推荐阅读：

- Parameter Server 详解

server 节点可以跟其他 server 节点通信，每个server负责自己分到的参数，server group 共同维持所有参数的更新。 **server manager** node 负责维护一些元数据的一致性，比如各个节点的状态，参数的分配情况等； **worker** 节点之间没有通信，只跟自己对应的server进行通信。每个worker group有一个**task scheduler**，负责向worker分配任务，并且监控worker的运行情况。当有新的worker加入或者退出，task scheduler 负责重新分配任务。

parameter server 中，参数都是可以被表示成(key, value)的集合，比如一个最小化损失函数的问题，key就是feature ID，而value就是它的权值。对于稀疏参数，不存在的key，就可以认为是0.

把参数表示成k-v， 形式更自然， 易于理，更易于编程解；

workers 跟 servers 之间通过 **push** 跟 **pull** 来通信。worker 通过 push 将计算好的梯度发送到 server，然后通过 pull 从server更新参数。为了提高计算性能和带宽效率，parameter server 允许用户使用 **Range Push** 跟 **Range Pull** 操作；

假设 R 是需要push或pull的 key 的range，那么可以进行如下操作：

```
w.push(R, dest)
w.pull(R, dest)
```

Implementation

servers 使用 一致性hash 来存储参数k-v键值对。用链式复制方式来提高系统容错性。使用 range based communication 和 range based vector clocks 来进一步优化系统；

Vector Clock

parameter server 使用 **vector clock** 来记录每个节点中参数的时间戳，能够用来跟踪状态或避免数据的重复发送。但是，假设有n个节点，m个参数，那么vector clock的空间复杂度就是O(n∗m). 显然，当有几千个节点和几十亿的参数时，对于内存和带宽来说都是不可实现的。

好在，parameter server 在push跟pull的时候，都是rang-based，这就带来了一个好处：这个range里面的参数共享的是同一个时间戳，这显然可以大大降低了空间复杂度。

每次从一个range里再提取一个range，最多会生成3个新的 vector clocks（一分为三） . 假设k是算法中产生的所有的range，那么空间复杂度就变成了O(k∗m)，因为k 远小于参数个数，所以空间复杂度大大降低了；

Messages

一条 message 包括：时间戳，len(range)对k-v.

[vc(R),(k1,v1),...,(kp,vp)]kj∈Randj∈{1,...p}

这是parameter server 中最基本的通信格式，不仅仅是共享的参数才有，task 的message也是这样的格式，只要把这里的(key, value) 改成 (task ID, 参数/返回值)。

由于机器学习问题通常都需要很高的网络带宽，因此信息的压缩是必须的。

key的压缩： 因为训练数据通常在分配之后都不会发生改变，因此worker没有必要每次都发送相同的key，只需要接收方在第一次接收的时候缓存起来就行了。第二次，worker不再需要同时发送key和value，只需要发送value 和 key list的hash就行。这样瞬间减少了一般的通信量。 **value的压缩：** 假设参数时稀疏的，那么就会有大量的0存在。因此，为了进一步压缩，我们只需要发送非0值。parameter server使用 **Snappy** 快速压缩库来压缩数据、高效去除0值。

> **key** 的压缩和 **value** 的压缩可以同时进行。

用户自定义过滤： 对于机器学习优化问题比如梯度下降来说，并不是每次计算的梯度对于最终优化都是有价值的，用户可以通过自定义的规则过滤一些不必要的传送，再进一步压缩带宽cost：

1. **发送很小的梯度值是低效的：** 因此可以自定义设置，只在梯度值较大的时候发送；
2. **更新接近最优情况的值是低效的：** 因此，只在非最优的情况下发送，可通过KKT来判断；

Replication and Consistency

parameter server 在数据一致性上，使用的是传统的**一致性哈希**算法，参数key与server node id被插入到一个hash ring中。

具体的一致性hash算法不是本文的重点，这里不过多介绍了，不清楚的同学建议看看其他文献熟悉一下。 只要知道它的作用是在分布式系统中，动态增加和移除节点的同时还能保证系统存储与key分配的性能效率；

从上图可以看出，每个节点都复制了它逆时钟方向的k个节点中的key。图中，k=2，S1 赋值了 S2 和 S3 内的key。

两种方式保证slave跟master之间的数据一致性：

1.默认的复制方式: **Chain replication** (强一致性, 可靠)：

a. **更新**：只能发生在数据头节点,然后更新逐步后移，直到更新到达尾节点，并由尾节点向客户确认更新成功； b. **查询**：为保证强一致性，客户查询只能在尾节点进行；

2.Replication after **Aggregation**：

两个worker 节点分别向server传送x和y。server 首先通过一定方式（如：f(x+y) ）进行aggregate，然后再进行复制操作；

当有n个worker的时候，复制只需要k/n的带宽。通常来说，k（复制次数）是一个很小的常数，而n的值大概是几百到几千；

Server Management

要想实现系统的容错以及动态的扩展系统规模，必须要求系统能够支持动态添加和移除节点。

> **当有一个 server节点添加 进来的时候会发生什么呢？** \1. server manager 会对新的节点分配一些range 的key，这会造成其他server节点的key的变化； \2. 新节点会获取数据做为训练用，另外会复制k份到slave。 \3. server manager 将节点的变化情况广播出去。接收方可能会移除不再属于自己的数据，并且将未完成的任务提交给新来的节点；
>
> **当有一个 worker节点（W）添加 进来的时候会发生什么呢？** 跟server差不多，相对更简单一些： \1. task scheduler 为W分配数据； \2. 这个 worker 节点通过网络或者文件系统得到分配到的训练数据。接着，W会从服务器pull参数； \3. task scheduler 会广播节点的变化情况，可能会使一些节点释放一部分训练数据；