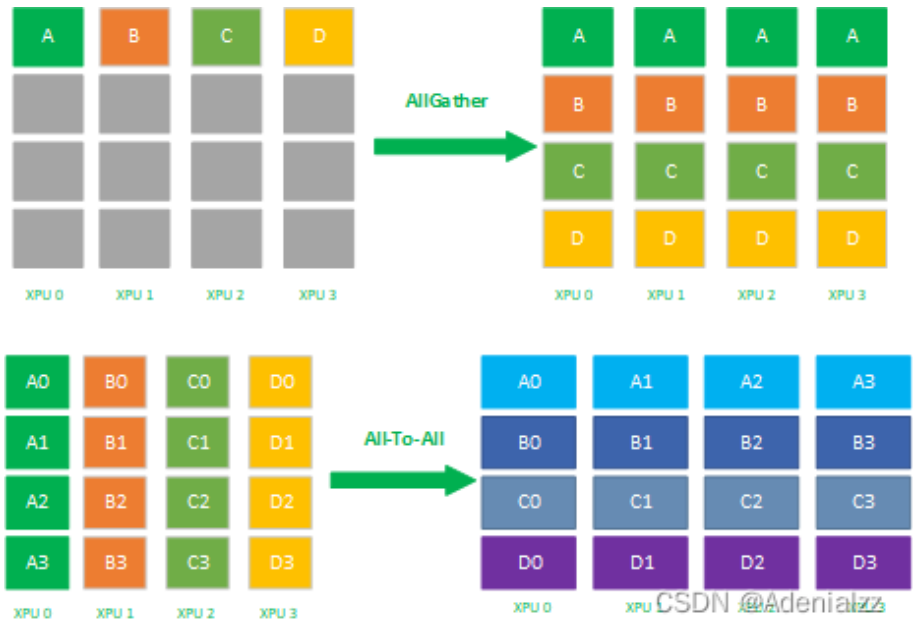



图 4.9 All-to-All 示意图

9) Point-to-Point（点对点通信）

**定义**：两个节点间的直接数据传输

**语义**：

a) Send：一个节点发送数据到指定节点

b) Receive：一个节点接收来自指定节点的数据

**应用场景**：

c) 流水并行中相邻stage之间的激活传递

d) 环形拓扑中的相邻节点通信

10) 总结

| 通信原语 | 通信模式 | 数据特征 | 是否规约 | 反向操作 | 主要应用 |
|---|---|---|---|---|---|
| **Broadcast** | 1 对多 | 所有节点数据相同 | 否 | - | 参数初始化 |
| **Scatter** | 1 对多 | 每个节点数据不同 | 否 | Gather | 数据分发 |
| **Gather** | 多对 1 | 主节点收集所有数据 | 否 | Scatter | 结果收集 |
| **AllGather** | 多对多 | 所有节点数据相同 | 否 | ReduceScatter | 参数同步 |
| **Reduce** | 多对 1 | 主节点获得规约结果 | 是 | - | 梯度聚合 |
| **ReduceScatter** | 多对多 | 每个节点数据不同 | 是 | AllGather | 梯度分散 |
| **AllReduce** | 多对多 | 所有节点数据相同 | 是 | - | 梯度同步 |
| **All-to-All** | 多对多 | 全局数据转置 | 否 | All-to-All | MoE 通信 |

## 第3条　网络拓扑介绍

网络拓扑定义了节点间的物理/逻辑连接方式，决定了**在哪里**（Where）进行通信。不同的拓扑结构会影响通信算法的选择和性能。

1) Ring（环形拓扑）

**结构特点：**

a) 每个节点连接到两个邻居（前驱和后继）

b) 形成一个封闭的环

c) 节点度数：2（每个节点有2条链路）

**优点：**

d) 实现简单，对网络拓扑要求低

e) 充分利用每个节点的上下行带宽

f) 通信量最优：每个节点通信量为 2 × (N-1) × datasize / N

g) 适合带宽受限场景

**缺点：**

h) 延迟随节点数线性增加：O(N)

i) 对于小数据包，延迟累加明显

j) 不利于大规模集群扩展

**适用场景：**

k) 中小规模集群（N < 32）

l) 带宽受限环境

m) 需要高带宽利用率的场景

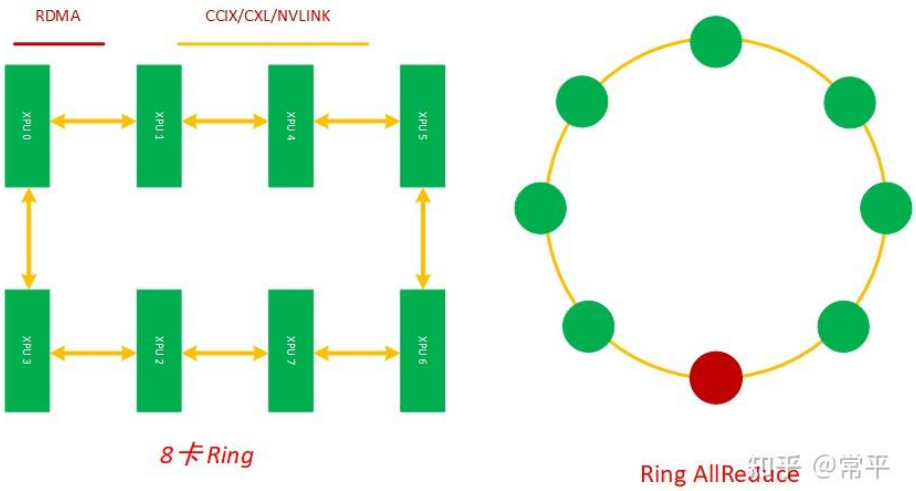



图 4.10 Ring 拓扑图

2) Full-Mesh（全互联拓扑）

**结构特点：**

a) 每个节点与所有其他节点直接相连

b) 节点度数：N-1（每个节点有N-1条链路）

c) 总链路数：N×(N-1)/2

**优点：**

d) 任意两节点间延迟最低：O(1)

e) 支持多种高效算法（Tree、Butterfly等）

f) 灵活性高，可根据数据量选择不同算法

**缺点：**

g) 硬件成本高（需要大量物理链路或高速交换机）

h) 如果使用直接广播算法，通信量大：2 × (N-1) × datasize

**适用场景：**

i) 小规模高性能集群

j) 单节点内多GPU互联（NVLink）

k) 数据中心内机架级互联

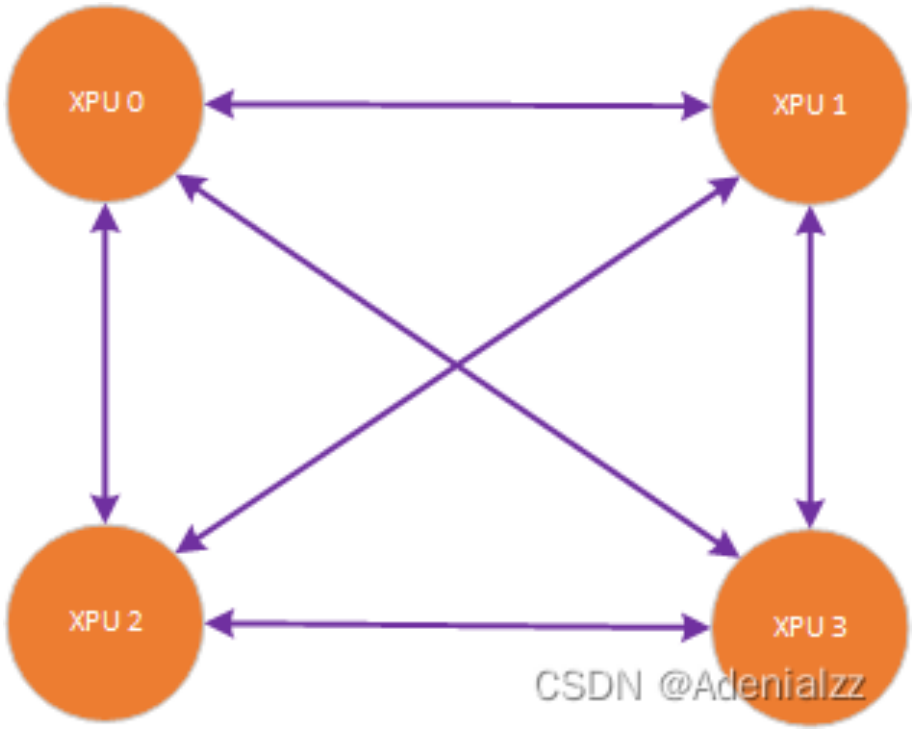



图 4.11 Full-Mesh 拓扑图

3) Binary Tree（二叉树形拓扑）

**结构特点：**

a) 节点按层次组织成树结构

b) 根节点、中间节点、叶节点

c) 二叉树：每个节点最多2个子节点

d) 节点度数：1-3（根节点0个父节点，叶节点0个子节点）

**优点：**

e) 通信量最优：每个节点通信量为 2 × datasize

f) 延迟适中：$O(\log_2 N)$

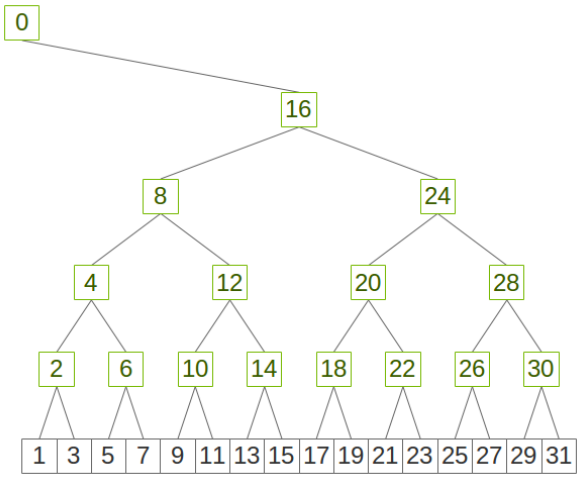
g) 在通信量和延迟间取得平衡

**缺点：**

h) 根节点和中间节点成为瓶颈

i) 单树实现带宽利用率低（只用了单向带宽）

j) **改进方案：**Double Binary Tree（双二叉树）




4) Double Binary Tree（双二叉树）：构建两棵互补的树，充分利用双向带宽

**适用场景：**

a) 中等规模集群

b) Fat-Tree数据中心网络

c) 需要平衡通信量和延迟的场景

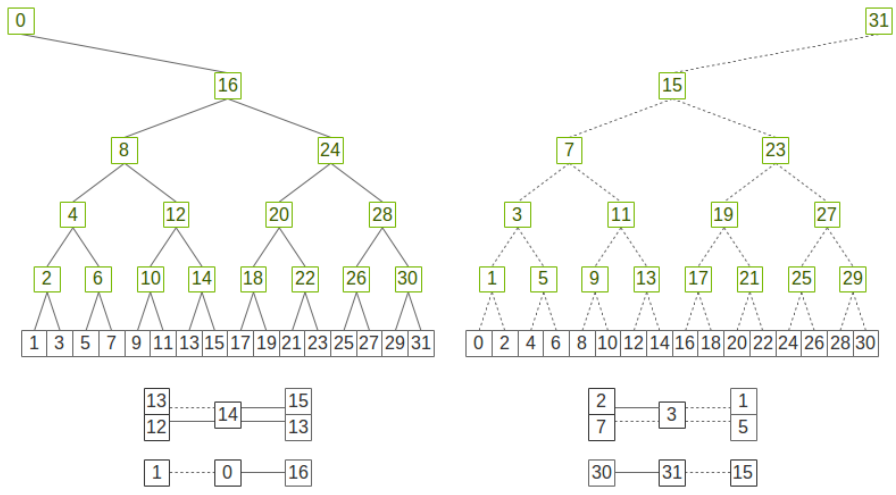



图 4.12 Double Binary Tree 拓扑图

5) 2D Torus（二维环面拓扑）

**结构特点：**

a) 节点排列成二维网格

b) 每个维度形成环形连接

c) 节点度数：4（每个节点连接上下左右4个邻居）

**优点：**

d) 结合多维度并行通信

e) 可分解为多个低维度Ring操作

f) 适合规则的节点布局

**缺点：**

g) 需要规则的节点数（如n×m）

h) 实现复杂度较高

**适用场景：**

i) 超算集群（如Google TPU Pod）

j) 大规模规则布局的GPU集群

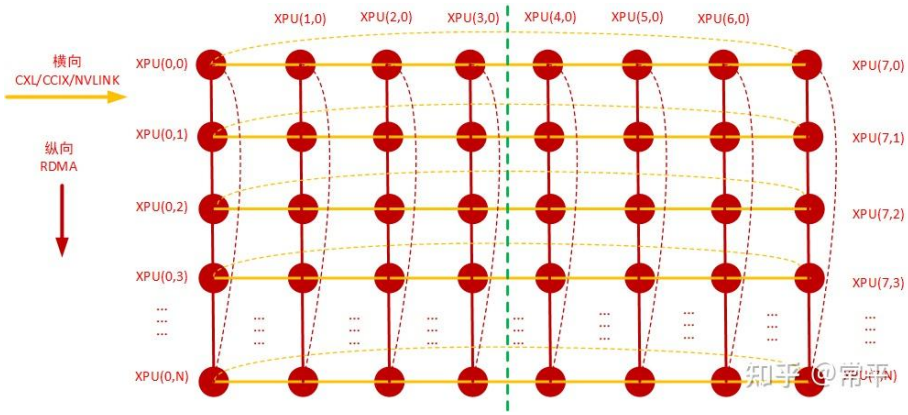



图 4.13 2D Torus 拓扑图

6) Fat-Tree（胖树拓扑）

**结构特点：**

a) 树形结构，但上层链路带宽更高（"更胖"）

b) 或者上层交换机端口数更多

c) 保证任意叶节点间有足够带宽

**优点：**

d) 解决传统树的根节点瓶颈问题

e) 适合数据中心网络

f) 支持高效的Halving-Doubling算法

**缺点：**

g) 硬件成本高

h) 交换机配置复杂

**适用场景：**

i） 大型数据中心

j） 多机多卡训练环境

k） 需要高带宽聚合的场景

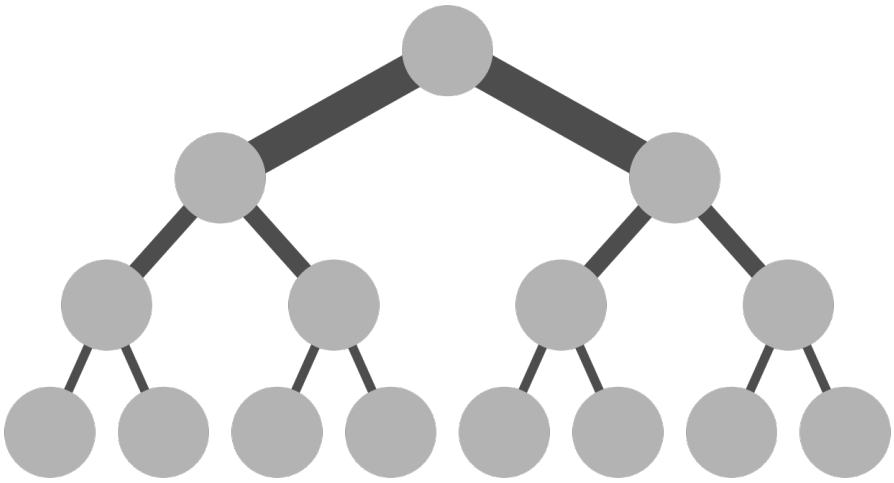


图 4.14 Fat-Tree 拓扑图

**第4条　互联技术与带宽参数**

1） 节点内互联（Intra-node）

a） **NVLink**

带宽：300-600 GB/s（NVLink 3.0/4.0）

延迟：< 1 μs

用途：同一节点内GPU直连

b） **PCIe**

带宽：16-64 GB/s（PCIe Gen4/Gen5）

延迟：1-2 μs

用途：GPU-CPU互联，同节点GPU互联

c） **自研互联**

带宽：根据芯片设计（如 64 GB/s）

延迟：< 1 μs

用途：自研芯片间直连

2） 节点间互联（Inter-node）

a） **InfiniBand**

带宽：200-400 Gbps（HDR/NDR）

延迟：1-2 μs

用途：HPC集群标准互联

b) **RoCE（RDMA over Converged Ethernet）**

带宽：100-400 Gbps

延迟：2-5 μs

用途：数据中心以太网RDMA

c) **以太网**

带宽：10-100 Gbps

延迟：10-100 μs

用途：通用网络互联

**第5条　关键参数定义**

1) **TP Group内互联带宽（intra_bw）**

**定义**：TP Group内芯片之间的互联带宽

**单位**：GB/s（字节每秒）

**典型值**：

NVLink直连：300-600 GB/s

自研芯片直连：64 GB/s

PCIe Gen5：32-64 GB/s

**用途**：AllReduce等TP Group内的通信

**在模型中的参数**：`intra_bw`

2) **TP Group间互联带宽（inter_bw）**

**定义**：不同TP Group之间的互联带宽

**单位**：GB/s（字节每秒）

**典型值**：

InfiniBand HDR：25 GB/s（200 Gbps）

RoCE：12.5-50 GB/s（100-400 Gbps）

PCIe Gen5：16-32 GB/s

**用途**：跨TP Group的通信，如MoE的All-to-All

**在模型中的参数**：`inter_bw`

3) **带宽利用率（bwurate）**

**定义**：实际可用带宽占理论带宽的比例

**典型值**：0.8 - 0.95

**影响因素**：

协议开销（TCP/IP、RDMA）

数据包大小（小包效率低）

网络拥塞

软件栈效率

**用途**：计算实际通信时间时的修正系数

4) **链路延迟（link_delay）**

**定义**：数据传输的固定延迟开销

**组成**：

Switch延迟：交换机转发延迟（~0.1 μs/跳）

Cable延迟：光纤/铜缆传播延迟（~0.04 μs/10m）

序列化延迟：数据打包/解包时间

**典型值**：

单机内：< 1 μs

单交换机：~0.28 μs（2×switch + 2×cable）

多跳：0.5-2 μs

**用途**：All-to-All等操作的固定时间成本

**在模型中的参数**：`linkdelay`

5) **启动延迟（start_lat）和同步延迟（sync_lat）**

start_lat：通信操作启动的软件开销

sync_lat：多节点同步的时间开销

**典型值**：1-10 μs

**用途**：Tree等多步算法的固定成本累加

| 拓扑类型 | 节点度数 | 硬件成本 | 通信延迟 | 适用规模 | 典型场景 |
|---|---|---|---|---|---|
| **Ring** | 2 | 低 | O(N) | 中小型 | 通用集群 |
| **Full-Mesh** | N-1 | 高 | O(1) | 小型 | 单节点内 GPU |
| **Tree** | 1-3 | 中 | O(log N) | 中型 | 数据中心 |
| **2D Torus** | 4 | 中 | O(√N) | 大型 | 超算集群 |
| **Fat-Tree** | 变化 | 高 | O(log N) | 大型 | 数据中心 |

**第6条　通信算法实现与性能计算**

通信算法定义了**如何**（How）在特定拓扑上实现通信原语，并推导出性能计算公式。

1) AllReduce的不同实现AllReduce是最重要的通信原语，不同拓扑和算法实现差异显著。

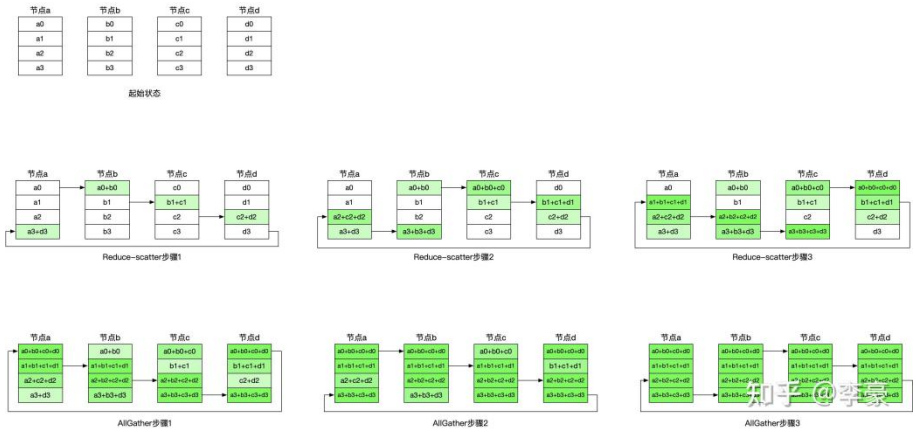



图 4.15 Ring AllReduce 示意图

2) Ring AllReduce

**适用拓扑**：Ring拓扑或任意拓扑模拟Ring

**阶段1：ReduceScatter**

**初始状态**：每个节点有 datasize 大小的数据

**数据分片**：每个节点将自己的 datasize 数据分成 N 个分片（chunk），每个分片大小 = datasize / N

**通信过程**：进行 N-1 轮通信，每轮：

每个节点将当前分片发送给后继节点

接收前驱节点发送的分片

对接收到的分片与本地对应分片进行规约（如SUM）

**结果**：每个节点拥有一个完整规约后的分片（大小为 datasize/N，但已经是所有节点对应分片的求和）

**阶段2：AllGather**

进行N-1轮通信，每轮：

每个节点将已完成规约的分片发送给后继节点

接收前驱节点发送的完整分片

结果：每个节点拥有所有分片的完整规约结果

**单节点通信量分析**：假设N个节点，每个节点有datasize字节数据

**ReduceScatter阶段：**

每轮通信量：datasize / N

通信轮数：N-1

总通信量：(N-1) × datasize / N

**AllGather阶段：**

每轮通信量：datasize / N

通信轮数：N-1

总通信量：(N-1) × datasize / N

**总通信量（每个节点）：**

$$\text{commsize} = 2 \times \frac{(N-1) \times \text{datasize}}{N}$$

**性能公式：**

$$T_{\text{AllReduce}}^{\text{Ring}} = \frac{2 \times \frac{(N-1) \times \text{datasize}}{N}}{\text{bandwidth} \times \text{bwurate}} + 2(N-1) \times \text{latency}$$

其中：

第一项：数据传输时间

第二项：N-1轮通信的延迟累加

**简化公式（当N较大时）：**

当N较大时，$(N-1)/N \approx 1$，可简化为：

$$\text{commsize} \approx 2 \times \text{datasize}$$

$$T_{\text{AllReduce}}^{\text{Ring}} \approx \frac{2 \times \text{datasize}}{\text{bandwidth} \times \text{bwurate}}$$

**特点总结：代码实现**（对应TP参数）：

| 指标 | 值 |
| --- | --- |
| 通信量（每个节点） | $2 \times \frac{(N-1)}{N} \times \text{datasize}$ |
| 延迟复杂度 | $O(N)$ |
| 带宽利用率 | 高（充分利用上下行） |
| 实现难度 | 简单 |
| 适用场景 | 通用，特别是带宽受限环境 |

```python
# Ring AllReduce 通信时间计算
def calc_allreduce_ring(datasize, TP, intrabw, bwurate):
    """
    datasize: 每个节点需要同步的数据量（字节）
    TP: 张量并行度（节点数 N）
    intrabw: TP Group 内互联带宽（字节/秒）
    bwurate: 带宽利用率
    """
    # 通信数据量
    commsize = 2 * (TP - 1) * datasize / TP

    # 通信时间（微秒）
    T_allreduce = (commsize / (intrabw * bwurate)) * 1e6

    return T_allreduce
```

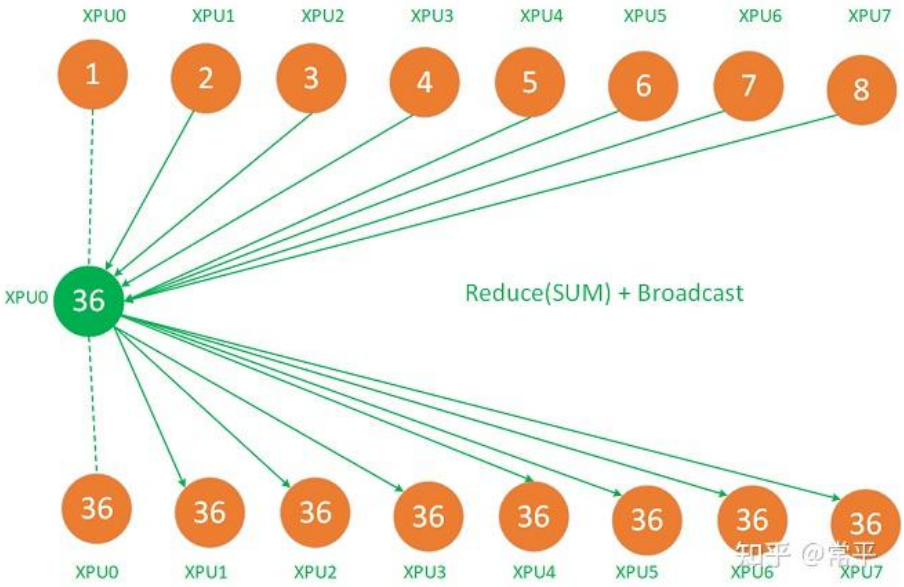



图 4.16 Full-Mesh AllReduce 示意图

3） ReduceBroadcast

**适用拓扑：**Full-Mesh全互联拓扑

**算法原理：**利用全互联拓扑，每个节点可以直接与所有其他节点通信。

**方法：**直接广播 + 本地聚合

每个节点将自己的数据广播到所有其他节点

每个节点接收所有其他节点的数据

每个节点本地进行规约运算（如SUM）

**通信量分析：**

每个节点发送：(N-1) × datasize（发送给其他N-1个节点）

每个节点接收：(N-1) × datasize（接收其他N-1个节点）

**总通信量（每个节点）：**

$$\text{commsize} = 2 \times (N-1) \times \text{datasize}$$

**性能公式：**

$$T_{\text{AllReduce}}^{\text{Full-Mesh}} = \frac{2 \times (N-1) \times \text{datasize}}{\text{bandwidth} \times \text{bwurate}} + \text{latency}$$

其中：

第一项：数据传输时间

第二项：单轮通信延迟（理论上可并行完成）

**与Ring对比：**

通信量更大：Ring为 $2 \times \frac{(N-1)}{N} \times \text{datasize}$，Full-Mesh为 $2 \times (N-1) \times \text{datasize}$

延迟更低：Full-Mesh为O(1)，Ring为O(N)

适合小规模（N<8）且延迟敏感的场景

**特点总结：代码实现：**

| 指标 | 值 |
| --- | --- |
| 通信量（每个节点） | $2 \times (N-1) \times \text{datasize}$ |
| 延迟复杂度 | $O(1)$ |
| 带宽利用率 | 中（需要高带宽支撑） |
| 实现难度 | 简单 |
| 适用场景 | 小规模、高带宽网络 |

```python
# Full-Mesh AllReduce 通信时间计算
def calc_allreduce_fullmesh(datasize, TP, intrabw, bwurate):
    """
    datasize: 每个节点需要同步的数据量（字节）
    TP: 张量并行度（节点数 N）
    intrabw: TP Group 内互联带宽（字节/秒）
    bwurate: 带宽利用率
    """
    # 通信数据量
    commsize = 2 * (TP - 1) * datasize

    # 通信时间（微秒）
    T_allreduce = (commsize / (intrabw * bwurate)) * 1e6

    return T_allreduce
```

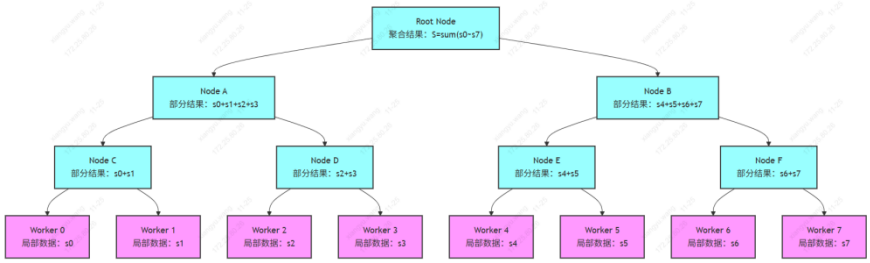



图 4.17 Tree AllReduce 示意图

4) Tree AllReduce（单树）

**适用拓扑：**Tree拓扑或Full-Mesh上构建逻辑树

**算法原理：**使用树形结构进行AllReduce，分为两个阶段：

**阶段1：Reduce（自底向上）**

从叶节点开始，向上逐层传递并规约

每个节点将规约结果发送给父节点

父节点接收所有子节点数据并规约

最终根节点获得全局规约结果

**阶段2：Broadcast（自顶向下）**

根节点将规约结果向下广播

每个节点将结果发送给子节点

最终所有节点获得相同的规约结果

**通信量分析**（以二叉树为例）：

树高度：$h = \lceil \log_2 N \rceil$（N为叶子节点数）

**Reduce阶段：**

叶子节点：向上发送 datasize

中间节点：接收子节点数据并规约后向上发送

根节点：接收子节点数据并规约

**Broadcast阶段：**

根节点：向下广播 datasize

中间节点：接收父节点数据并向下广播

叶子节点：接收 datasize

**通信量统计：**

**单个叶子节点（实际Worker）：**

发送：datasize（Reduce阶段）

接收：datasize（Broadcast阶段）

总计：$\text{commsize}_{\text{worker}} = 2 \times \text{datasize}$

**网络总通信量：**

二叉树边数 = N 1（完全二叉树性质）

每条边在两个阶段各传输一次 datasize

总计：$\text{commsize}_{\text{total}} = 2 \times (N-1) \times \text{datasize}$

**示例**（8个Worker）：

总节点数 = 15（8个叶子 + 7个内部节点）

边数 = 14

网络总通信量 = 2 × 14 × datasize = 28 × datasize

**性能公式**（单个叶子节点视角）：

$$T_{\text{AllReduce}}^{\text{Tree}} = \frac{2 \times \text{datasize}}{\text{bandwidth} \times \text{bwurate}} + 2\lceil \log_2 N \rceil \times \text{latency}$$

**公式说明：**

这是**单个叶子节点（Worker）的通信时间**

**分子"2 × datasize"是该节点在两个阶段的收发总量（Reduce发送 +**

Broadcast接收）

**bandwidth指单向带宽**（上行或下行）

第一项：数据传输时间（单个节点视角）

第二项：2h轮通信的延迟累加（h为树高，Reduce h轮 + Broadcast h轮）

**注意：**

网络总通信量 = 2 × (N - 1) × datasize（所有边的总和）

但由于树形拓扑的层次性，各层可并行，所以单个节点时间不是总量除以节点数

**特点总结：代码实现：带宽利用率分析：**

单树AllReduce存在带宽利用不足的问题：

**叶子节点**：Reduce阶段仅发送，Broadcast阶段仅接收，任意时刻仅使用单向带宽

**根节点**：Reduce阶段仅接收，Broadcast阶段仅发送，任意时刻仅使用单向带宽

**中间节点**：虽有收发操作，但不同阶段收发不平衡，带宽利用率较低

**核心问题**：所有节点均未充分利用双向带宽，整体带宽利用率约50%

**改进方向**：通过Double Binary Tree实现双向带宽的充分利用

| 指标 | 值 |
| --- | --- |
| Worker 节点通信量 | $2 \times$ datasize（最优） |
| 网络总通信量 | $2 \times (N-1) \times$ datasize |
| 延迟复杂度 | $O(\log_2 N)$ |
| 带宽利用率 | 低（单树只用单向带宽） |
| 实现难度 | 中等 |
| 适用场景 | 中等规模，平衡通信量和延迟 |

```
import math

# Tree AllReduce 通信时间计算
def calc_allreduce_tree(datasize, TP, intrabw, bwurate, startlat, synclat):
    """
    参数：
    datasize: 节点数据量（字节）
    TP: 节点数 N
    intrabw: 单向带宽（字节/秒）
    bwurate: 带宽利用率
    startlat: 启动延迟（微秒）
    synclat: 同步延迟（微秒）
    """
    commsize = 2 * datasize   # Reduce 发送 + Broadcast 接收
```

```
    tree_height = math.ceil(math.log2(TP))

    # 注意：使用单向带宽
    T_allreduce = (commsize / (intrabw * bwurate)) * 1e6 + \
                  2 * tree_height * (startlat + synclat)

    return T_allreduce
```

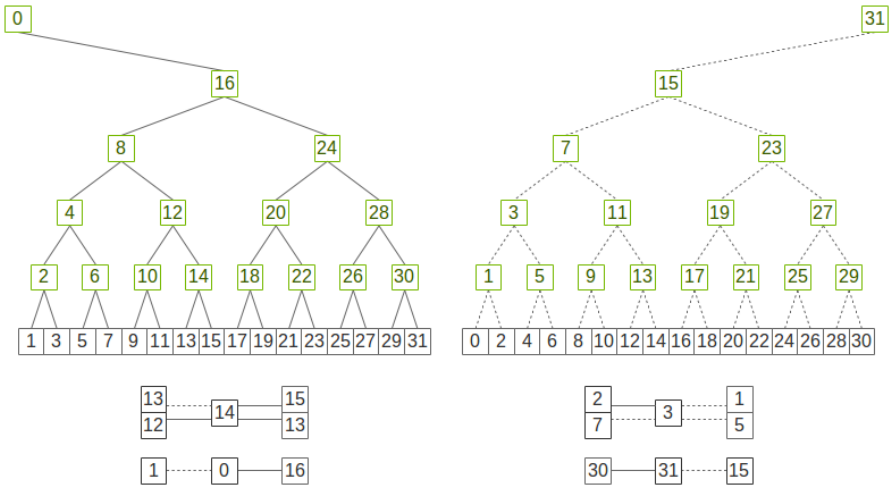



图 4.18 Double Binary Tree 拓扑图

5) Double Binary Tree AllReduce（NCCL多机默认）

**适用拓扑：**Full-Mesh或高带宽网络

**算法原理：**构建两棵互补的二叉树T1和T2，充分利用双向带宽。

**核心思想：**

T1的叶节点是T2的中间节点

T1的中间节点是T2的叶节点

两棵树同时工作，各负责一半数据

每个节点既发送又接收，充分利用双向带宽

**构建规则：关键：**物理连接不增加，只是更充分地利用已有双向带宽

**双向带宽利用机制**（以8节点为例）：**机制说明：**

数据分为两半（Data1和Data2），T1和T2各处理一半，同时执行

节点角色互补：在T1是叶子的节点，在T2是中间节点（反之亦然）

同一时刻，节点在T1中发送的同时在T2中接收，实现双向带宽利用

叠加两棵树后，除根节点和叶子节点外，中间节点均有2个父节点和2个子节点

**通信量最优性：**

每棵树处理 datasize/2，每个节点在单棵树上：发送 datasize/2，接

收 datasize/2

两棵树总计：发送 datasize/2 × 2 = datasize，接收 datasize/2 × 2 = datasize

每个节点总通信量：2 × datasize，与Ring AllReduce相同，达到理论最优

延迟复杂度更优：O(log N) vs Ring的O(N)

**性能公式：**

$$T_{\text{AllReduce}}^{\text{DBT}} = \frac{\text{datasize}}{\text{bandwidth} \times \text{bwurate}} + 2\lceil \log_2 N \rceil \times \text{latency}$$

**说明：**每棵树处理datasize/2，单节点通信量为datasize，相比单树通信时间减半。

**特点对比：代码实现：NCCL实现：**节点间通信默认使用Double Binary Tree，节点内通信根据拓扑自动选择Ring或Direct

a) **Shift构建：**将节点编号左移1位构建T2（T2 = (T1 << 1) % N）

b) **Mirror构建：**将节点编号镜像构建T2（T2[i] = T1[N-1-i]）

**逻辑拓扑 vs 物理拓扑：**

| 层面 | 说明 |
|---|---|
| 物理互联 | 节点间物理连接保持不变（如 Full-Mesh 或交换机互联） |
| 逻辑拓扑 | T1 和 T2 定义不同通信路径，使用相同物理链路 |
| 带宽利用 | 同一物理链路，T1 用上行，T2 用下行 |

| 节点 | T1 角色 | T1 操作 | T2 角色 | T2 操作 | 带宽利用 |
|---|---|---|---|---|---|
| Worker0 | 叶子 | 发送 Data1 | 中间节点 | 接收 Data2 | 上行+下行 |
| Worker1 | 中间节点 | 接收 Data1 | 叶子 | 发送 Data2 | 下行+上行 |
| Worker2 | 叶子 | 发送 Data1 | 中间节点 | 接收 Data2 | 上行+下行 |
| Worker3 | 中间节点 | 接收 Data1 | 叶子 | 发送 Data2 | 下行+上行 |

| 指标 | Tree | Double Binary Tree |
|---|---|---|
| 单节点通信量 | 2 × datasize | 2 × datasize |
| 带宽利用率 | ~50% | ~100% |
| 通信时间 | $\frac{2D}{BW}$ | $\frac{D}{BW}$ |
| 延迟复杂度 | $O(\log_2 N)$ | $O(\log_2 N)$ |
| 实现难度 | 中 | 高 |
| 典型应用 | 中等规模 | NCCL 多机通信 |

```
import math

# Double Binary Tree AllReduce 通信时间计算
def calc_allreduce_dbt(datasize, TP, intrabw, bwurate, startlat, synclat):
    """
    参数：
```

```
    datasize: 节点数据量（字节）
    TP: 节点数 N
    intrabw: 单向带宽（字节/秒）
    bwurate: 带宽利用率
    startlat: 启动延迟（微秒）
    synclat: 同步延迟（微秒）

    说明：两棵树各处理 datasize/2，并行执行，每个节点总通信量为 datasize
    """
    commsize_per_node = datasize  # 每个节点在两棵树上的总通信量
    tree_height = math.ceil(math.log2(TP))

    T_allreduce = (commsize_per_node / (intrabw * bwurate)) * 1e6 + \
                  2 * tree_height * (startlat + synclat)

    return T_allreduce
```

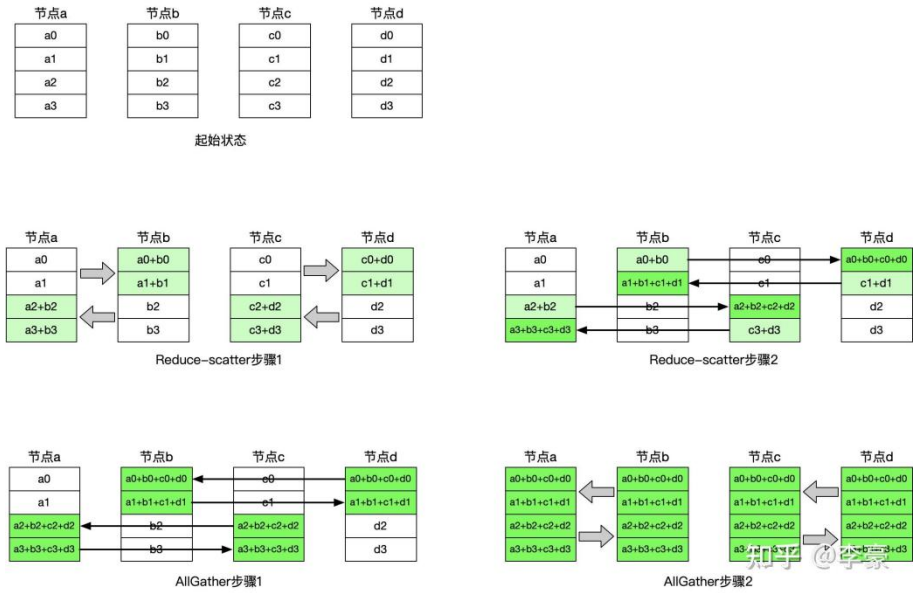



图 4.19 Halving-Doubling 示意图

6) Halving-Doubling AllReduce

**适用拓扑：**Fat-Tree等拓扑，需要节点数为2的幂次

**算法原理：**每次选择距离倍增的节点进行通信，每次通信量倍减（Reduce阶段）或倍增（AllGather阶段）。

**Reduce阶段（Halving）：**

第1轮：节点i与节点$i \oplus 2^0$通信，交换并规约一半数据

第2轮：节点i与节点$i \oplus 2^1$通信，交换并规约四分之一数据

...

第k轮：节点i与节点i⊕2^(k-1)通信

共$\log_2 N$轮

**AllGather阶段（Doubling）：**

与Reduce阶段相反

每次交换的数据量倍增

共$\log_2 N$轮

**通信量分析：**

Reduce阶段：datasize × (1/2 + 1/4 + ... + 1/N) = datasize × (N-1)/N

AllGather阶段：datasize × (1/2 + 1/4 + ... + 1/N) = datasize × (N-1)/N

**总通信量（每个节点）：**

$$\text{commsize} = 2 \times \frac{(N-1)}{N} \times \text{datasize} \approx 2 \times \text{datasize}$$

**性能公式：**

$$T_{\text{AllReduce}}^{\text{H-D}} = \frac{2 \times \frac{(N-1)}{N} \times \text{datasize}}{\text{bandwidth} \times \text{bwurate}} + 2\log_2 N \times \text{latency}$$

**特点总结：代表实现：** 昇腾HCCL、阿里ACCL

| **指标** | **值** |
|---|---|
| 通信量（每个节点） | $2 \times \frac{(N-1)}{N} \times \text{datasize}$ |
| 延迟复杂度 | $O(\log_2 N)$ |
| 带宽利用率 | 高 |
| 实现难度 | 中等 |
| 适用场景 | Fat-Tree 拓扑，2 的幂次节点数 |

| 算法 | 拓扑要求 | 通信量 | 延迟 | 带宽利用率 | 适用规模 | 代表 |
|---|---|---|---|---|---|---|
| **Ring** | 任意 | $2 \times \frac{(N-1)}{N}$ | O(N) | 很高 | 中小型 | NCCL 节点内 |
| **直接广播** | Full-Mesh | $2 \times (N-1)$ | O(1) | 中 | 小型 | 单机小卡数 |
| **单树** | Tree/Full-Mesh | $2 \times S$ | O(log N) | 低 | 中型 | 传统 MPI |
| **双树（DBT）** | Full-Mesh | $2 \times S$ | O(log N) | 很高 | 大型 | NCCL 多机 |
| **Halving-Doubling** | Fat-Tree | $2 \times \frac{(N-1)}{N}$ | O(log N) | 高 | 中大型 | HCCL/ACCL |

**选择建议**：

```
if 节点数 <= 8:
    if 全互联:
```

```
        选择 直接广播（延迟最低）
    else:
        选择 Ring（通用）
elif 节点数 <= 32:
    if 拓扑 == Fat-Tree and 节点数为 2 的幂:
        选择 Halving-Doubling
    else:
        选择 Ring
else:  # 大规模集群
    if 多机通信:
        选择 Double Binary Tree（NCCL 默认）
    else:
        选择 Ring（节点内通信）
```

7) All-to-All的算法实现All-to-All通信可以采用多种算法实现，不同算法在延迟、带宽利用率和可扩展性方面各有优劣。

a) **Pairwise Exchange（成对交换，直接法）**

**原理：**

每个节点直接与其他所有节点进行点对点数据交换

所有通信可以并行进行（如果网络支持）

类似于Full-Mesh拓扑上的直接通信

**执行步骤**（以3节点为例）：

初始状态：

`GPU0: [A0, A1, A2] （A0留给自己，A1发给GPU1，A2发给GPU2）

GPU1: [B0, B1, B2] （B0发给GPU0，B1留给自己，B2发给GPU2）

GPU2: [C0, C1, C2] （C0发给GPU0，C1发给GPU1，C2留给自己）`

通信阶段（并行执行）：

`GPU0 ⇄ GPU1: 交换 A1 ↔️ B0 GPU0 ⇄ GPU2: 交换 A2 ↔️ C0

GPU1 ⇄ GPU2: 交换 B2 ↔️ C1`

最终结果：

`GPU0: [A0, B0, C0] GPU1: [A1, B1, C1] GPU2: [A2, B2, C2]`

**通信轮数：**1轮（所有交换并行）

**每个节点通信量：**

发送：$(N-1)\times\frac{\text{datasize}}{N}$

接收：$(N-1)\times\frac{\text{datasize}}{N}$

总计：$2 \times (N-1) \times \frac{\text{datasize}}{N}$

**通信时间**（理想并行情况）：

$$T_{\text{Pairwise}} = \frac{(N-1) \times \text{datasize}/N}{\text{bandwidth}} + \text{latency}$$

当N较大时，简化为：

$$T_{\text{Pairwise}} \approx \frac{\text{datasize}}{\text{bandwidth}} + \text{latency}$$

**优点：**

延迟最低（只需1轮）

适合小规模通信（N ＜ 8）

**缺点：**

节点数多时网络拥塞严重（需要$\frac{N(N-1)}{2}$条并发连接）

对网络带宽要求高

可扩展性差

b) **Ring-based（基于环，循环法）**

**原理**：节点按环形排列，每个节点只与相邻节点通信，数据逐步在环上传递，每轮转发一个数据块，减少网络拥塞，提高带宽利用率

**执行步骤**（以3节点为例）：

初始状态：

`GPU0: [A0, A1, A2] GPU1: [B0, B1, B2] GPU2: [C0, C1, C2]`

轮次1：沿环传递（GPU0→GPU1→GPU2→GPU0）

`GPU0 → GPU1: A2 GPU1 → GPU2: B2 GPU2 → GPU0: C2 结果：GPU0: [A0, A1, C2] GPU1: [A2, B1, B2] GPU2: [C0, C1, B2*] (*表示后续会被替换)`

轮次2：继续沿环传递

`GPU0 → GPU1: A1 GPU1 → GPU2: A2 GPU2 → GPU0: C1 结果：GPU0: [A0, B0*, C1] GPU1: [A1, B1, A2*] GPU2: [A2, C1*, C2]`

轮次3：完成最后一轮

`GPU0 → GPU1: C1 GPU1 → GPU2: A1 GPU2 → GPU0: C0 结果：GPU0: [A0, B0, C0] ✓ GPU1: [A1, B1, C1] ✓ GPU2: [A2, B2, C2] ✓`

**通信轮数**：$N-1$轮

**每个节点每轮通信量：** $\frac{\text{datasize}}{N}$

**通信时间：**

$$T_{\text{Ring}} = (N-1) \times \left(\frac{\text{datasize}/N}{\text{bandwidth}} + \text{latency}\right)$$

简化为：

$$T_{\text{Ring}} = \frac{(N-1)}{N} \times \frac{\text{datasize}}{\text{bandwidth}} + (N-1) \times \text{latency}$$

当N较大时：

$$T_{\text{Ring}} \approx \frac{\text{datasize}}{\text{bandwidth}} + (N-1) \times \text{latency}$$

**优点：**网络负载均衡（每次只有N条并发连接），可扩展性好（适合大规模N），带宽利用率高

**缺点：**延迟高（需要N-1轮），对延迟敏感的场景不适用

c) **Bruck（递归加倍，对数法）**

**原理：**采用递归加倍策略，类似于Halving-Doubling AllReduce，通信轮数为$\lceil \log_2 N \rceil$，每轮交换的数据量倍增

**通信轮数：**$\lceil \log_2 N \rceil$轮

**通信时间：**

$$T_{\text{Bruck}} = \log_2 N \times \left(\frac{\text{datasize}/2}{\text{bandwidth}} + \text{latency}\right)$$

**优点：**

延迟较低（对数级）

适合中等规模（N = 16-128）

**缺点：**

实现复杂

需要特殊的数据重排逻辑

**算法对比总结**：

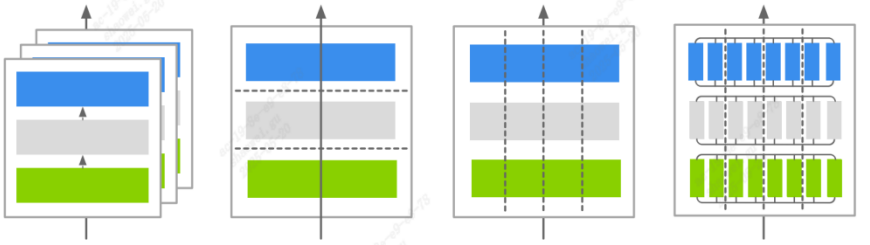
| 算法 | 通信轮数 | 每轮通信量 | 总通信时间（简化） | 网络负载 | 适用规模 |
|---|---|---|---|---|---|
| **Pairwise** | 1 | $(N-1) \times \frac{D}{N}$ | $\frac{D}{B} + \alpha$ | 高（$\frac{N(N-1)}{2}$条连接） | 小（N<8） |
| **Ring** | $N-1$ | $\frac{D}{N}$ | $\frac{(N-1)D}{NB} + (N-1)\alpha$ | 低（N 条连接） | 大（N>16） |
| **Bruck** | $\log_2 N$ | $\frac{D}{2}$（平均） | $\frac{D\log N}{2B} + \log N \cdot \alpha$ | 中 | 中（8-128） |

**注**：D = datasize，B = bandwidth，α = latency

**NCCL 的实际选择**：**小规模**（N ≤ 8）：Pairwise（延迟优先）；**中规模**（8 < N ≤ 32）：Bruck（平衡）；**大规模**（N > 32）：Ring（带宽优先）；**异构网络**：拓扑感知算法（如 NVLink 内用 Pairwise，跨节点用 Ring）

**优化技术**：**流水线（Pipelining）**：将数据分成更小的块，分阶段交换，隐藏延迟；**拓扑感知（Topology-aware）**：根据 NVLink/InfiniBand 拓扑选择最优路径；**通信计算重叠**：使用 CUDA Streams 实现异步通信，与计算并行

## 第7条　并行策略决定通信策略



| 类型 | 操作 | 节点规模 | 数据量 | 备注 |
|---|---|---|---|---|
| TP | allreduce、allgather、reducescatter | 64chip | MB-GB | Matmul流水，隐藏部分通信 |
| PP | Send/Recv | 2chip | KB-MB | 可隐藏流水 |
| DP | allreduce | 1000+ chip | GB | 计算通信重叠，可隐藏流水 |
| EP | all2all | 256chip | KB-MB | 计算通信串行，不可隐藏 |

图 4.20 并行方式

1）张量并行（Tensor Parallelism，TP）

**通信原语**：AllReduce、AllGather、ReduceScatter

**通信时机**：

前向传播：每个Transformer层的矩阵乘后需要AllReduce

反向传播：每个层的梯度计算需要通信

频率：每层2次（前向1次，反向1次）

**通信量**：

单次通信量 = batch_size × seq_len × hidden_size × dtype

例如：batch=32，seq=2048，hidden=8192，FP16 → 1 GB/层

**通信频率**：

频率：极高（每层前向、反向各1次）

70B模型（80层）→ 每步160次通信

**带宽敏感度**：极高（通信次数多）

**延迟敏感度**：极高（每次通信阻塞计算）

**必须使用高速互联**：NVLink（节点内）、InfiniBand（节点间）；不推荐使用PCIe或以太网（延迟过高）

**推荐算法**：Ring（节点内）、Tree（节点间）

**关键特性：**

TP通信是计算关键路径，延迟直接影响训练速度

TP组规模通常限制在8-16（单节点或高速互联范围）

通信与计算强耦合，难以重叠

**通信量计算公式：**

$$\text{单层通信量} = 2 \times \text{batch}_{\text{size}} \times \text{seq}_{\text{len}} \times \text{hidden}_{\text{size}} \times \text{sizeof(dtype)}$$

2) 序列并行（Sequence Parallelism, SP）与 TP+SP 组合

**背景问题：**纯TP存在的问题

在标准张量并行中，存在两类通信：LayerNorm/Dropout（非张量并行层）：

输入：[B, S/TP, H]（序列切分）

↓ 本地计算（无通信）

输出：[B, S/TP, H]（序列切分）

```

**通信原语：**AllGather + ReduceScatter（替代AllReduce）

**通信时机：**

a) **列并行**（Column Parallel）：输出需要AllReduce聚合

b) **行并行**（Row Parallel）：输入需要AllGather复制

**问题：**在非张量并行的层（如LayerNorm、Dropout），输入数据在TP组内是**完全复制的**，每个TP设备都存储完整的 [batch, seq_len, hidden_size]，造成显存浪费（seq_len越长越严重），长序列场景（seq_len=32K、64K）下显存成为瓶颈

**序列并行（SP）的核心思想：**

将序列维度也切分到TP组内，**在非张量并行的层**（LayerNorm、Dropout）上切分seq_len维度，每个设备只存储：[batch, seq_len/TP, hidden_size]，节省显存：原本复制TP份，现在只存1/TP

**TP+SP的通信模式：**

```

Attention/MLP（张量并行层）：

输入：[B, S/TP, H]（序列切分）

↓ AllGather (ReduceScatter的逆操作)

转换：[B, S, H/TP]（张量切分）

↓ 矩阵乘计算

输出：[B, S, H/TP]

↓ ReduceScatter (AllReduce的变体)

转换：[B, S/TP, H] （序列切分）

c) TP层之前：AllGather（序列切分 → 张量切分）

d) TP层之后：ReduceScatter（张量切分 → 序列切分）

e) 非TP层：无通信（直接在序列切分状态下计算）

**通信量对比**（单层，TP=8）：

| 方案 | 通信原语 | 通信量 | 显存占用 |
|---|---|---|---|
| 纯TP | AllReduce | 2 × B×S×H | B×S×H（每个设备） |
| TP+SP | AllGather + ReduceScatter | 2 × (TP-1)/TP × B×S×H | B×S/TP×H（每个设备） |

**通信量分析**：

**AllGather（序列 → 张量）**：

$$\text{AllGather通信量} = \frac{(\text{TP}-1)}{\text{TP}} \times B \times S \times H \times \text{sizeof(dtype)}$$

**ReduceScatter（张量 → 序列）**：

$$\text{ReduceScatter通信量} = \frac{(\text{TP}-1)}{\text{TP}} \times B \times S \times H \times \text{sizeof(dtype)}$$

**总通信量**（单层）：

$$\text{总通信量} = 2 \times \frac{(\text{TP}-1)}{\text{TP}} \times B \times S \times H \times \text{sizeof(dtype)}$$

当TP=8时：$2 \times \frac{7}{8} = 1.75 \times B \times S \times H$

**性能要求**：

**带宽敏感度**：极高（通信量与纯TP相近）

**延迟敏感度**：极高（每层多次通信）

**推荐互联**：NVLink（节点内）、InfiniBand（节点间）

**推荐算法**：Ring（AllGather和ReduceScatter都用Ring）

**关键特性**：

f) 通信量与纯TP基本相同（略少一点）

g) **显存节省显著**：LayerNorm/Dropout等层的显存降低到1/TP

h) 长序列场景受益明显（seq_len=32K时，8卡可节省7/8显存）

i) 需要框架支持（如Megatron-LM的Sequence Parallelism）

**适用场景**：

j) 长序列训练（seq_len > 8K）

k） 显存受限（无法容纳完整序列）

l） 与TP组合使用（不单独使用）

**通信量计算公式：**

$$\text{单层通信量} = 2 \times \frac{(\text{TP}-1)}{\text{TP}} \times B \times S \times H \times \text{sizeof(dtype)}$$

3） 数据并行（Data Parallelism, DP）

**通信原语：**AllReduce

**通信时机：**

反向传播完成后，同步所有节点的梯度

每个训练步（step）结束时执行一次

**通信量：**

单次通信量 = 模型参数量 × 数据类型大小

例如：70B模型（FP16）→ 70B × 2字节 = 140 GB

**通信频率：**

频率：每个训练步1次

可通过梯度累积降低频率（如每4步同步1次）

**带宽敏感度：**高（通信量大，需要高带宽）

**延迟敏感度：**中（AllReduce本身有O(N)或O(log N)延迟）

**推荐互联：**

节点内：NVLink（900 GB/s）

节点间：InfiniBand/RoCE（400 Gbps）

**推荐算法：**Ring（通用）、Double Binary Tree（大规模）

**通信量计算公式：**

$$\text{通信量} = 2 \times \text{参数量} \times \text{sizeof(dtype)}$$

4） 流水并行（Pipeline Parallelism, PP）

**通信原语：**Point-to-Point（Send/Recv）

**通信时机：**

前向传播：将激活值传递给下一阶段

反向传播：将梯度传递给上一阶段

微批次（micro-batch）间流水执行

**通信量：**

单次通信量 = micro_batch_size × seq_len × hidden_size × dtype

例如：micro_batch=4, seq=2048, hidden=8192, FP16 → 128 MB/微

批次

**通信频率：**

频率：中等（取决于微批次数量）

每个阶段与相邻阶段通信（非全局通信）

num_micro_batches × 2（前向+反向）

**带宽敏感度：**中（通信量中等）

**延迟敏感度：**高（延迟影响流水效率，产生气泡）

**推荐互联：**

节点间：InfiniBand/RoCE（延迟 < 5 μs）

可容忍相对较低带宽，但需低延迟

**推荐拓扑：**线性拓扑（stage0 → stage1 → ... → stageN）

**关键特性：**

通信只在相邻阶段间进行，非全局通信

气泡时间（bubble time）受延迟影响大

可与计算部分重叠（异步通信）

**通信量计算公式：**

$$\text{单阶段通信量} = 2 \times \text{num}_{\text{micro\_batches}} \times \text{micro}_{\text{batch\_size}} \times \text{seq}_{\text{len}} \times \text{hidden}_{\text{size}} \times \text{sizeof(dtype)}$$

5) 专家并行（Expert Parallelism, EP/MoE）

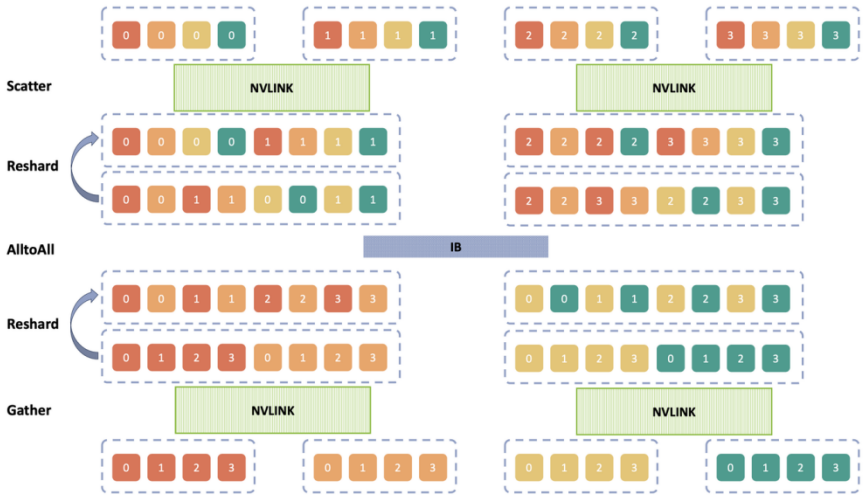



图 4.21 负载均衡 All-to-All 示意图

**通信原语**：All-to-All

**通信时机**：MoE 层通过两次 All-to-All 通信实现 token 在专家间的分发与收集。第一次 All-to-All 称为 **Dispatch（分发）目的是将 token 根据路由决策发送到负责的专家所在节点，第二次 All-to-All 称为 Combine（收集）**，目的是将不同专家针对同一个 token 的输出结果汇总（一般是发回 token 原始所在节点）。

**通信量**：取决于 top_k、路由策略和负载均衡，基础 EP：top_k × B × S × H × sizeof(dtype)，如果 EP+TP（moe_tp>1）：还需额外的 AllGather 通信

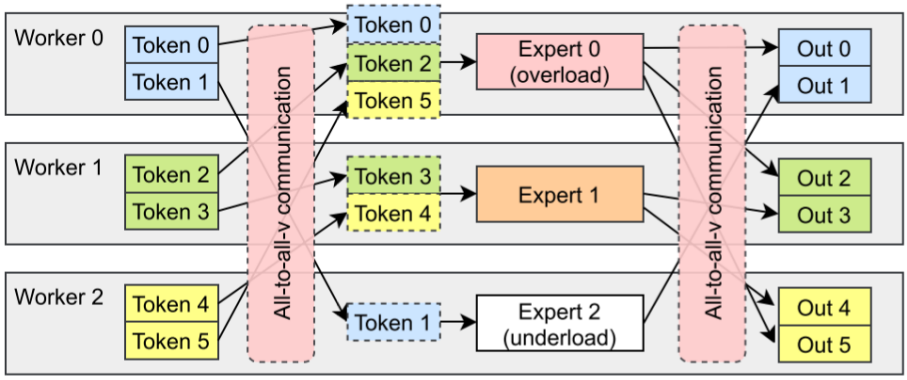



图 4.22 负载不均衡 All-to-All 示意图

专家并行具体过程（EP，moe_tp = 1）：**每个专家完整地部署在一个设备上，不同专家分布在不同设备上，通信只需跨专家节点的 All-to-All（以 3 节点、3 专家为例）：

6) **初始状态：**

节点0：有token [0, 1]

节点1：有token [2, 3]

节点2：有token [4, 5]

7) **路由决策**

路由权重是**动态计算的**，每个token都不同（根据token内容），路由网络（Gate）是一个可学习的参数，在训练中更新，每个token的副本会被发送到top-k个专家所在的节点，路由权重用于Combine后的加权聚合，如果top_k=2，通信量是top_k=1的两倍

8) **Dispatch通信**（All-to-All）：

根据top-k路由，以图中单节点专家数=1且top-k=1为例：

节点0将token 0 → 节点0（Expert 0），将token 1 → 节点2（Expert 2）

节点1将token 2 → 节点0（Expert 0），将token 3 → 节点1（Expert 1）

节点2将token 4 → 节点1（Expert 1），将token 5 → 节点0（Expert 0）

**关键**：top-k > 1 时，同一个token可能被发送到多个节点（top-k个）

**问题**：负载不均衡，可以通过设置单个专家接收token的上限，让后续超限token选择次好的专家进行负载均衡。

**Dispatch 通信量**：top_k × num_tokens × hidden_size ×

sizeof(dtype)

**Dispatch通信时间**：使用跨节点带宽（inter_bw），考虑CPU指令响应延迟

$$T_{\text{dispatch-EP}} = \frac{\text{datasize}}{\text{inter}_{\text{bw}} \times \text{bwurate}} \times 10^6 + \text{linkdelay} + \text{cpufetch}$$

9) **Dispatch后状态**：

节点0：收到所有被路由到Expert 0的token

节点1：收到所有被路由到Expert 1的token

节点2：收到所有被路由到Expert 3的token

**关键**：单节点专家数 > 1 时，单节点可能接收重复的token，通信逻辑里可以只发送一次，在节点中复用即可

10) **专家计算（Local Computation，无通信）**

每个节点在本地执行专家前向计算（FFN），每个专家是一个独立的FFN网络，有自己的参数（W1, W2等），**这些权重是训练好的固定参数，对所有token都一样**，不同batch、不同token使用**同一组**专家权重

11) **计算后状态**：

节点0：有Expert 0的输出，out 0、out 2、out 5

节点1：有Expert 1的输出，out 3、out 4

节点2：有Expert 2的输出，out 1

12) **Combine通信**（All-to-All，与Dispatch相反）：

节点0将out 0 → 节点0、out 2 → 节点1、out 5 → 节点2

节点1将out 3 → 节点1、out 4 → 节点2

节点2将out 1 → 节点0

**关键**：top-k > 1 时，每个token会收到top-k个专家的输出

**问题**：Dispatch的负载不均衡会直接传导到Combine——专家处理的token越多，Combine时需要发送的输出也越多。**两次All-to-All通信的是对称的**

**Combine通信量**：top_k × num_tokens × hidden_size × sizeof(dtype)

**Combine通信时间**：使用跨节点带宽（inter_bw），考虑CPU指令响应延迟

$$T_{\text{combine-EP}} = \frac{\text{datasize}}{\text{inter}_{\text{bw}} \times \text{bwurate}} \times 10^6 + \text{linkdelay} + \text{cpufetch}$$

13) **Combine后状态**（恢复原始分布，但每个token有多个专家输出）：

节点0收到：token 0的所有专家输出，token 1的所有专家输出

节点1收到：token 2的所有专家输出，token 3的所有专家输出

节点2收到：token 4的所有专家输出，token 5的所有专家输出

**形状：**[batch_size, seq_len, top_k, hidden_size]

14) **局部加权聚合**（Combine后的本地操作，无通信）：

每个token经过top-k个专家，得到top-k个输出向量，使用路由网络计算的权重进行加权求和，最终输出维度：[batch, seq_len, hidden_size]（与输入维度相同。

**计算过程：**

```
# Combine 后每个 token 有 top_k 个专家输出
expert_outputs = [expert_0_output, expert_1_output, ...]  # 形状：[top_k, hidden_size]
gate_weights = [weight_0, weight_1, ...]                  # 来自路由网络，形状：[top_k]

# 加权求和
final_output = sum(weight_i * expert_i_output for i in range(top_k))
# 形状：[hidden_size]
```

15) 专家并行 + 张量并行（EP + TP，moe_tp > 1）

专家内部使用张量并行，不同专家分布在不同设备上（EP维度），每个专家内部使用张量并行（TP维度），分布在moe_tp个设备上，通信需要跨专家节点的All-to-All + 专家内部的Scatter/Gather,或者跨到专家group内的所有节点的All-to-All+专家内部的AllGather

a) **MoE完整流程总结（EP+TP场景）：**

| 步骤 | 操作 | 通信原语 | 数据分布变化 | 数据形状 | 通信量（主设备视角） |
|---|---|---|---|---|---|
| 0. 输入 | MoE 层输入 | - | 数据并行分布 | [B, S, H] | - |
| 1. 路由 | 计算 top-k 专家 | 无（本地） | 按 batch（不变） | [B, S, top_k] | 0 |
| 2. Dispatch-EP | token 分发到专家节点 | All-to-All | 按 batch → 按 expert | [tokens_per_expert, H] | top_k × B×S×H |
| 2'. Dispatch-TP | 专家内 token 分发 | Scatter（仅 moe_tp>1） | 分发到各 TP 设备 | [tokens/moe_tp, H] | (moe_tp-1)/moe_tp × datasize |
| 3. Expert 计算 | FFN 前向 | 无（本地） | 按 expert（不变） | [tokens/moe_tp, H] | 0 |
| 4'. Combine-TP | 专家内输出收集 | Gather（仅 moe_tp>1） | 收集各 TP 输出 | [tokens_per_expert, H] | (moe_tp-1)/moe_tp × datasize |
| 4. Combine | 专家输出收集 | All-to-All | 按 expert → 按 batch | [B, S, top_k, H] | top_k × B×S×H |

| | | | | | |
|---|---|---|---|---|---|
| -EP | | | | | |
| 5. 加权聚合 | 多专家输出求和 | 无（本地） | 按 batch（不变） | [B, S, H] | 0 |

b) **方案1：Scatter/Gather（当前方案）**

**Dispatch后的Scatter（分发token到专家内各TP设备）：**

**通信类型：**Scatter（一对多分发）

**过程：**跨EP的All-to-All将token发送到专家组后，由专家组内的一个主设备（如rank 0）将token分发到moe_tp个TP设备

**通信量：**主设备发送$\frac{(\text{moe}_{\text{tp}}-1)}{\text{moe}_{\text{tp}}} \times \text{datasize}$

**使用带宽：**intra_bw（专家内高速互联，如NVLink）

$$T_{\text{Scatter-TP}} = \frac{\frac{(\text{moe}_{\text{tp}}-1)}{\text{moe}_{\text{tp}}} \times \text{datasize}}{\text{intra}_{\text{bw}} \times \text{bwurate}} \times 10^6$$

**Combine前的Gather（收集专家输出到一个设备）：**

**通信类型：**Gather（多对一收集）

**过程：**各TP设备将自己处理的token输出发送到专家组内的一个主设备，然后通过跨EP的All-to-All发送回原始位置

**通信量：**主设备接收$\frac{(\text{moe}_{\text{tp}}-1)}{\text{moe}_{\text{tp}}} \times \text{datasize}$

$$T_{\text{Gather-TP}} = \frac{\frac{(\text{moe}_{\text{tp}}-1)}{\text{moe}_{\text{tp}}} \times \text{datasize}}{\text{intra}_{\text{bw}} \times \text{bwurate}} \times 10^6$$

**方案1总通信时间：**

$$T_{\text{intra-expert}}^{\text{方案1}} = T_{\text{Scatter-TP}} + T_{\text{Gather-TP}} = 2 \times \frac{\frac{(\text{moe}_{\text{tp}}-1)}{\text{moe}_{\text{tp}}} \times \text{datasize}}{\text{intra}_{\text{bw}} \times \text{bwurate}} \times 10^6$$

c) **方案2：Group-wise All2All + AllGather（优化方案）**

**核心思想：**省略专家组内的Scatter/Gather，转化为AllGather。**TP切分的是权重矩阵，不是输入数据（token）**，以FFN层为例：`Y = X • W1`，其中X是输入token，W1是权重，列并行：W1按列切分为`[W1_0, W1_1, ..., W1_n]`，每个TP设备持有`W1_i`，但需要**完整的输入X**，各设备计算`Yi = X • W1_i`，输出拼接得到完整Y，**因此**，当使用Group-wise All2All把token分发到所有节点，token数据需要在TP设备间广播/同步（通过AllGather）

**Dispatch阶段（Group-wise All-to-All + AllGather）：**

**步骤1：**Group-wise All-to-All

发送端按TP group分组，直接将token发送到目标专家的各个TP rank，

例如：token要去Expert 0（moe_tp=4），直接分别发送到Expert 0的rank 0, 1, 2, 3，每个TP设备接收到`1/moe_tp`的token数据，**省略了**专家组内的Scatter步骤

**步骤2**：专家组内AllGather

各TP设备接收到部分token后，通过AllGather同步完整token数据，**原因**：TP切分的是权重，但需要完整的输入token进行矩阵计算，**通信量**：$\frac{(\text{moe}_{\text{tp}}-1)}{\text{moe}_{\text{tp}}} \times \text{datasize}$

$$T_{\text{AllGather-TP}} = \frac{\frac{(\text{moe}_{\text{tp}}-1)}{\text{moe}_{\text{tp}}} \times \text{datasize}}{\text{intra}_{\text{bw}} \times \text{bwurate}} \times 10^6$$

**Combine阶段（ReduceScatter + Group-wise All-to-All）：**

**步骤1**：专家组内ReduceScatter

各TP设备完成FFN计算后，得到部分输出（因为权重是切分的），通过ReduceScatter聚合输出并分发：`Y = [Y0, Y1, ..., Yn] → 聚合并分发`。**原因**是各设备的输出需要求和（通过ReduceScatter完成求和并分发）

**通信量**：$\frac{(\text{moe}_{\text{tp}}-1)}{\text{moe}_{\text{tp}}} \times \text{datasize}$

**步骤2**：Group-wise All-to-All

各TP rank持有部分完整输出后，直接通过All-to-All发送回原始节点，**省略了**专家组内的Gather步骤

$$T_{\text{ReduceScatter-TP}} = \frac{\frac{(\text{moe}_{\text{tp}}-1)}{\text{moe}_{\text{tp}}} \times \text{datasize}}{\text{intra}_{\text{bw}} \times \text{bwurate}} \times 10^6$$

**方案2总通信时间：**

$$T_{\text{intra-expert}}^{\text{方案2}} = T_{\text{AllGather-TP}} + T_{\text{ReduceScatter-TP}} = 2 \times \frac{\frac{(\text{moe}_{\text{tp}}-1)}{\text{moe}_{\text{tp}}} \times \text{datasize}}{\text{intra}_{\text{bw}} \times \text{bwurate}} \times 10^6$$

d) **两种方案对比：**

| 方案 | Dispatch 通信 | Combine 通信 | 优点 | 缺点 |
|---|---|---|---|---|
| 方案 1：Scatter/Gather | All-to-All → Scatter | Gather → All-to-All | 实现简单，单点发送 | Scatter/Gather 串行，带宽利用率低 |
| 方案 2：Group-wise + AllGather | Group-wise All-to-All → AllGather | ReduceScatter → Group-wise All-to-All | AllGather/ReduceScatter 并行，带宽利用率高 | 实现复杂，需要 TP group 信息 |

e) **通信量对比：**

两种方案的**理论通信量相同**：都是$2 \times \frac{(\mathrm{moe_{tp}}-1)}{\mathrm{moe_{tp}}} \times \mathrm{datasize}$

但方案2的**实际性能更好**，因为AllGather/ReduceScatter可以充分利用双向带宽，方案1的Scatter/Gather是单向通信，带宽利用率约50%，方案2的AllGather/ReduceScatter是双向通信，带宽利用率接近100%

f) **选择建议：**

moe_tp较小(≤2)：方案1实现简单，性能差异不大；moe_tp较大(>2)：方案2性能优势明显，推荐使用。现代框架（如Megatron-LM、DeepSpeed）通常采用方案2