# 并行计算 实验四

PB19071501 李平治

## I. 问题描述

使用参数服务器（Parameter Server）架构训练一个机器学习模型，对MNIST手写数字图片进行分类。自行选择参数服务器的同步模式。自行选择机器学习模型以及训练算法。使用MPI实现。

> 可以使用Python，可以使用numpy等基础数学库。不可使用PyTorch、Tensorflow、Scikit-learn等机器学习框架，不可使用Horovord等分布式机器学习框架

## II. 算法设计

### *i.问题分析*

简单的同步式数据并行参数服务器**(Parameter Server)**：使用一个进程作为Server，其余进程作为Worker。Server负载模型初始化和收集权重并更新，Worker负责样本选取和梯度计算。然而，这种策略存在三个主要问题：

- Server进程容易成为瓶颈，等待通信消耗大量时间，负载不均衡
- 通信和计算没有重叠
- 不同进程耗时可能不一样

本次实验针对第一种问题进行了改进，使用**Ring All-reduce**技术，使得每个进程都同时是Server和Worker，均衡了负载。

### *ii.算法描述*

#### 1. Serial Baseline

- 初始化模型
- 随机选取一组数据，反向传播计算出梯度
- 用梯度更新网络权重
- 重复迭代2-3

## 2. Parameter Server

- Server初始化模型参数，发送到所有Worker
- 每个Worker随机读取一组数据，反向传播计算出梯度
- Worker将梯度发送到Server，Server等待接收
- Server将梯度作平均，更新模型
- Server将更新后的模型发送到Worker上
- 重复迭代2-5

## 3. Ring All-reduce

- 0号进程进行模型初始化，发送到所有其他进程
- 每个进程并行随机读取一组数据，反向传播计算出梯度
- 全体进程组进行梯度的Allreduce，此时每个进程上的梯度都相同，为原先全体进程上梯度之和
- 每个进程将梯度除以总进程数计算平均值，再更新网络权重
- 重复迭代2-4

# III. 实验评测

## *i.实验配置*

本次实验在服务器上运行，相关配置如下:

- **24v**CPUs (Intel(R) Xeon(R) CPU E5-2650 v4 @ 2.20GHz)
- Linux version 3.10.0-1160.el7.x86_64 (gcc version 4.8.5, Red Hat 4.8.5-44)
- **Python** 3.9.11:
  - mpi4py==3.0.3
  - numpy==1.21.2
  - matplotlib==3.5.2
- **MPI: openmpi 4.0.2**

由于三种算法每次迭代的训练数据量不同，为了保证总训练数据量相同。本次实验训练参数配置与迭代次数如下:

- 学习率`lr=1e-4`
- 总训练轮次数`epochs=5`
- 训练集图片数`X.shape[0]=60000`
- Mini batch大小为`batch_size=32`
- 三种算法迭代次数(取整):
  - 对于串行baseline，每次迭代计算1个batch，因此迭代次数为`epochs*X.shape[0]/batch_size`
  - 对于Parameter Server，每次迭代计算(size-1)个batch，因此迭代次数为`epochs*X.shape[0]/(batch_size*(size-1))`
  - 对于Ring All-reduce，每次迭代计算(size)个batch，因此迭代次数

为`epochs*X.shape[0]/(batch_size*size)`

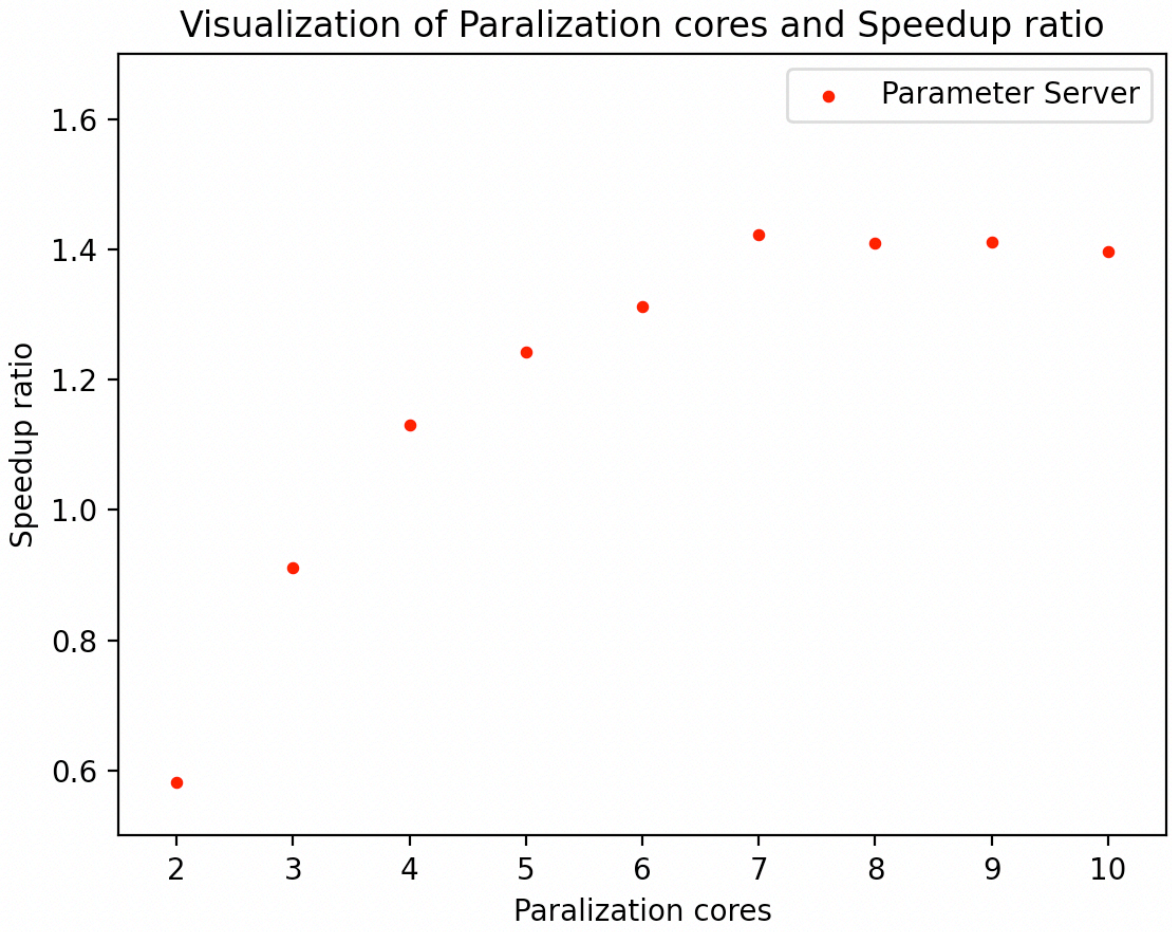
# *ii.实验结果*

## 1. Serial Baseline

总运行时间: **8.641s**

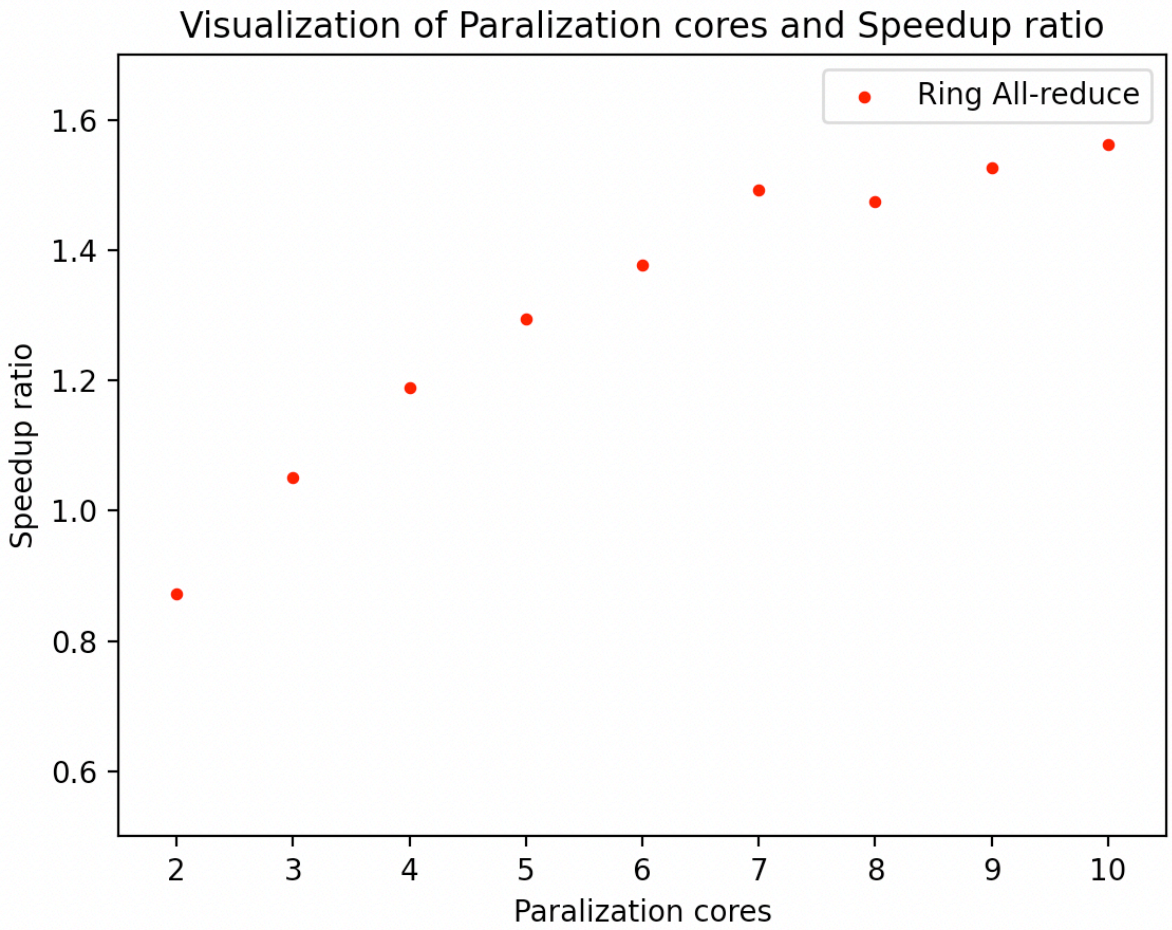
## 2. Parameter Server

| 并行进程数 | 运行时间/s | 加速比 |
|---|---|---|
| 2 | 14.841 | 0.582 |
| 3 | 9.475 | 0.912 |
| 4 | 7.648 | 1.130 |
| 5 | 6.954 | 1.243 |
| 6 | 6.583 | 1.313 |
| 7 | 6.075 | 1.422 |
| 8 | 6.126 | 1.411 |
| 9 | 6.119 | 1.412 |
| 10 | 6.184 | 1.397 |

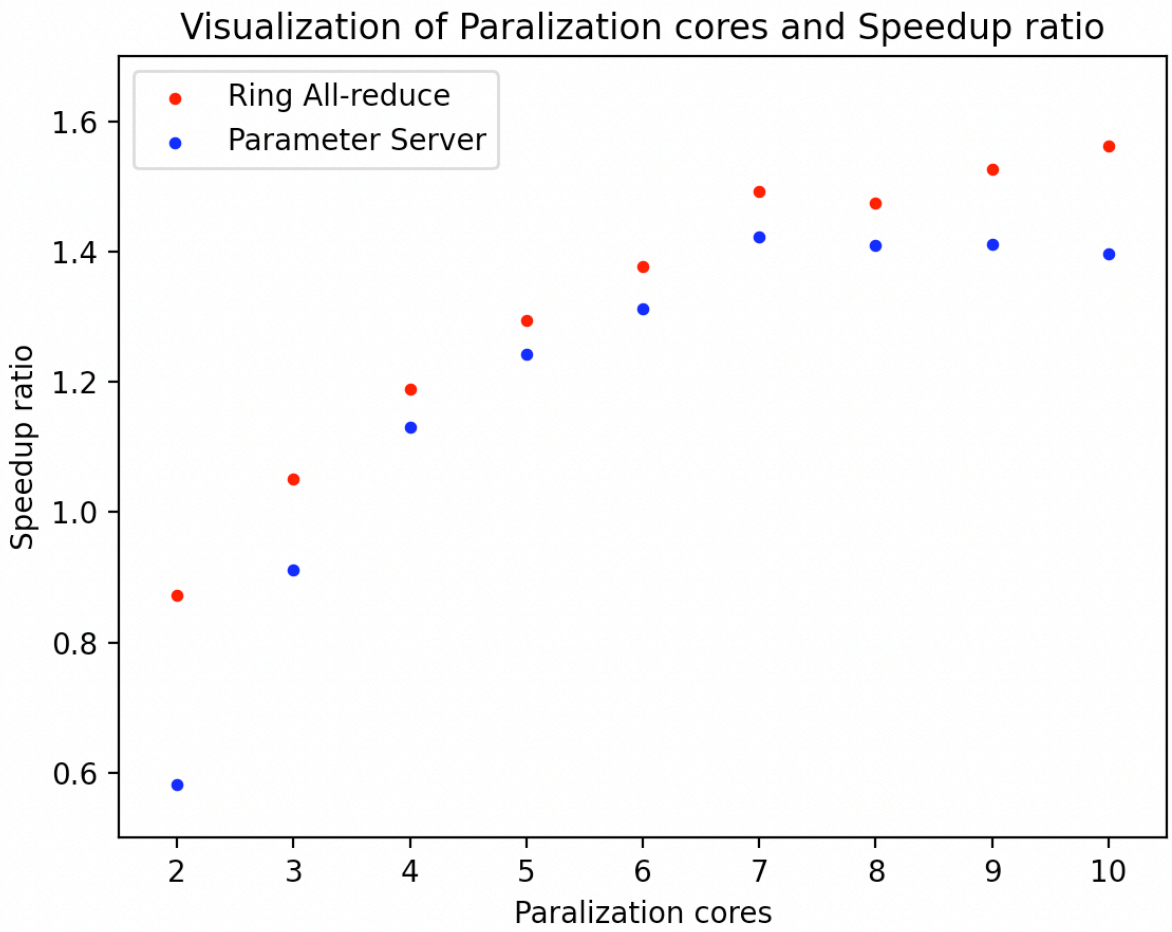



### 3. Ring All-reduce

| 并行进程数 | 运行时间/s | 加速比 |
|---|---|---|
| 2 | 9.914 | 0.872 |
| 3 | 8.221 | 1.051 |
| 4 | 7.263 | 1.190 |
| 5 | 6.678 | 1.294 |
| 6 | 6.275 | 1.377 |
| 7 | 5.787 | 1.493 |
| 8 | 5.859 | 1.475 |
| 9 | 5.659 | 1.527 |
| 10 | 5.529 | 1.563 |


Visualization of Paralization cores and Speedup ratio
Ring All-reduce
Speedup ratio
1.6
1.4
1.2
1.0
0.8
0.6
2
3
4
5
6
7
8
9
10
Paralization cores

**4. 性能对比分析**




- 结果表明，Ring All-reduce算法在实验范围内的任意并行数下，性能均要好于Parameter Server算法，这是由于RA算法中每个进程都在进行梯度运算，而PS算法中有一个Server进程没有进行梯度运算，负载很轻，从而产生了空等
- 在较低并行数下，两者性能均弱于单进程串行算法；在中间较大范围内，Ring All-reduce和Parameter Server均表现出了不错的性能
- 当并行数达到一定大小时(N≥7)，Ring All-reduce和Parameter Server的加速比几乎停滞而不再随并行数增大而增大，这是由于通信的开销增长几乎和多进程并行的性能提高相互抵消

# IV. 结论

本次实验考量了不同并行数下的并行算法效果，同时对比测量了通信优化的实际效果。结果表面随着并行数的增加，运行时间通常有所减少，加速比增加；随着通信次数的减小，加速比也会随之提高。

# 参考

[1] MNIST dataset: http://yann.lecun.com/exdb/mnist/

[2] Robbins et al. A Stochastic Approximation Method. Sep., 1951

[3] Ring All-reduce: https://towardsdatascience.com/visual-intuition-on-ring-allreduce-for-distributed-deep-learning-d1f34b4911da

# 附录

本次实验的代码都与本文档一同打包提交，其中`base_mnist.py`为串行算法作对比用，`param_server_mnist.py`为参数服务器并行算法，`ring_allreduce_mnist.py`为RingAll-reduce并行算法，`utils.py`为上述三种算法所用到的机器学习相关函数。