(1) 基础知识

- 0x00 摘要
- 0x01 分布式并行训练
  - 1.1 分布式并行训练的必要
  - 1.2 分布式训练
  - 1.3 训练并行机制
    - 1.3.1 三种机制
    - 1.3.2 如何使用
  - 1.4 数据并行训练
- 0x02 通信 & 架构
  - 2.1 方法和架构
  - 2.2 异步 vs 同步
- 0x03 具体架构
  - 3.1 MapReduce
  - 3.2 参数服务器 (PS)
  - 3.3 Decentralized Network
- 0x04 All Reduce
  - 4.1 参数服务器劣势
  - 4.2 并行任务通信分类
  - 4.3 MPI AllReduce
- 0x05 ring-allreduce
  - 5.1 特点
  - 5.2 策略
    - 5.2.1 结构
    - 5.2.2 Scatter-Reduce
      - 5.2.2.1 分块
      - 5.2.2.2 第一次迭代
      - 5.2.2.3 全部迭代
    - 5.2.3 Allgather
      - 5.2.3.1 第一次迭代
      - 5.2.3.2 全部迭代
    - 5.2.4 Horovod 架构图
    - 5.2.5 百度思路
  - 5.3 区别

0x01 分布式并行训练

我们首先要介绍下分布式并行训练。

1.1 分布式并行训练的必要

传统的模型训练中，迭代计算只能利用当前进程所在主机上的所有硬件资源，可是单机扩展性始终有限。而目前的机器学习有如下特点：

- 样本数量大。目前训练数据越来越多，在大型互联网场景下，每天的样本量可以达到百亿级别。
- 特征维度多。因为巨大样本量导致机器学习模型参数越来越多，特征维度可以达到千亿或者万亿级别。
- 训练性能要求高。虽然样本量和模型参数巨大，但是业务需要我们在短期内训练出一个优秀的模型来验证。
- 模型实时上线。对于推荐资讯类应用，往往要求根据用户最新行为及时调整模型进行预测。

因此，单机面对海量数据和巨大模型时是无能为力的，有必要把数据或者模型分割成为多分，在多个机器上借助不同主机上的硬件资源进行训练加速。

1.2 分布式训练

本文所说的训练，指的是利用训练数据通过计算梯度下降的方式迭代地去优化神经网络参数，并最终输出网络模型的过程。在单次模型训练迭代中，会有如下操作：

- 首先利用数据对模型进行前向的计算。所谓的前向计算，就是将模型上一层的输出作为下一层的输入，并计算下一层的输出，从输入层一直算到输出层为止。
- 其次会根据目标函数，我们将反向计算模型中每个参数的导数，并且结合学习率来更新模型的参数。

而并行梯度下降的基本思想便是：多个处理器分别利用自己的数据来计算梯度，最后通过聚合或其他方式来实现并行计算梯度下降以加速模型训练过程。 比如两个处理器分别处理一半数据计算梯度 g_1，g_2，然后把两个梯度结果进行聚合更新，这样就实现了并行梯度下降。

1.3 训练并行机制

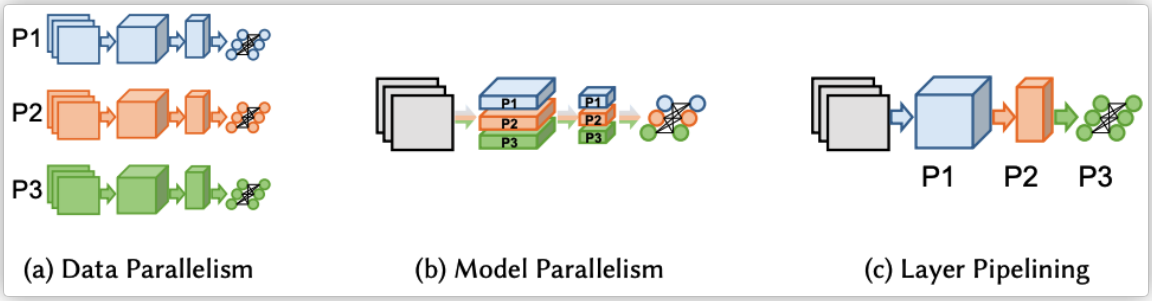
1.3.1 三种机制

由于使用小批量算法，可以把宽度（$\propto W$）和深度（$\propto D$）的前向传播和反向传播分发到并行的处理器上，这样深度训练的并行机制主要有三种：

- 第一个是模型并行机制（按照网络结构分区）。
    - 通常是针对一个节点无法存下整个模型的情况下，去对图进行拆分。
    - 将模型参数进行分布式存储。计算机上每个计算可以建模为一个有向无环图（DAG），顶点是计算指令，边是数据依赖（数据流）。 "基于图去拆分" 会根据每一层中的神经元（即四维张量中的C、H或W维）来把一张大的图拆分成很多部分，每个部分都会在很多设备上去计算。
    - 或者可以这么理解：深度学习的计算主要是矩阵运算，有时候矩阵非常大无法放到显存中，就只能把超大矩阵拆分了放到不同卡上计算。
    - 模型较后部分的计算必须等前面计算完成，因此不同节点间的计算实际是串行的。但每个部分计算互不妨碍，更像是流水线结构。
- 第二个是数据并行机制（按照输入样本分区）。
    - 更多场景下我们模型规模不大，在一张 GPU 可以容纳，但是训练数据量会比较大，这时候就采用数据并行机制。
    - 具体就是在多节点上并行分割数据和训练。
- 第三种不常用的并行机制是 流水线机制（按层分区）。
    - 在深度学习中，流水线可以是指重叠的计算，即在一层和下一层之间（当数据准备就绪时）连续计算；或者根据深度划分DNN，将层分配给特定处理器。

- 流水线可以看作是数据并行的一种形式，因为元素（样本）是通过网络并行处理的，但也可以看作是模型并行，因为流水线的长度是由DNN结构决定的。

具体可见下图：



1.3.2 如何使用

数据的并行往往意味着计算性能的可扩展，而模型的并行往往意味着内存使用的可扩展。

需要注意的是：数据并行和模型并行也并不冲突，两者可以同时存在，而流水线机制也可以和模型并行一起混用。比如，DistBelief分布式深度学习系统结合了三种并行策略。训练在同时复制的多个模型上训练，每个模型副本在不同的样本上训练（数据并行），每个副本上，依据同一层的神经元（模型并行性）和不同层（流水线）上划分任务，进行分布训练。

另外也需要根据具体问题具体分析，比如现代卷积神经网络主要由两种层构成，他们具有不一样的属性和性能。

- **卷积层**，占据了90% ~ 95% 的计算量，5% 的参数，但是对结果具有很大的表达能力。
- **全连接层**，占据了 5% ~ 10% 的计算量， 95% 的参数，但是对于结果具有相对较小的表达的能力。

综上：卷积层计算量大，所需参数系数 W 少，全连接层计算量小，所需参数系数 W 多。因此对于卷积层适合使用数据并行，对于全连接层适合使用模型并行。

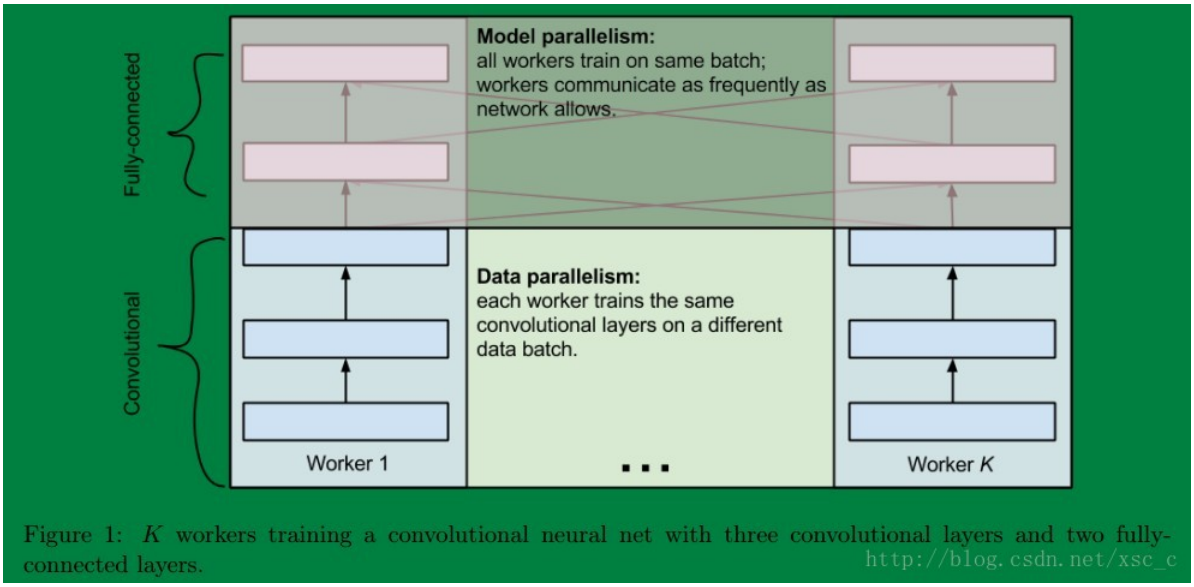


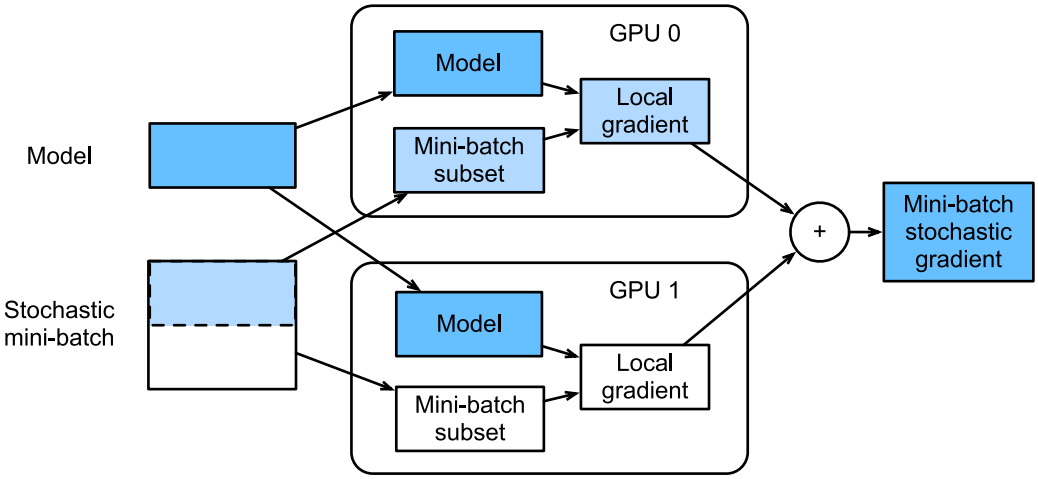
Figure 1: K workers training a convolutional neural net with three convolutional layers and two fully-connected layers.

1.4 数据并行训练

我们本系列主要讨论数据并行训练（其中的一种架构）。

数据并行训练只是一种逻辑架构。我们从沐神的书里面摘录：

> 假设机器上有kk个GPU。给定要训练的模型，每个GPU将独立地维护一组完整的模型参数，尽管GPU上的参数值是相同且同步的。例如，下图演示了在k=2k=2时使用数据并行的训练。



一般来说，训练过程如下：

- 在训练的任何迭代中，给定一个随机的小批量，我们将该小批量中的样本分成kk个部分，并将它们均匀地分在多个GPU上。
- 每个GPU根据分配给它的小批量子集计算模型参数的损失和梯度。
- 将kk个GPU中每个GPU的局部梯度聚合以获得当前的小批量随机梯度。
- 聚合梯度被重新分配到每个GPU。
- 每个GPU使用这个小批量随机梯度来更新它维护的完整的模型参数集。

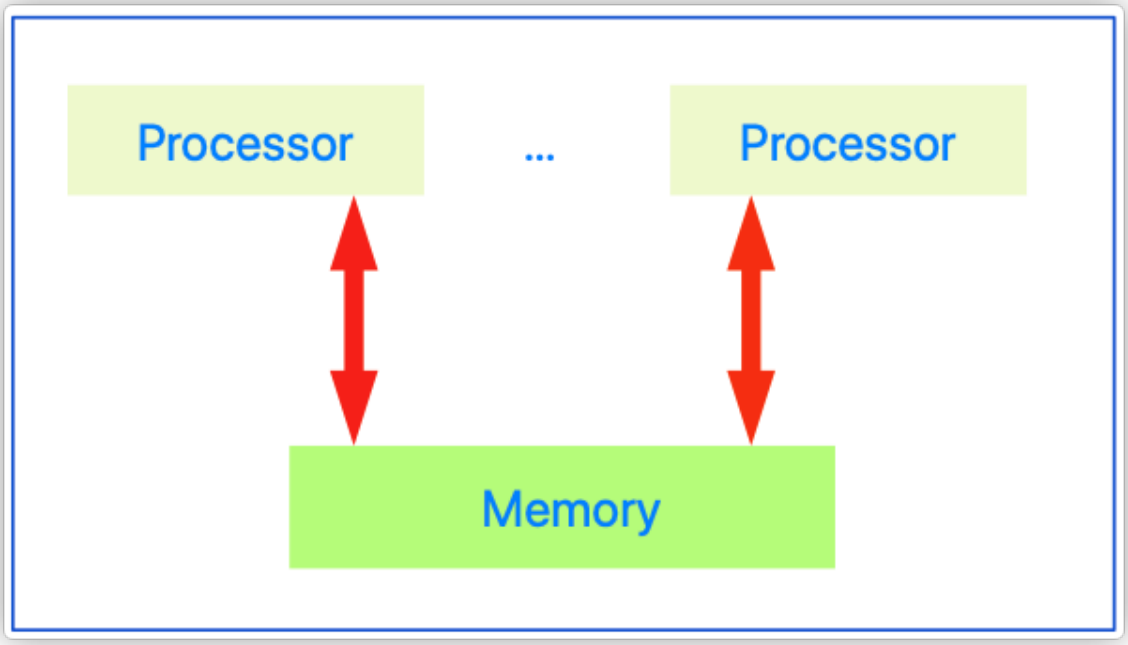
0x02 通信 & 架构

前面提到并行梯度下降的例子：两个处理器分别处理一般数据计算梯度 g_1, g_2，然后把两个梯度结果进行聚合，最后再把最新参数发给各个分布计算单元，这种训练算法叫**模型一致性方法**（consistent model methods）。这就涉及到了通信问题，即如何做聚合。

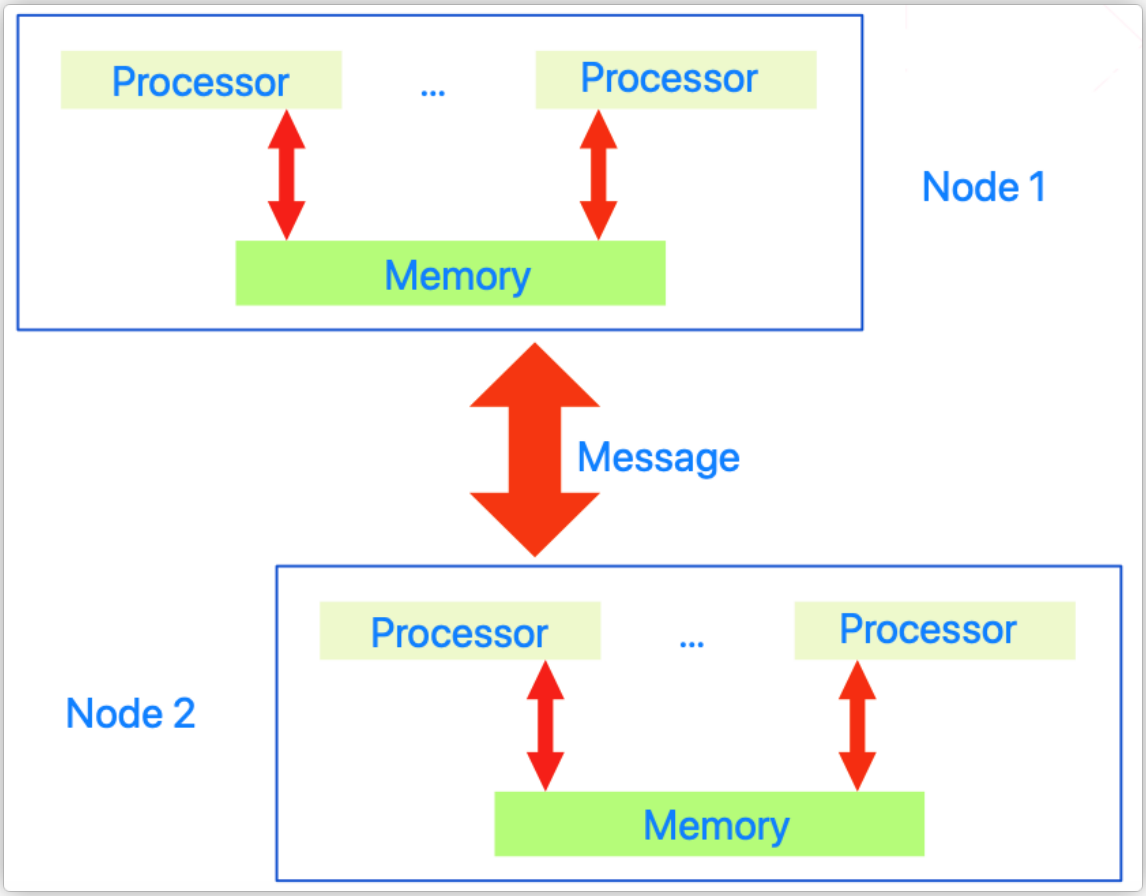
2.1 方法和架构

一般有两种通信方法：Share memory 和 Message passing。

- Share memory 就是所有处理器共享同一块内存，这样通信很容易，但是同一个节点内的处理器之间才可以共享内存，不同节点处理器之间无法共享内存。



- Message passing 就是不同节点之间用消息（比如基于 TCP/IP 或者 RDMA）进行传递/通信，这样容易扩展，可以进行大规模训练。



因此我们知道，Message passing 才是解决方案，于是带来了问题：如何协调这些节点之间的通讯。

有两种架构：

- Client-Server 架构: 一个 server 节点协调其他节点工作，其他节点是用来执行计算任务的 worker。
- Peer-to-Peer 架构：每个节点都有邻居，邻居之间可以互相通信。

2.2 异步 vs 同步

异步 vs 同步 是通信的另外一个侧面。

在数据并行训练之中，各个计算设备分别根据各自获得的batch，前向计算获得损失，进而反向传播计算梯度。计算好梯度后，就涉及到一个**梯度同步的问题**：每个 计算设备 都有根据自己的数据计算的梯度，如何在不同GPU之间维护模型的不同副本之间的一致性。 如果不同的模型以某种方式最终获得不同的权重，则权重更新将变得不一致，并且模型训练将有所不同。

怎么做这个同步就是设计分布式机器学习系统的一个核心问题。

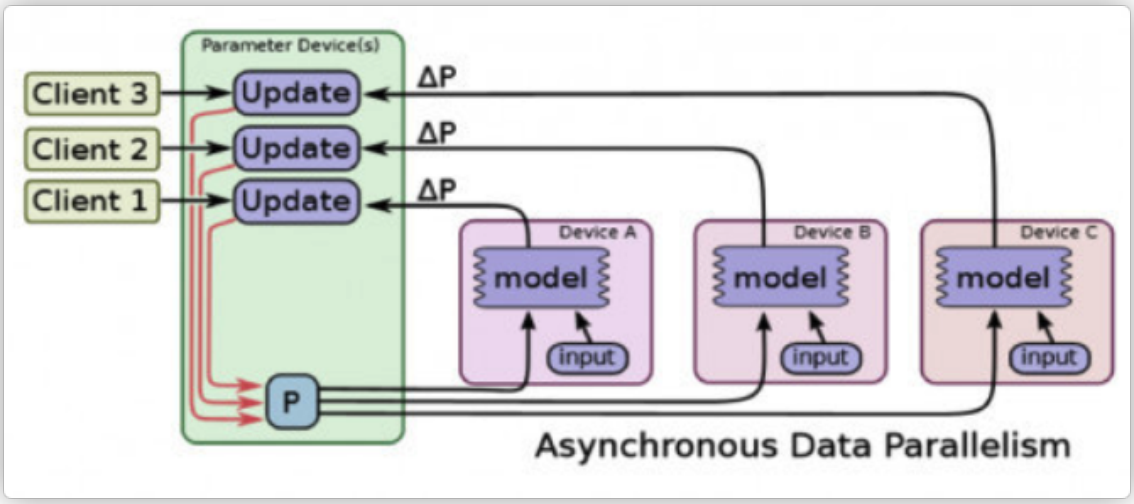
分布式训练的梯度同步策略可分为异步（asynchronous）梯度更新 和 同步（synchronous）梯度更新机制。

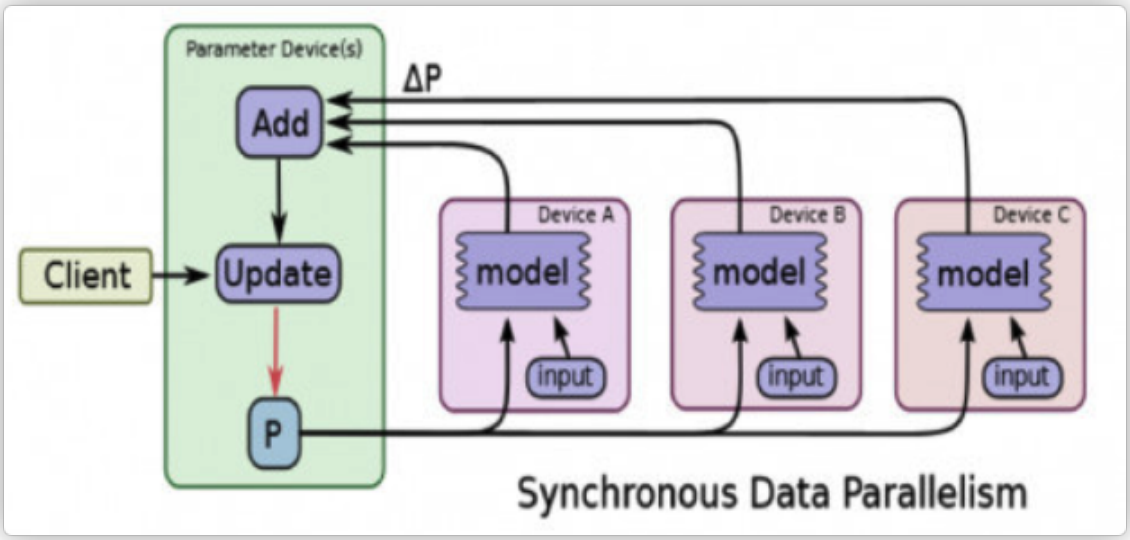
- 同步指的是所有的设备都是采用相同的模型参数来训练，等待所有设备的mini-batch训练完成后，收集它们的梯度然后取均值，然后执行模型的一次参数更新。
    - 同步训练相当于通过聚合很多设备上的mini-batch形成一个很大的batch来训练模型，Facebook就是这样做的，但是他们发现当batch大小增加时，同时线性增加学习速率会取得不错的效果。
    - 同步训练看起来很不错，但是实际上需要各个设备的计算能力要均衡，而且要求集群的通信也要均衡。
    - 因为每一轮结束时算得快的节点都需等待算得慢的节点算完，再进行下一轮迭代。类似于木桶效应，一个拖油瓶会严重拖慢训练进度，所以同步训练方式相对来说训练速度会慢一些。这个

拖油瓶一般就叫做 straggler。

- 异步训练中，各个设备完成一个mini-batch训练之后，不需要等待其它节点，直接去更新模型的参数，这样总体会训练速度会快很多。
    - 异步训练的一个很严重的问题是梯度失效问题（stale gradients），刚开始所有设备采用相同的参数来训练，但是异步情况下，某个设备完成一步训练后，可能发现模型参数其实已经被其它设备更新过了，此时这个梯度就过期了，因为现在的模型参数和训练前采用的参数是不一样的。由于梯度失效问题，异步训练虽然速度快，但是可能陷入次优解（sub-optimal training performance）。

具体如下图所示：





这两种更新方式各有优缺点：

- 异步更新可能会更快速地完成整个梯度计算。
- 同步更新 可以更快地进行一个收敛。

选择哪种方式取决于实际的应用场景。

0x03 具体架构

接下来，我们看看几种具体架构实现，先给出一个总体说明：

名称	通信	架构	并行性
MapReduce	消息传递	client-server	批同步
Parameter Server	消息传递	client-server	异步
Decentralized	消息传递	P2P	同步或异步

3.1 MapReduce

MapReduce是Client-Server架构。以 Spark 为例看看是如何进行并行化：

- Spark Driver 就是 Server，Spark Executor 就是 Worker 节点，每一个梯度下降过程包含一个广播、map和一个 reduce 操作。
- Server 定义了 map操作（就是具体的训练），也可以把信息广播到worker节点。
- Worker 会执行 map 操作进行训练，在此过程中，数据被分给 worker 进行计算。
- 计算结束后，worker把计算结果传回 driver 处理，这个叫做reduce。
- 在 reduce 过程中，Server 节点对 worker 传来的计算结果进行聚合之后，把聚合结果广播到各个worker节点，进行下一次迭代。

3.2 参数服务器 (PS)

Parameter server 也是一种client-server架构。和MapReduce不同在于 Parameter server 可以是异步的，MapReduce只有等所有map都完成了才能做reduce操作。

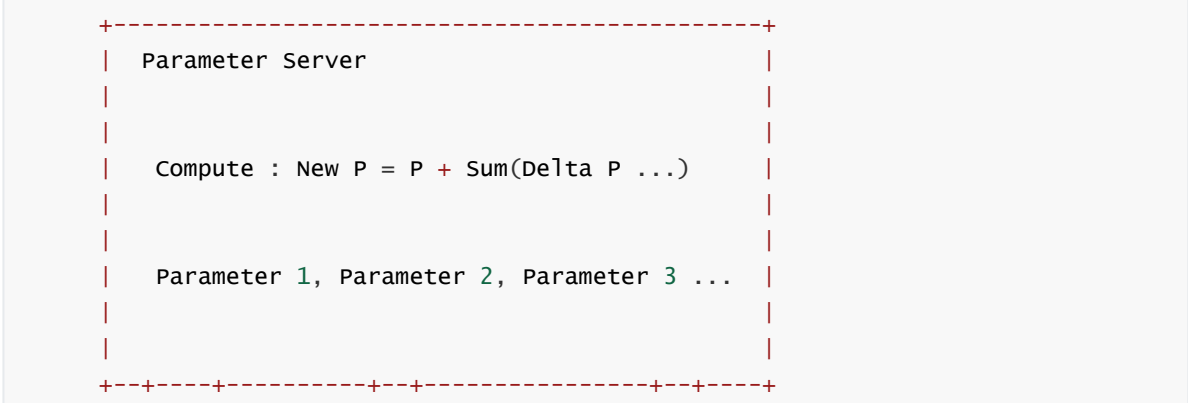
在参数服务器架构中，计算设备被划分为参数服务器（PS）和worker。

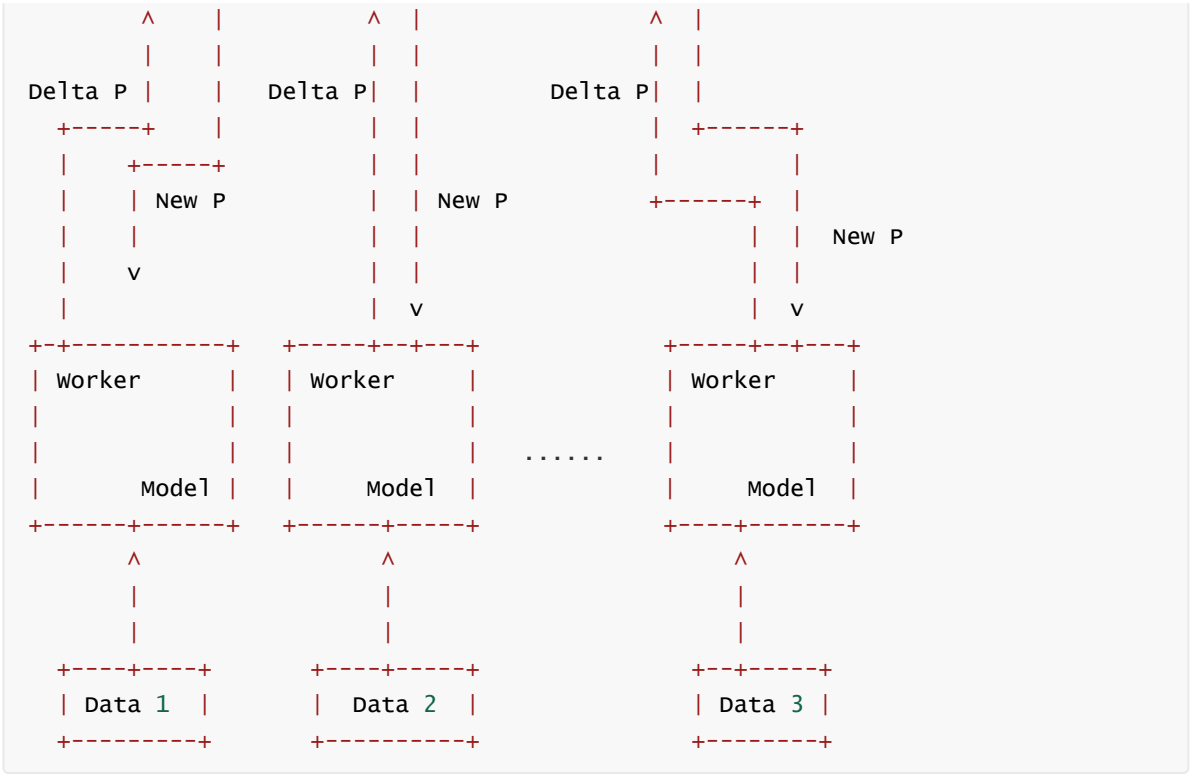
- 参数服务器（server）。是中心化的组件，主要是负责模型参数的存储，平均梯度和交换更新。参数服务器可以按照不同比例的参数服务器和工作线程进行配置，每个参数服务器都有着不同的配置数据。
- 工作节点（worker）。每个工作节点会负责它领域内的数据分片所对应模型参数的更新计算（比如前向和反向传播这类计算密集的运算），同时它们又会向参数服务器去传递它所计算的梯度，由参数服务器来汇总所有的梯度，再进一步反馈到所有节点。

具体步骤如下：

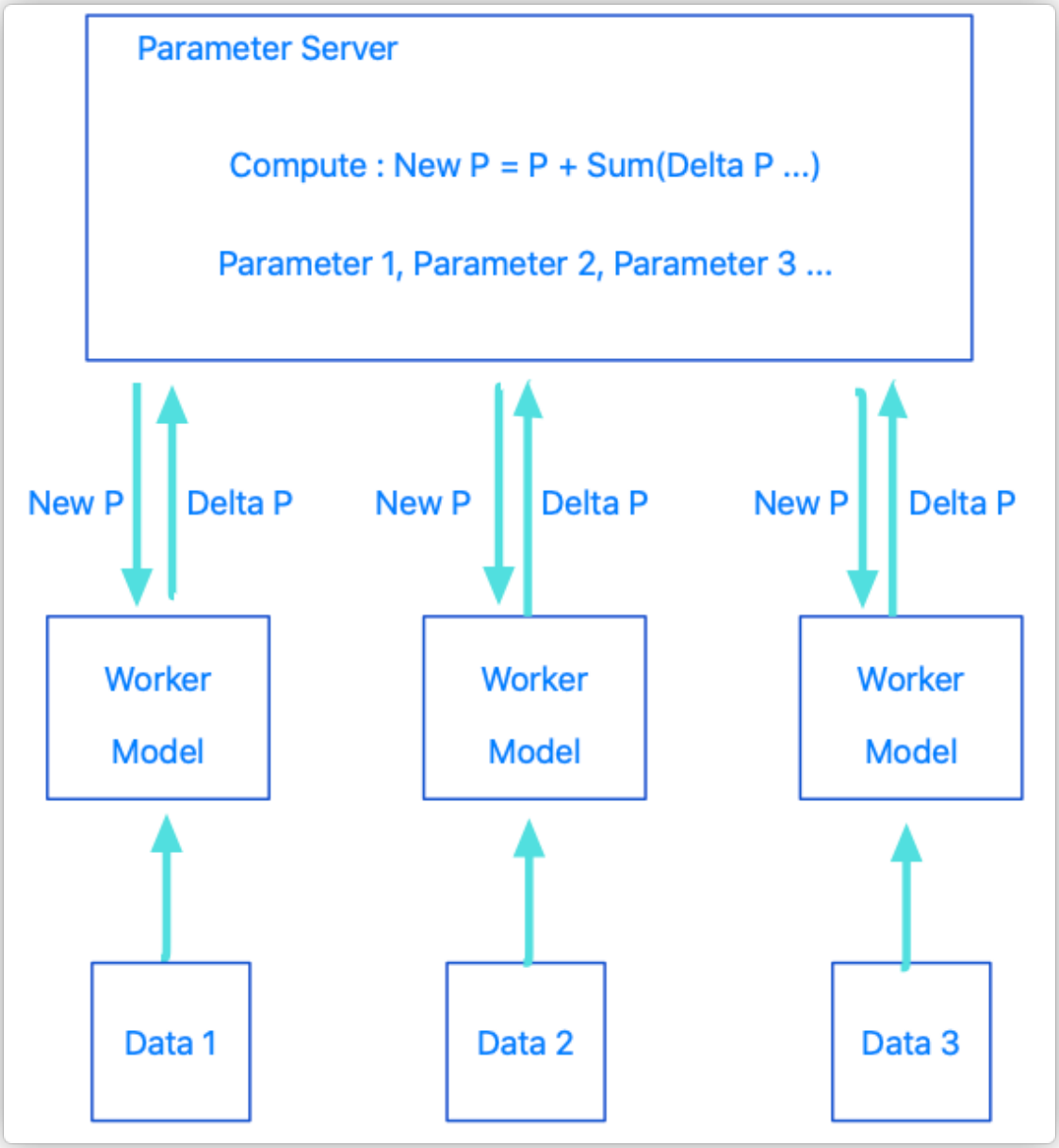
- 所有的参数都存储在参数服务器中，而 工作节点（worker） 是万年打工仔。
- 工作节点 们只负责计算梯度，待所有计算设备完成梯度计算之后，把计算好的梯度发送给参数服务器，这样参数服务器收到梯度之后，执行一定的计算（梯度平均等）之后，就更新其维护的参数，做到了在节点之间对梯度进行平均，利用平均梯度对模型进行更新。
- 然后参数服务器再把更新好的新参数返回给所有的工作节点，以对每个节点中的模型副本应用一致化更新。
- 打工仔们会再进行下一轮的前后向计算。

逻辑如下：




Delta P
New P
Worker
Model
Data 1
Delta P
New P
Worker
Model
Data 2
.....
Delta P
New P
Worker
Model
Data 3

手机如下：



参数服务器既可以用在数据并行上，也可以被用到模型并行训练上。比如可以将模型切分为多个部分，存储在不同的PS Server节点上，并提供方便的访问服务，这是参数服务器的本质。

3.3 Decentralized Network

Decentralized Network 就是去中心化网络，其特点如下：

- 去中心化网络没有一个中心节点，属于 Peer-to-Peer 架构。
- 采用 message passing 进行通信，且节点只和邻居通信。
- 并行方式可以采用异步或者同步。
- 去中心化网络的收敛情况取决于网络连接情况：
  - 连接越紧密，收敛性越快，当强连接时候，模型可以很快收敛；
  - 如果不是强连接，它可能不收敛；

0x04 All Reduce

因为本系列是 Horovod，所以我们要先说说参数服务器的劣势，下一个系列我们再说参数服务器优势。

4.1 参数服务器劣势

尽管参数服务器可以提升表现，但仍然面临几个问题：

- **确定工作者与参数服务器的正确比例**：如果使用一个参数服务器，它可能会成为网络或计算瓶颈。如果使用多个参数服务器，则通信模式变为“All-to-All”，这可能使网络饱和。
- **处理程序复杂性**：参数服务器的概念较多，这通常导致陡峭的学习曲线和大量的代码重构，压缩了实际建模的时间。
- **硬件成本**：参数服务器的引入也增加了系统的硬件成本。

人们发现，MPI_AllReduce 语义也可以很好地满足数据并行训练这一需要。

需要注意的是：AllReduce 既可以是去中心化，也可以是主从式的。

4.2 并行任务通信分类

并行任务的通信一般可以分为 Point-to-point communication 和 Collective communication。

- P2P 这种模式只有一个sender和一个receiver，实现起来比较简单，比如NV GPU Direct P2P技术服务于单机多卡的单机卡间数据通信 。
- Collective communication包含多个sender和多个receiver，一般的通信原语包括 broadcast，gather,all-gather，scatter，reduce，all-reduce，reduce-scatter，all-to-all等。

4.3 MPI_AllReduce

AllReduce（对 m 个独立参数 进行规约，并将规约结果返回给所有进程）其实是最显然和直接的分布式机器学习抽象，因为大部分算法的结构都是分布数据。在每个子集上面算出一些局部统计量，然后整合出全局统计量，并且再分配给各个节点去进行下一轮的迭代，这样一个过程就是AllReduce。

- 可以把每个 Worker 看作是 MPI 概念中的一个进程，比如可以用 4 个 Worker 组成了一个组，该组由 4 个进程组成。我们在这四个进程中对梯度进行一次 MPI_AllReduce。
- 根据 MPI_AllReduce 的语义，所有参与计算的进程都有结果，所以梯度就完成了分发。只要在初始化的时候，我们可以保证每个 Worker 的参数是一致的，那在后续的迭代计算中，参数会一直保持一致，因为梯度信息是一致的。
- AllReduce 跟 MapReduce 有类似，但后者采用的是面向通用任务处理的多阶段执行任务的方式，而AllReduce则让一个程序在必要的时候占领一台机器，并且在所有迭代的时候一直跑到底，来防止重新分配资源的开销，这更加适合于机器学习的任务处理。

所以，MPI_AllReduce 的语义可以很好地解决深度学习中梯度同步的问题。但是到底能不能使用它，还是要看下层的实现对这一场景是否足够友好。

0x05 ring-allreduce

百度提出使用新算法来平均梯度，取消 Reducer，并让这些梯度在所有节点之间交流，这被称为 ring-allreduce，他们使用 TensorFlow 也实现了这种算法（https://github.com/baidu-research/tensorflow-allreduce）。

5.1 特点

Ring-Allreduce特点如下：

- Ring Allreduce 算法使用定义良好的成对消息传递步骤序列在一组进程之间同步状态（在这种情况下为张量）。
- Ring-Allreduce 的命名中 Ring 意味着设备之间的拓扑结构为一个逻辑环形，每个设备都应该有一个左邻和一个右邻居，且本设备只会向它右邻居发送数据，并且从它的左邻居接受数据。

- Ring-Allreduce 的命名中的 Allreduce 则代表着没有中心节点，架构中的每个节点都是梯度的汇总计算节点。
- 此种算法各个节点之间只与相邻的两个节点通信，并不需要参数服务器。因此，所有节点都参与计算也参与存储，也避免产生中心化的通信瓶颈。
- 相比PS架构，Ring-Allreduce 架构是带宽优化的，因为集群中每个节点的带宽都被充分利用。
    - 在 ring-allreduce 算法中，每个 N 节点与其他两个节点进行 2 * (N-1) 次通信。在这个通信过程中，一个节点发送并接收数据缓冲区传来的块。在第一个 N - 1 迭代中，接收的值被添加到节点缓冲区中的值。在第二个 N - 1 迭代中，接收的值代替节点缓冲区中保存的值。百度的文章证明了这种算法是带宽上最优的，这意味着如果缓冲区足够大，它将最大化地利用可用的网络。
- 在深度学习训练过程中，计算梯度采用BP算法，其特点是后面层的梯度先被计算，而前面层的梯度慢于后面层，Ring-allreduce架构可以充分利用这个特点，在前面层梯度计算的同时进行后面层梯度的传递，从而进一步减少训练时间。
- Ring架构下的同步算法将参数在通信环中依次传递，往往需要多步才能完成一次参数同步。在大规模训练时会引入很大的通信开销，并且对小尺寸张量（tensor）不够友好。对于小尺寸张量，可以采用批量操作（batch）的方法来减小通信开销。

综上所述，Ring-based AllReduce 架构的网络通讯量如果处理适当，不会随着机器增加而增加，而仅仅和模型 & 网络带宽有关，这针对参数服务器是个巨大的提升。

5.2 策略

Ring-based AllReduce 策略包括 Scatter-Reduce 和 AllGather 两个阶段。

- 首先是scatter-reduce，scatter-reduce 会逐步交换彼此的梯度并融合，最后每个 GPU 都会包含完整融合梯度的一部分，是最终结果的一个块。

  假设环中有 N 个 worker，每个 worker 有长度相同的数组，需要将 worker 的数组进行求和。在 Scatter-Reduce 阶段，每个 worker 会将数组分成 N 份数据块，然后 worker 之间进行 N 次数据交换。在第 k 次数据交换时，第 i 个 worker 会将自己的 (i - k) % N 份数据块发送给下一个 worker。接收到上一个 worker 的数据块后，worker 会将其与自己对应的数据块求和。

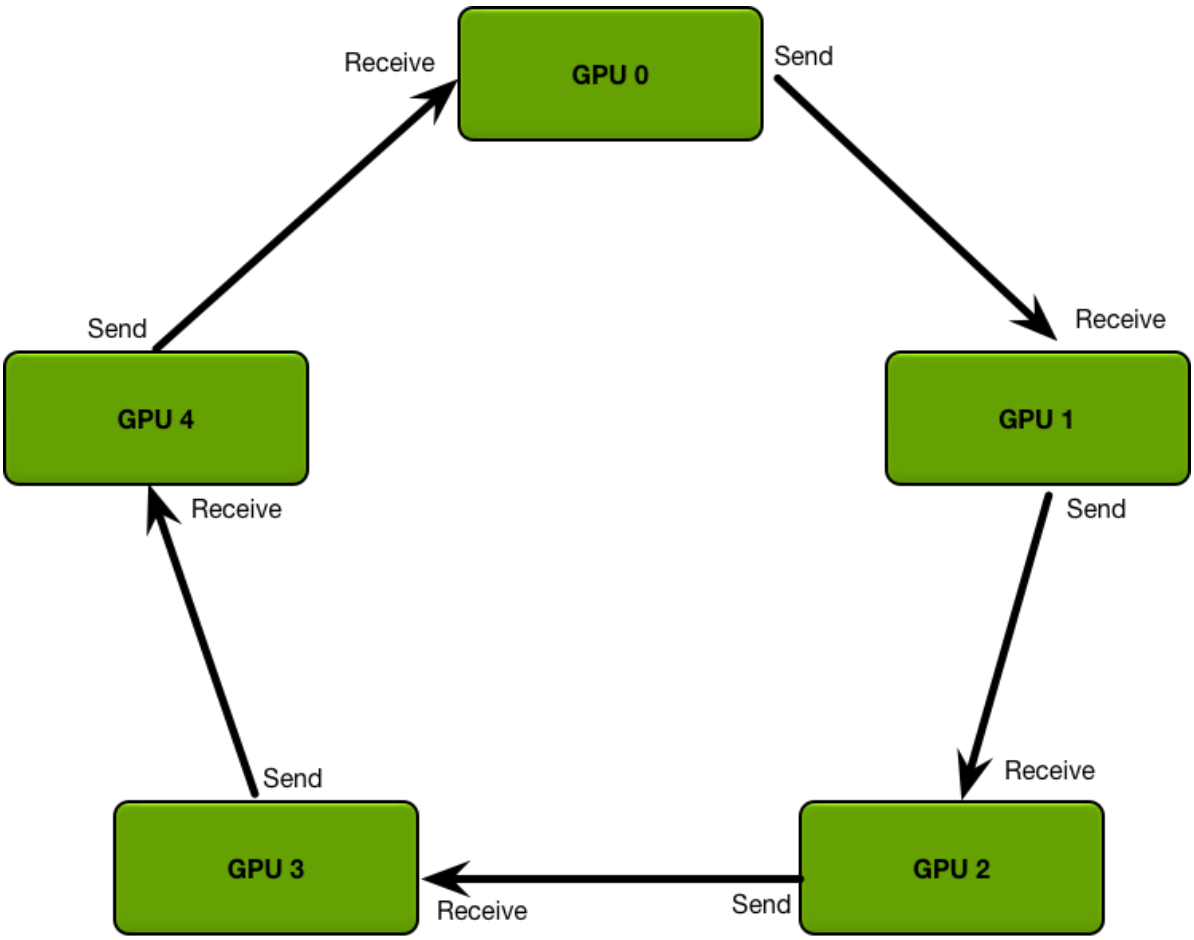
- 然后是allgather。GPU 会逐步交换彼此不完整的融合梯度，最后所有 GPU 都会得到完整的最终融合梯度。

  在执行完 Scatter-Reduce 后，每个 worker 的数组里都有某个数据块是最终求和的结果，现在需要将各数据块的最后求和结果发送到每个 worker 上。和 Scatter-Reduce 一样，也需要 N 次循环。在第 k 次循环时，第 i 个 worker 会将其第 (i+1-k)%N 个数据块发送给下一个 worker 。接收到前一个 worker 的数据块后，worker 会用接收的数据快覆盖自己对应的数据块。进行 N 次循环后，每个 worker 就拥有了数组各数据块的最终求和结果了。

以下部分来自 https://andrew.gibiansky.com/blog/machine-learning/baidu-allreduce/，这是我能找到最优秀的解读。

5.2.1 结构

环形结构如下，每个 GPU 应该有一个左邻居和一个右邻居；它只会向其右侧邻居发送数据，并从其左侧邻居接收数据。：



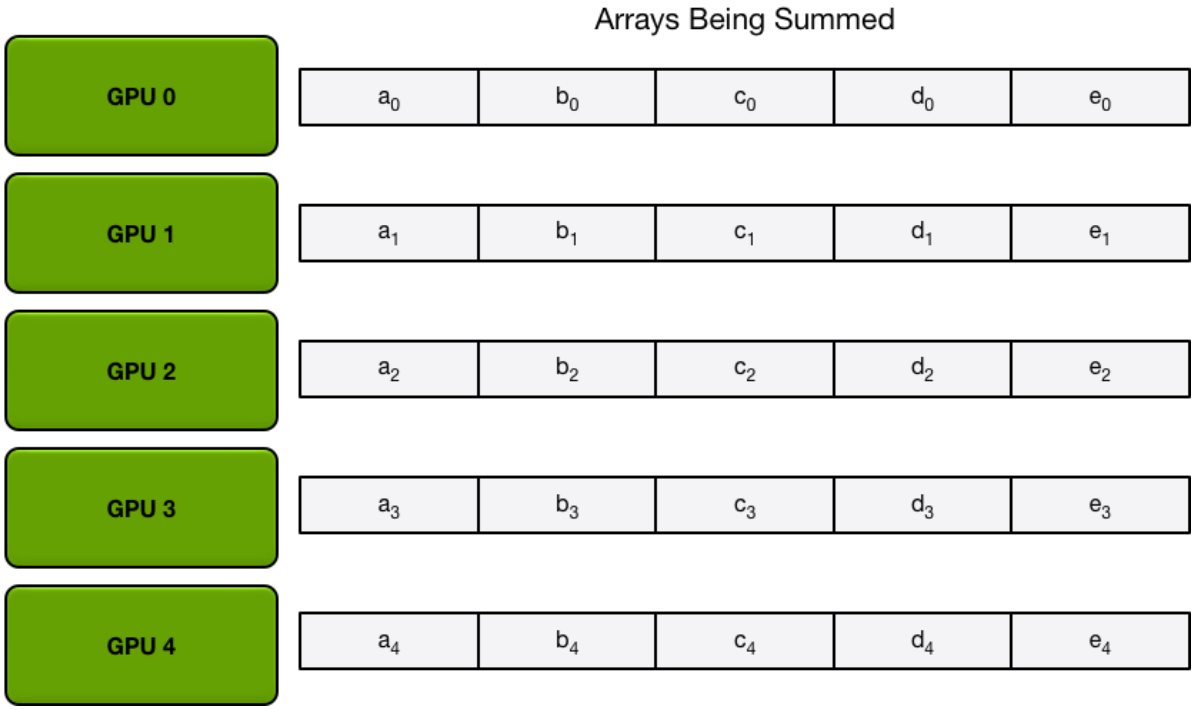
5.2.2 Scatter-Reduce

scatter-reduce：会逐步交换彼此的梯度并融合，最后每个 GPU 都会包含完整融合梯度的一部分。

为简单起见，我们假设目标是按元素对单个大型浮点数数组的所有元素求和；系统中有 N 个 GPU，每个 GPU 都有一个相同大小的数组，在 allreduce 的最后环节，每个 GPU 都应该有一个相同大小的数组，其中包含原始数组中数字的总和。

5.2.2.1 分块

首先，GPU 将阵列划分为 N 个较小的块（其中 N 是环中的 GPU 数量）。



接下来，GPU 将进行 N-1 次 scatter-reduce 迭代。

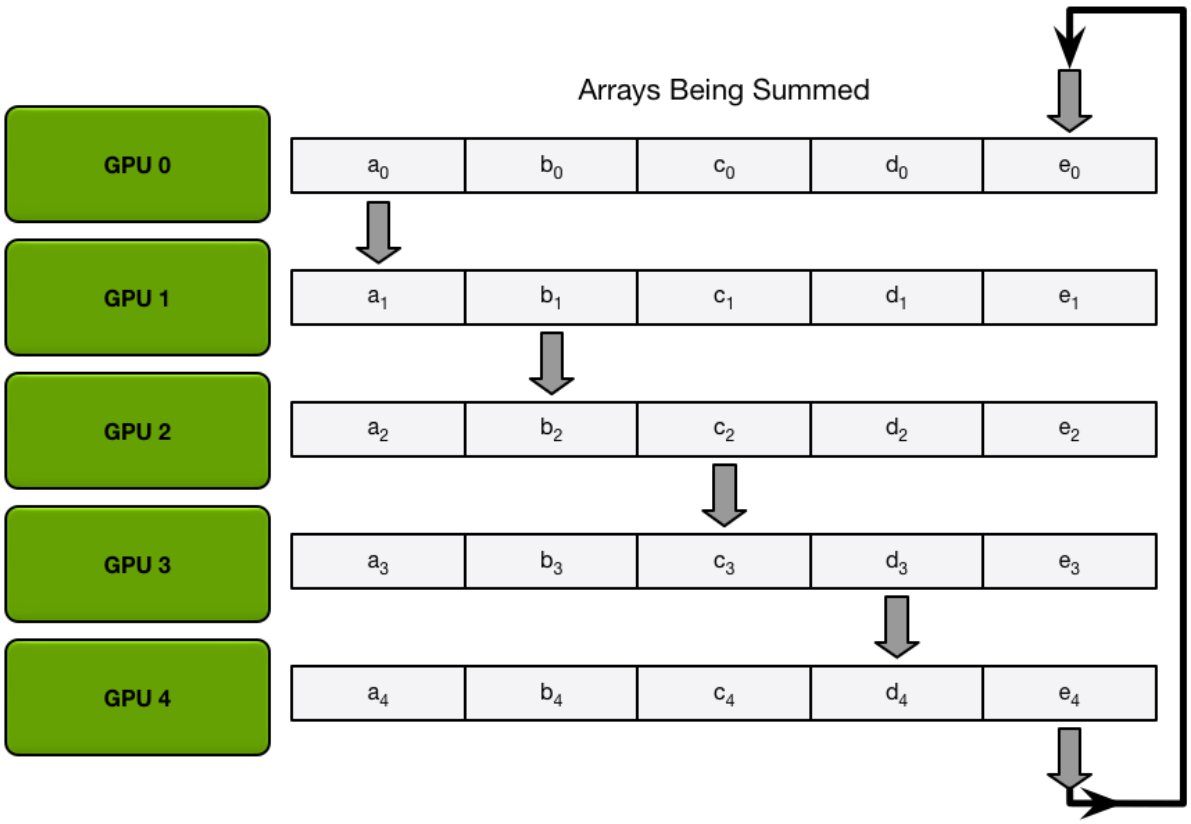
在每次迭代中，GPU 会将其一个块发送到其右邻居，并将从其左邻居接收一个块并累积到该块中。每个 GPU 发送和接收的数据块每次迭代都不同。第 n 个 GPU 通过发送块 n 和接收块 n – 1 开始，然后逐步向后进行，每次迭代发送它在前一次迭代中接收到的块。

5.2.2.2 第一次迭代

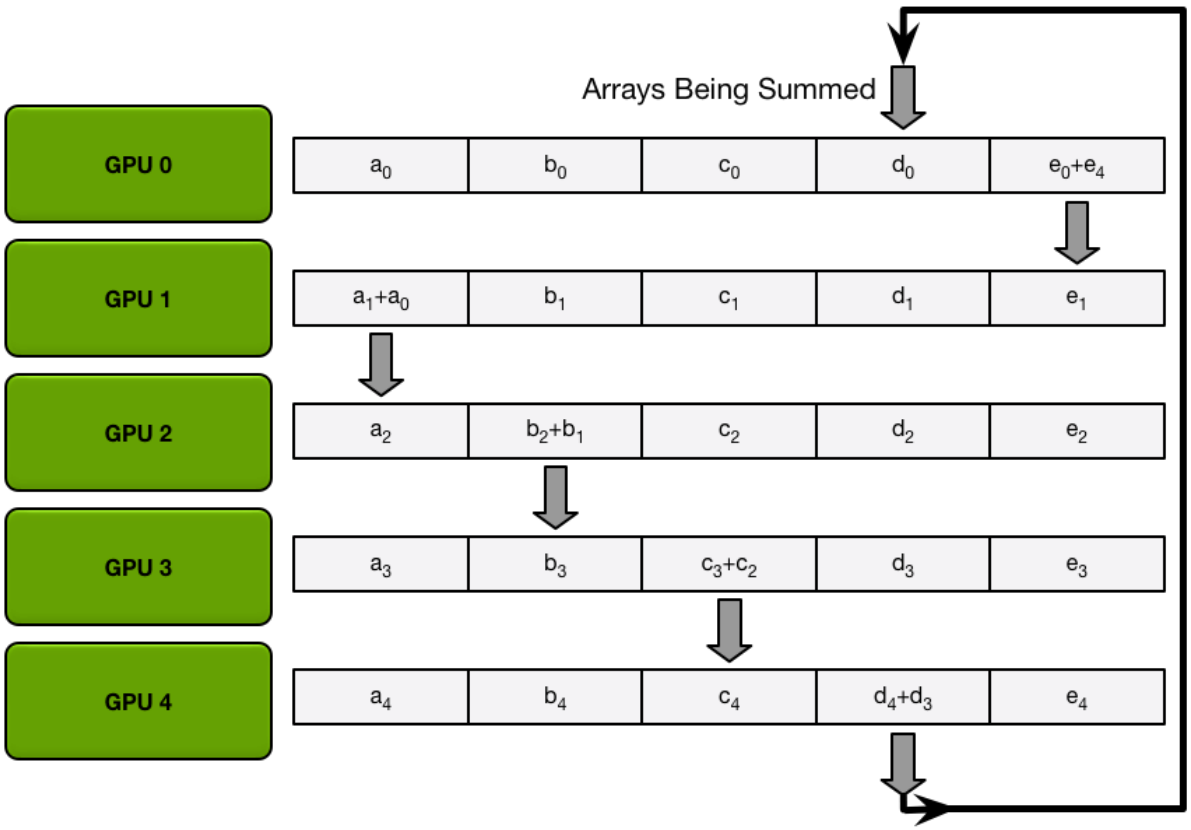
在第一次迭代中，上图中的五个 GPU 将发送和接收以下块：

GPU	发送	收到
0	块 0	块 4
1	块 1	块 0
2	块 2	块 1
3	块 3	块 2
4	块 4	块 3

scatter-reduce 的第一次迭代中的数据传输如下：



第一次发送和接收完成后，每个 GPU 都会有一个块，该块由两个不同 GPU 上相同块的总和组成。例如，第二个 GPU 上的第一个块将是该块中来自第二个 GPU 和第一个 GPU 的值的总和。

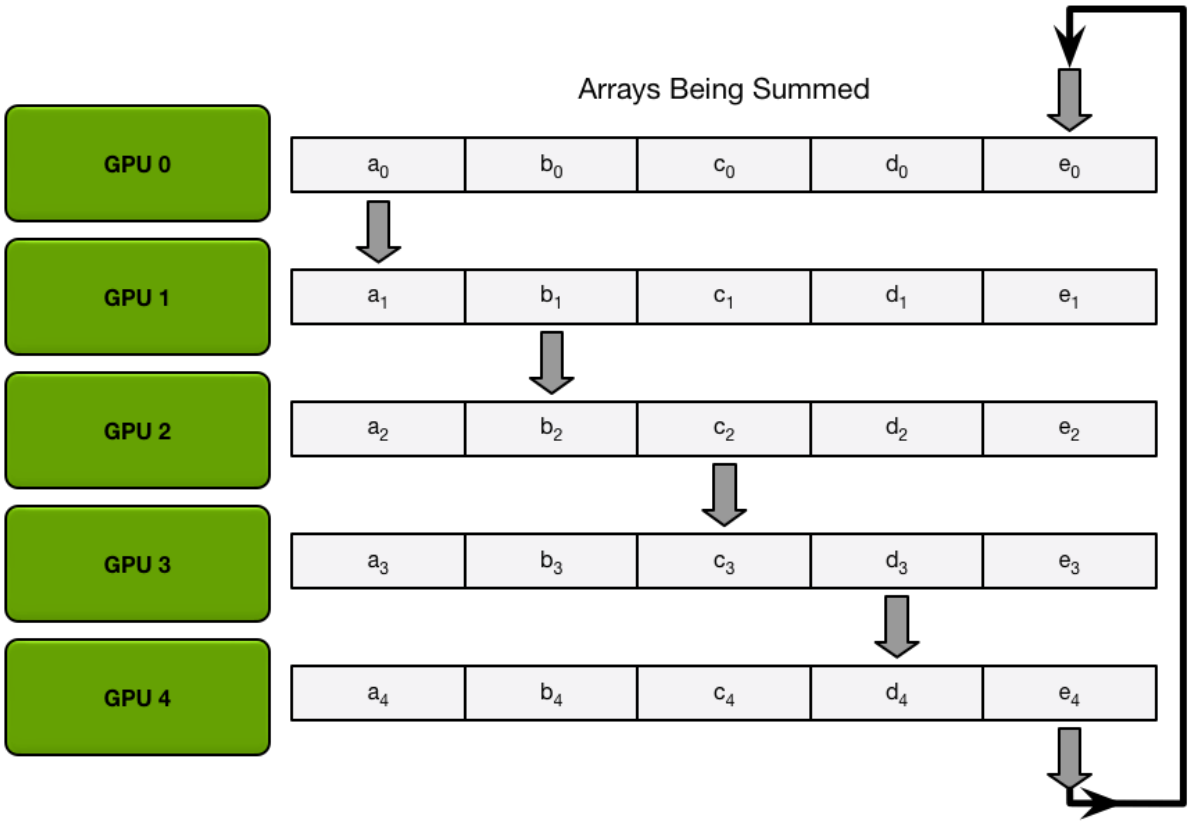


5.2.2.3 全部迭代

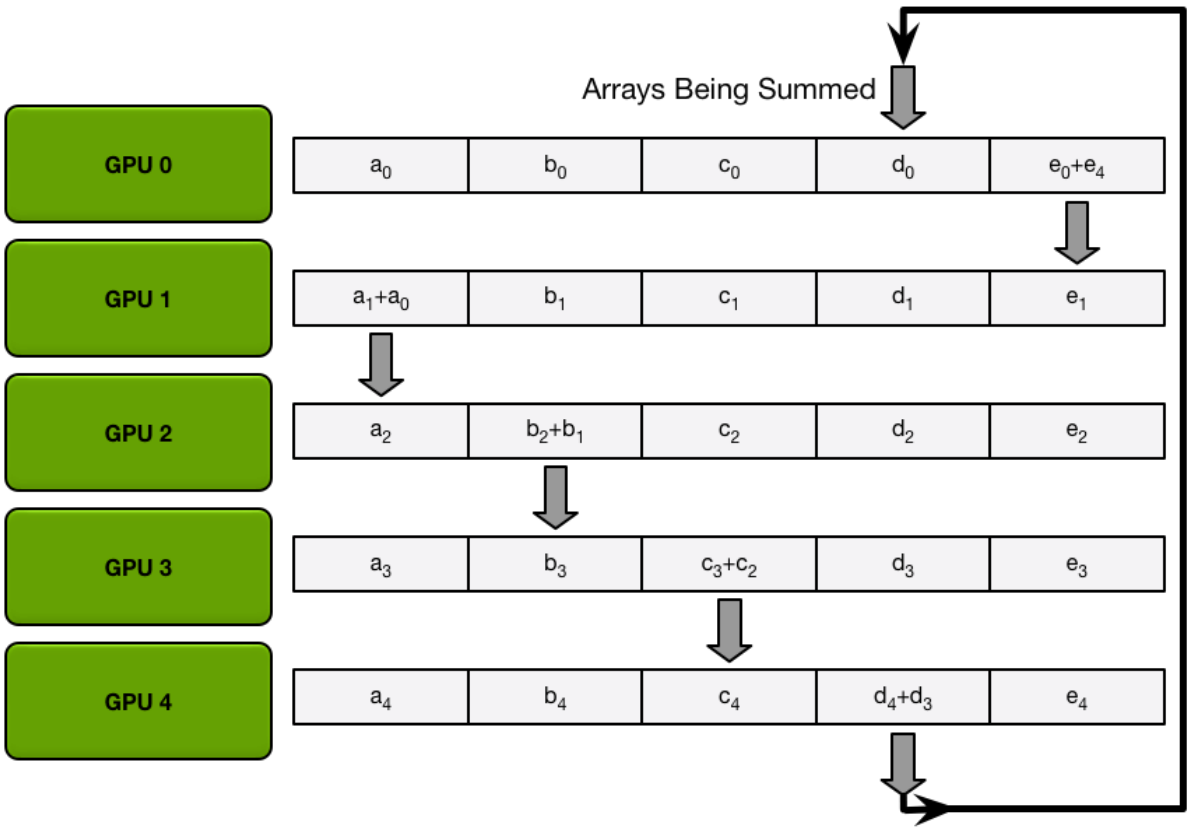
在后续迭代中，该过程继续直到最后。最终每个 GPU 将有一个块，这个块包含所有 GPU 中该块中所有值的总和。

下面系列图展示了所有数据传输和中间结果，从第一次迭代开始，一直持续到scatter-reduce完成。

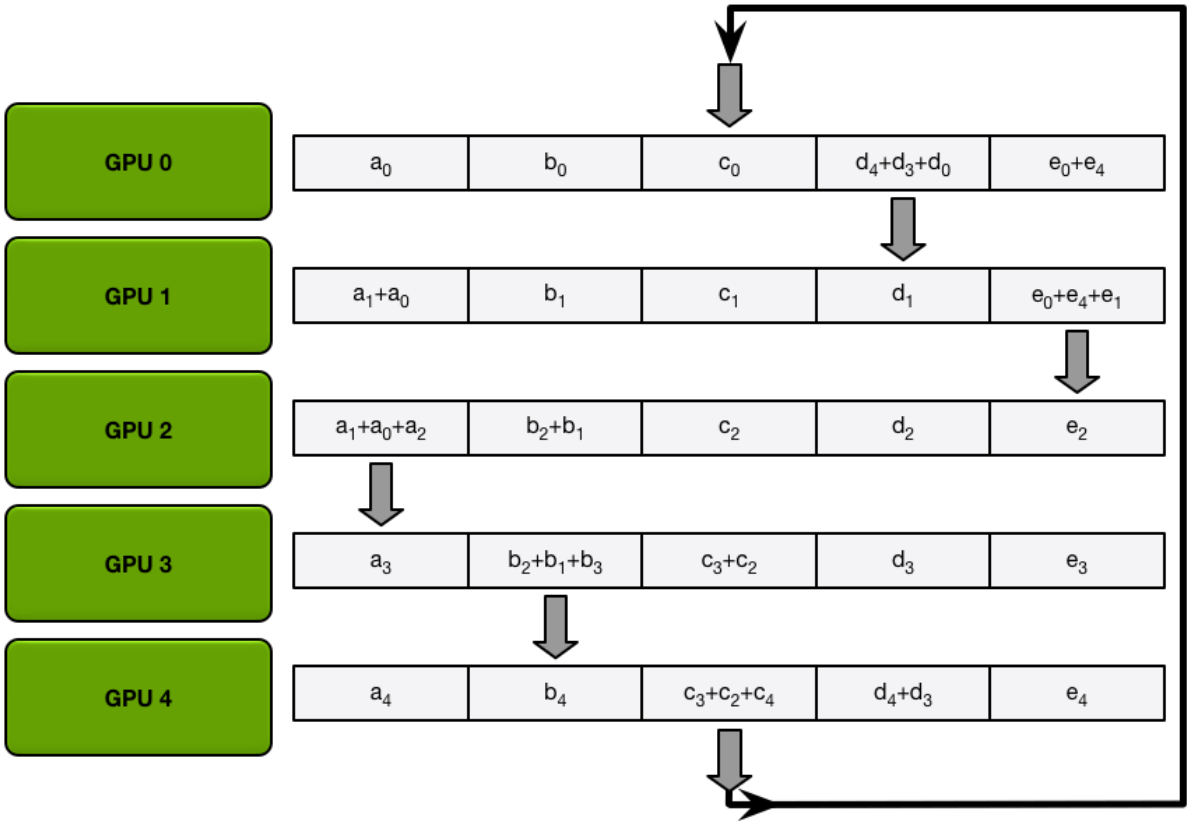
第一次迭代



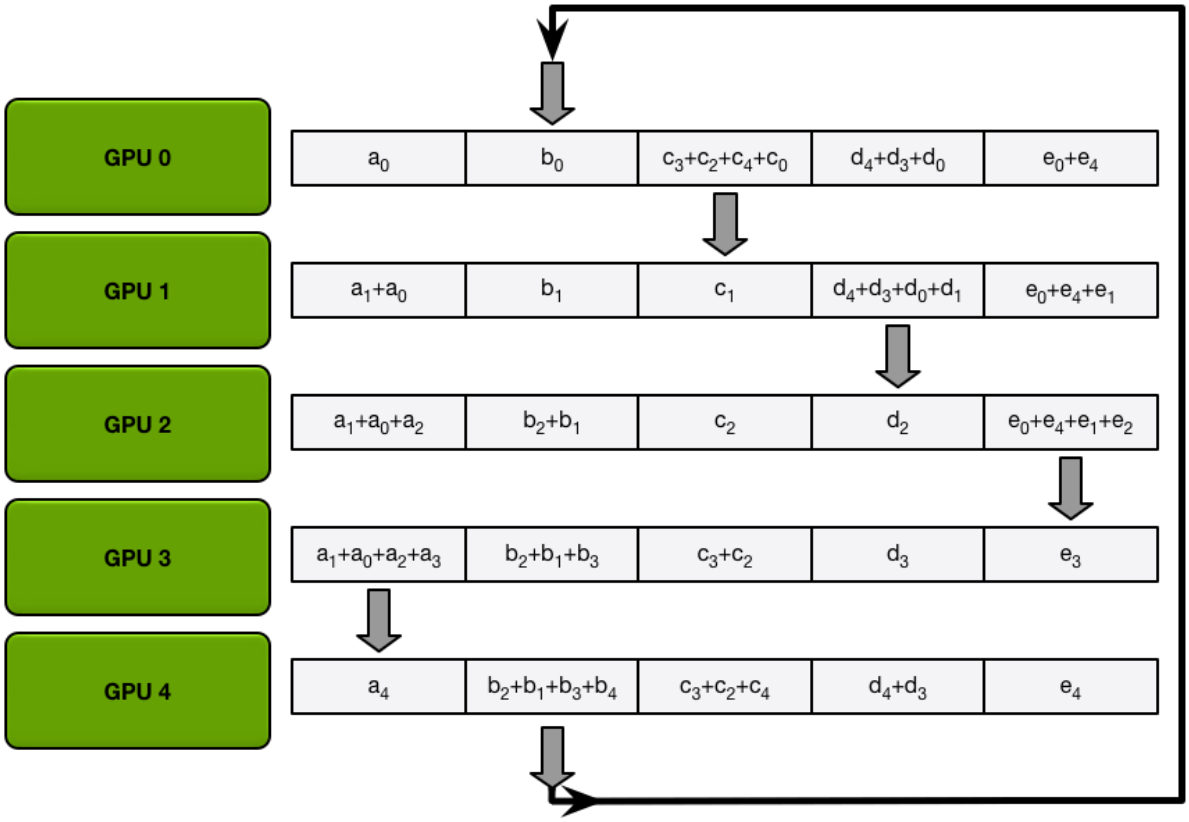
第二次迭代


Arrays Being Summed
GPU 0
GPU 1
GPU 2
GPU 3
GPU 4
a0 b0 c0 d0 e0+e4
a1+a0 b1 c1 d1 e1
a2 b2+b1 c2 d2 e2
a3 b3 c3+c2 d3 e3
a4 b4 c4 d4+d3 e4

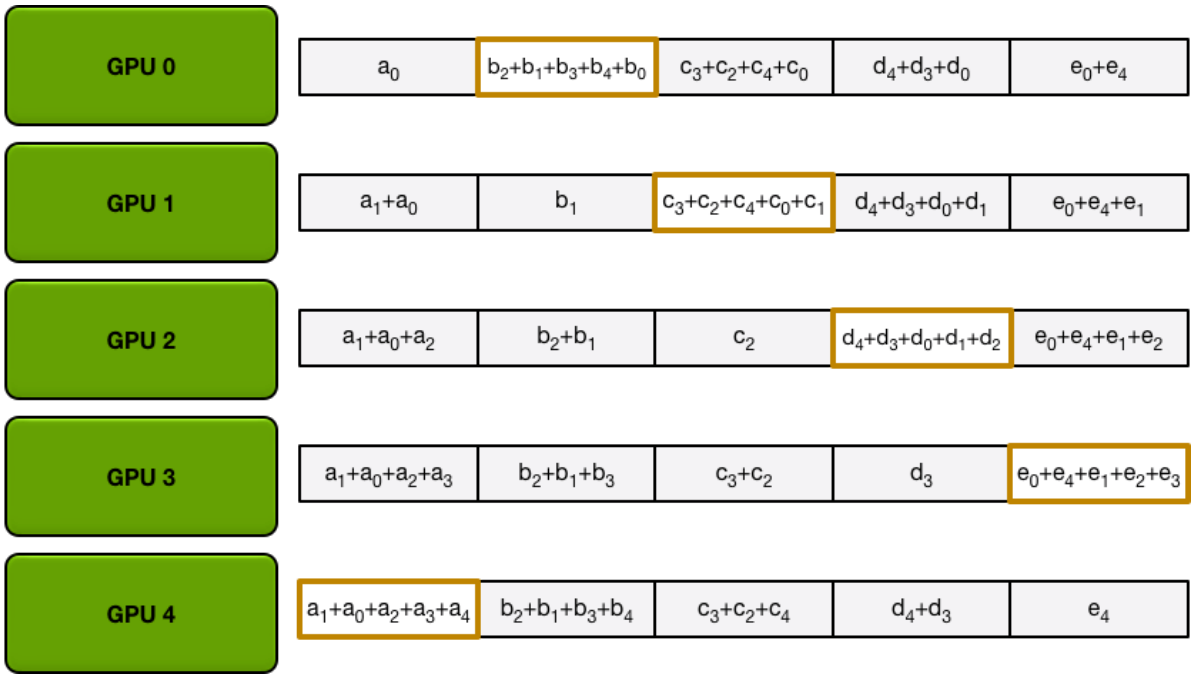
第三次迭代


GPU 0
GPU 1
GPU 2
GPU 3
GPU 4
a0 b0 c0 d4+d3+d0 e0+e4
a1+a0 b1 c1 d1 e0+e4+e1
a1+a0+a2 b2+b1 c2 d2 e2
a3 b2+b1+b3 c3+c2 d3 e3
a4 b4 c3+c2+c4 d4+d3 e4

第四次迭代



所有 scatter-reduce 传输后的最终状态



5.2.3 Allgather

在 scatter-reduce 步骤完成后，在每个 GPU 的数组中都有某一些值（每个 GPU 有一个块）是最终值，其中包括来自所有 GPU 的贡献。为了完成 allreduce，GPU 必须接下来交换这些块，以便所有 GPU 都具有最终所需的值。

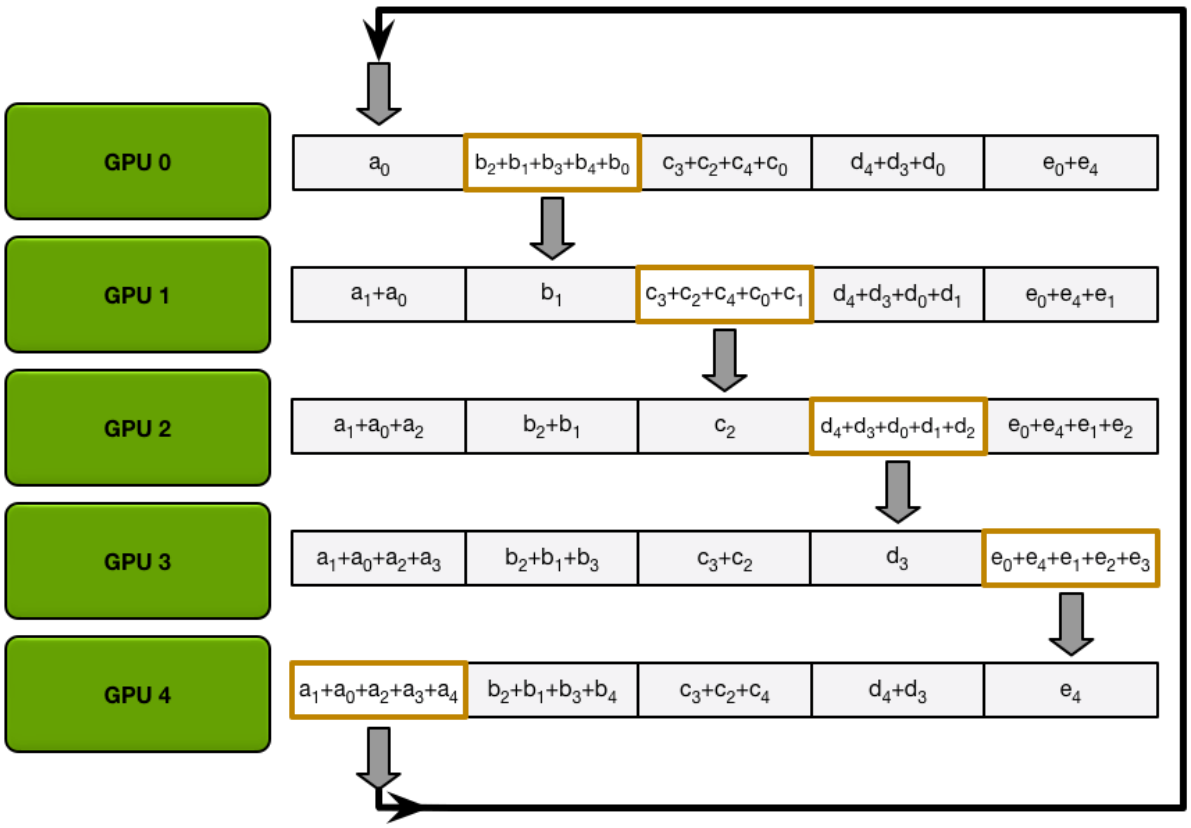
ring allgather 与 scatter-reduce 进行相同的处理（发送和接收的 N-1 次迭代），但是他们这次不是累积 GPU 接收的值，而只是简单地覆盖块。第 n 个 GPU 开始发送第 n+1 个块并接收第 n 个块，然后在以后的迭代中始终发送它刚刚接收到的块。

5.2.3.1 第一次迭代

例如，在我们的 5-GPU 设置的第一次迭代中，GPU 将发送和接收以下块：

图形处理器	发送	收到
0	块 1	块 0
1	块 2	块 1
2	块 3	块 2
3	块 4	块 3
4	块 0	块 4

allgather 的第一次迭代中的数据传输如下。

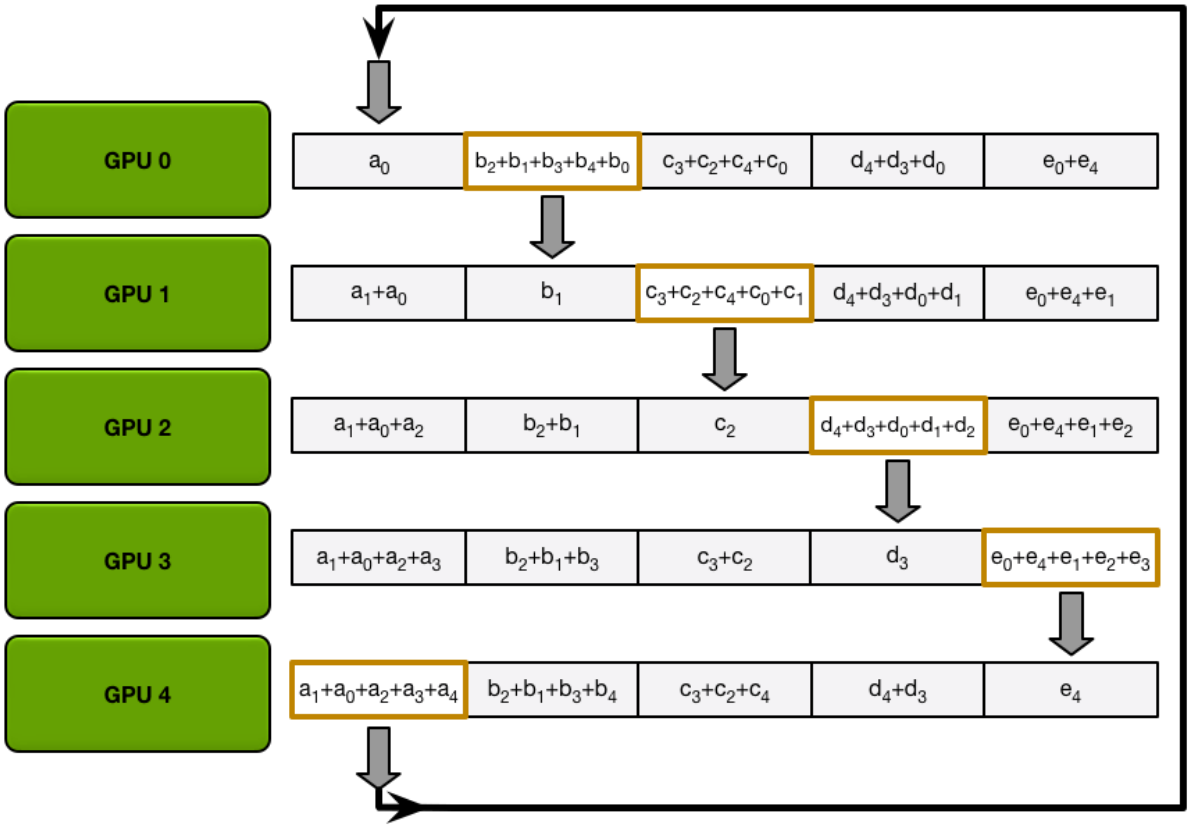


第一次迭代完成后，每个 GPU 都会有最终数组的两个块。在接下来的迭代中，该过程继续一直到最后，最终每个 GPU 将拥有整个数组的完全累加值。

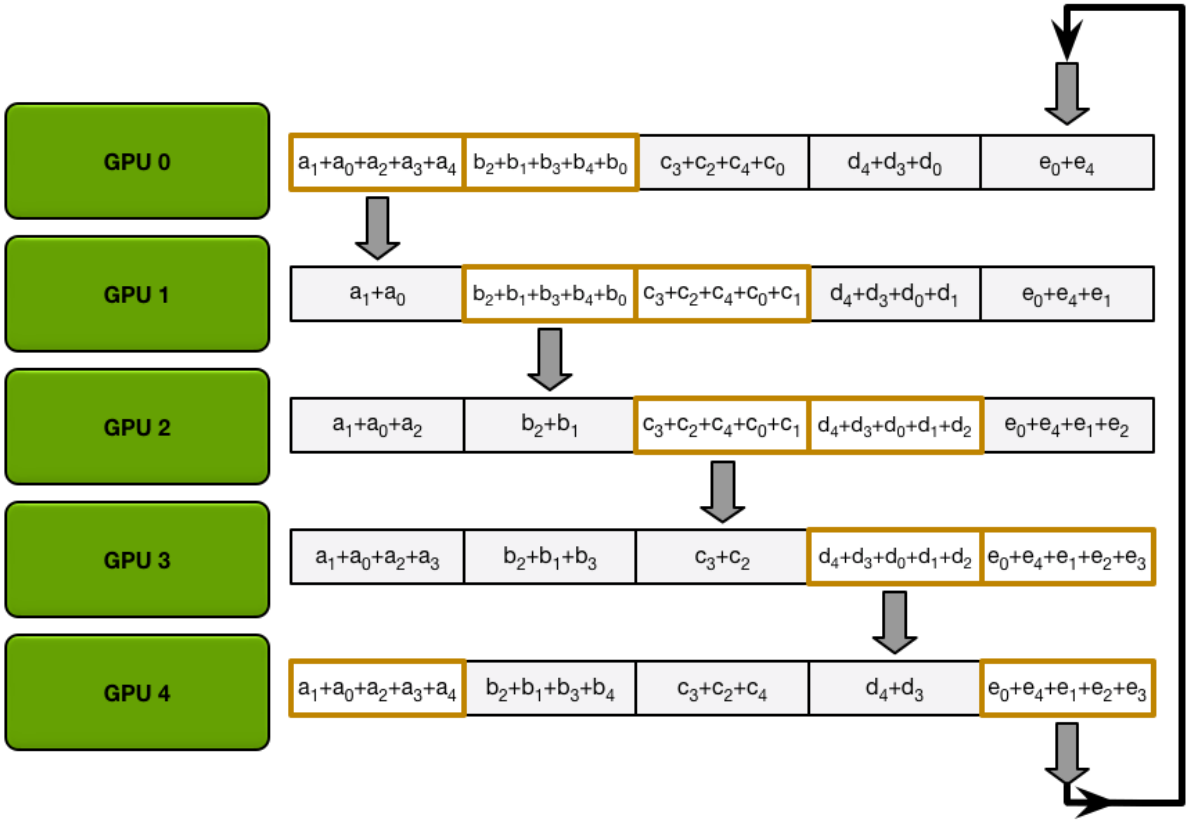
5.2.3.2 全部迭代

下面系列图展示了所有数据传输和中间结果，从第一次迭代开始，一直持续到全部收集完成。

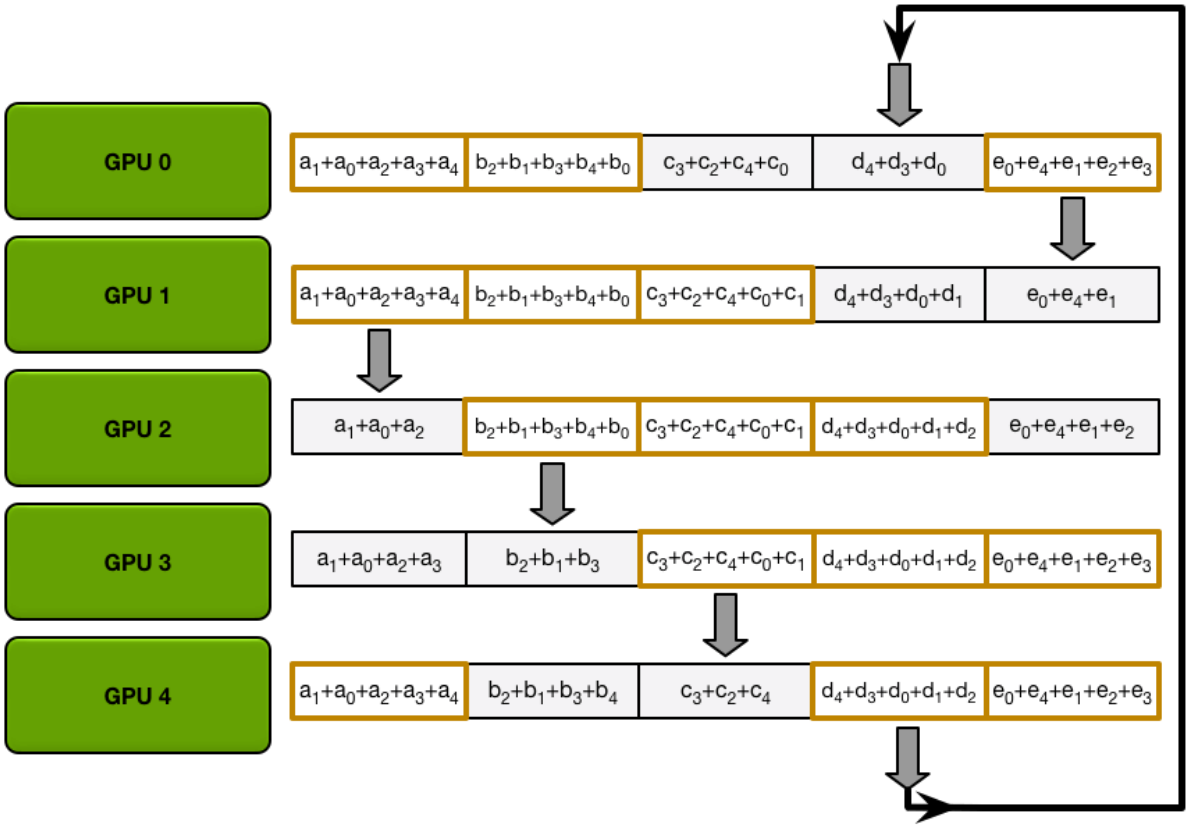
Allgather 数据传输（迭代 1）



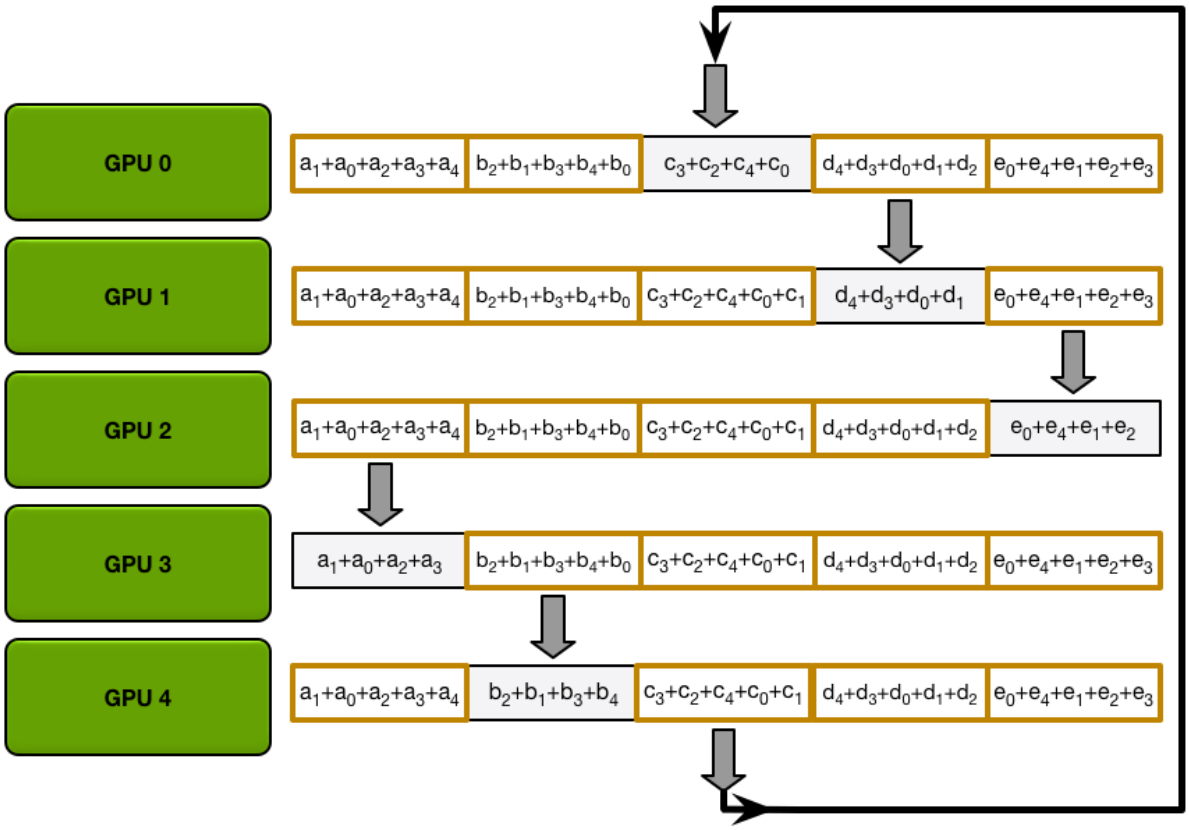
Allgather 数据传输（迭代 2）如下：



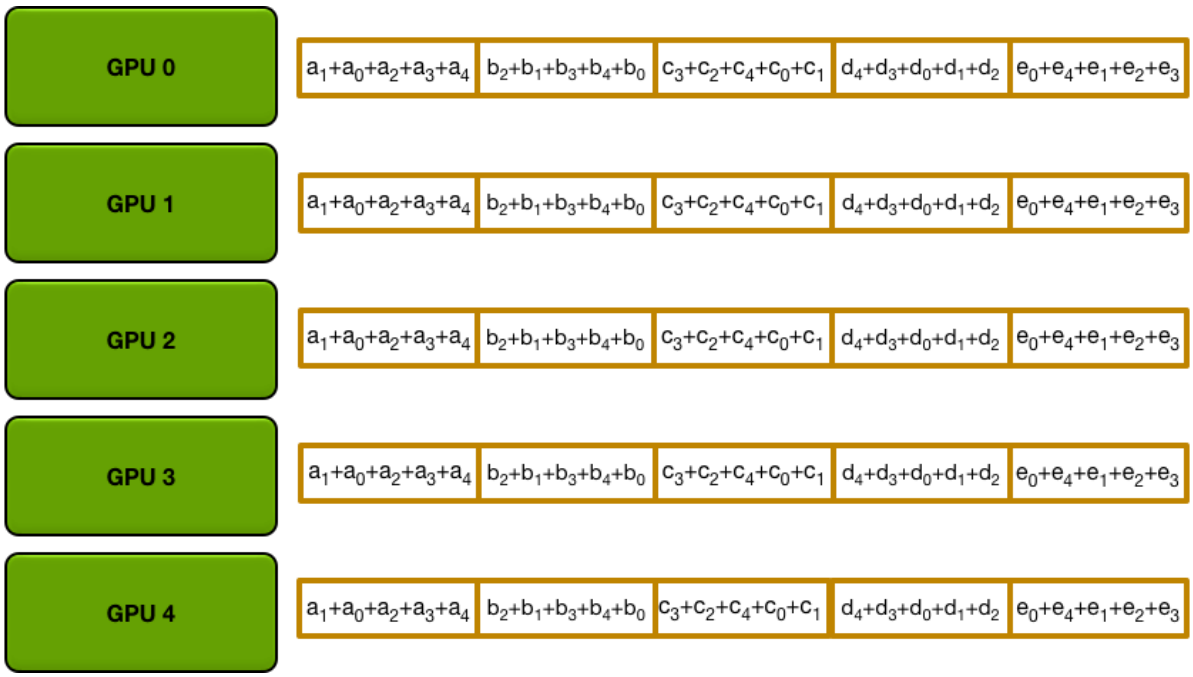
Allgather 数据传输（迭代 3）



Allgather 数据传输（迭代 4）

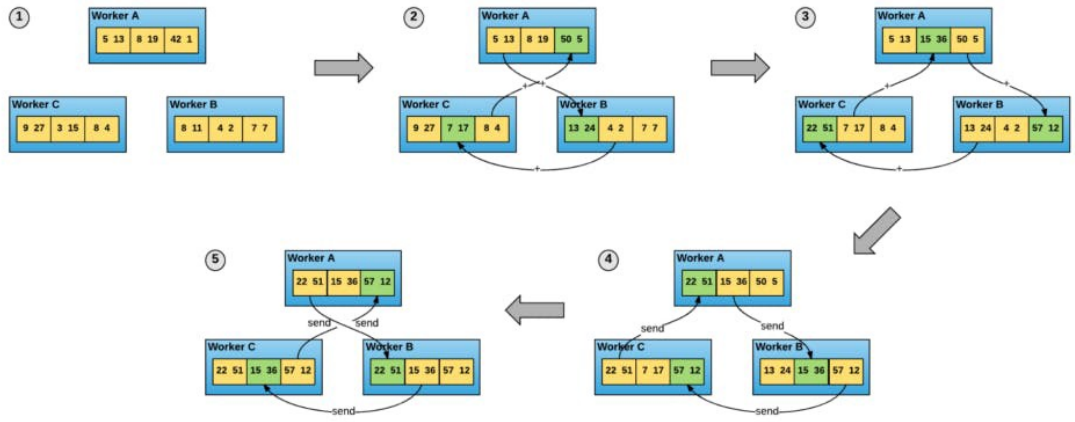


所有全部转移后的最终状态。



5.2.4 Horovod 架构图

工作原理也可以借助Horovod的发布帖子 来看看。



5.2.5 百度思路

或者我们从百度的源码中也可以直接看到思路，现在摘录给大家。

具体代码参见 https://github.com/baidu-research/tensorflow-allreduce/commit/66d5b855e90b0949e9fa5cca5599fd729a70e874#diff-3d530d590e551619acd776cfe7eaff06R517

tensorflow/contrib/mpi_collectives/ring.h

```
/* Perform a ring allreduce on the data. Allocate the necessary output tensor
and
 * store it in the output parameter.
 *
 * Assumes that all MPI processes are doing an allreduce of the same tensor,
 * with the same dimensions.
 *
 * A ring allreduce is a bandwidth-optimal way to do an allreduce. To do the
allreduce,
 * the nodes involved are arranged in a ring:
 *
 *                   .--0--.
```

```
 *                  /       \
 *                 3         1
 *                  \       /
 *                   *--2--*
 *
 *  Each node always sends to the next clockwise node in the ring, and receives
 *  from the previous one.
 *
 *  The allreduce is done in two parts: a scatter-reduce and an allgather. In
 *  the scatter reduce, a reduction is done, so that each node ends up with a
 *  chunk of the final output tensor which has contributions from all other
 *  nodes.  In the allgather, those chunks are distributed among all the nodes,
 *  so that all nodes have the entire output tensor.
 *
 *  Both of these operations are done by dividing the input tensor into N
 *  evenly sized chunks (where N is the number of nodes in the ring).
 *
 *  The scatter-reduce is done in N-1 steps. In the ith step, node j will send
 *  the (j - i)th chunk and receive the (j - i - 1)th chunk, adding it in to
 *  its existing data for that chunk. For example, in the first iteration with
 *  the ring depicted above, you will have the following transfers:
 *
 *      Segment 0:  Node 0 --> Node 1
 *      Segment 1:  Node 1 --> Node 2
 *      Segment 2:  Node 2 --> Node 3
 *      Segment 3:  Node 3 --> Node 0
 *
 *  In the second iteration, you'll have the following transfers:
 *
 *      Segment 0:  Node 1 --> Node 2
 *      Segment 1:  Node 2 --> Node 3
 *      Segment 2:  Node 3 --> Node 0
 *      Segment 3:  Node 0 --> Node 1
 *
 *  After this iteration, Node 2 has 3 of the four contributions to Segment 0.
 *  The last iteration has the following transfers:
 *
 *      Segment 0:  Node 2 --> Node 3
 *      Segment 1:  Node 3 --> Node 0
 *      Segment 2:  Node 0 --> Node 1
 *      Segment 3:  Node 1 --> Node 2
 *
 *  After this iteration, Node 3 has the fully accumulated Segment 0; Node 0
 *  has the fully accumulated Segment 1; and so on. The scatter-reduce is
complete.
 *
 *  Next, the allgather distributes these fully accumululated chunks across all
nodes.
 *  Communication proceeds in the same ring, once again in N-1 steps. At the ith
step,
 *  node j will send chunk (j - i + 1) and receive chunk (j - i). For example,
at the
 *  first iteration, the following transfers will occur:
 *
 *      Segment 0:  Node 3 --> Node 0
 *      Segment 1:  Node 0 --> Node 1
 *      Segment 2:  Node 1 --> Node 2
 *      Segment 3:  Node 2 --> Node 3
```

```
 *
 * After the first iteration, Node 0 will have a fully accumulated Segment 0
 * (from Node 3) and Segment 1. In the next iteration, Node 0 will send its
 * just-received Segment 0 onward to Node 1, and receive Segment 3 from Node 3.
 * After this has continued for N - 1 iterations, all nodes will have a the
fully
 * accumulated tensor.
 *
 * Each node will do (N-1) sends for the scatter-reduce and (N-1) sends for the
allgather.
 * Each send will contain K / N bytes, if there are K bytes in the original
tensor on every node.
 * Thus, each node sends and receives 2K(N - 1)/N bytes of data, and the
performance of the allreduce
 * (assuming no latency in connections) is constrained by the slowest
interconnect between the nodes.
 *
 */
```

5.3 区别

在中等规模模型情况下，all-reduce 更适合。当规模巨大时候则应该使用参数服务器。

参数服务器 适合的是高纬稀疏模型训练，它利用的是维度稀疏的特点，每次 pull or push 只更新有效的值。但是深度学习模型是典型的dense场景，embedding做的就是把稀疏变成稠密。所以这种 pull or push 的不太适合。而 网络通信上更优化的 all-reduce 适合中等规模的深度学习。

又比如由于推荐搜索领域模型的 Embedding 层规模庞大以及训练数据样本长度不固定等原因，导致容易出现显存不足和卡间同步时间耗费等问题，所以 all-reduce 架构很少被用于搜索推荐领域。

至此，背景知识已经介绍完毕，下一篇我们开始介绍 Horovod 的使用。