南開大學
Nankai University

计算机学院

并行程序设计期末报告

ANN 的并行优化

姓名：林语盈

学号：2012174

专业：计算机科学与技术

2022 年 7 月 10 日

摘要

本文首先介绍期末大作业选题的问题描述。接下来，介绍相关的并行算法。然后，介绍最终并行版本的设计思路。最后，进行不同平台、不同数据规模下的实验和结果分析。

项目源代码链接 https://github.com/AldebaranL/Parallel-programming-Homework

关键字: ANN; SIMD; openMP;MPI;CUDA

目录

1 问题描述

ANN，Artificial Network，人工神经网络，泛指由大量的处理单元 (神经元) 互相连接而形成的复杂网络结构，是对人脑组织结构和运行机制的某种抽象、简化和模拟。

ANN 可以有很多应用场景，在本次期末大作业中拟将其运用于简单的特征分类实际问题中，具体地，在鸢尾花数据集中根据花的属性判断花的种类，在乳腺癌良恶性预测数据集中根据病人特征判断其乳腺癌是良性还是恶性。

- 问题输入: 训练样本及其标签数据矩阵
- 问题输出: 模型的参数矩阵，当输入新的训练样本，可用以计算其预测值。

ANN 的模型结构，如图1.1所示

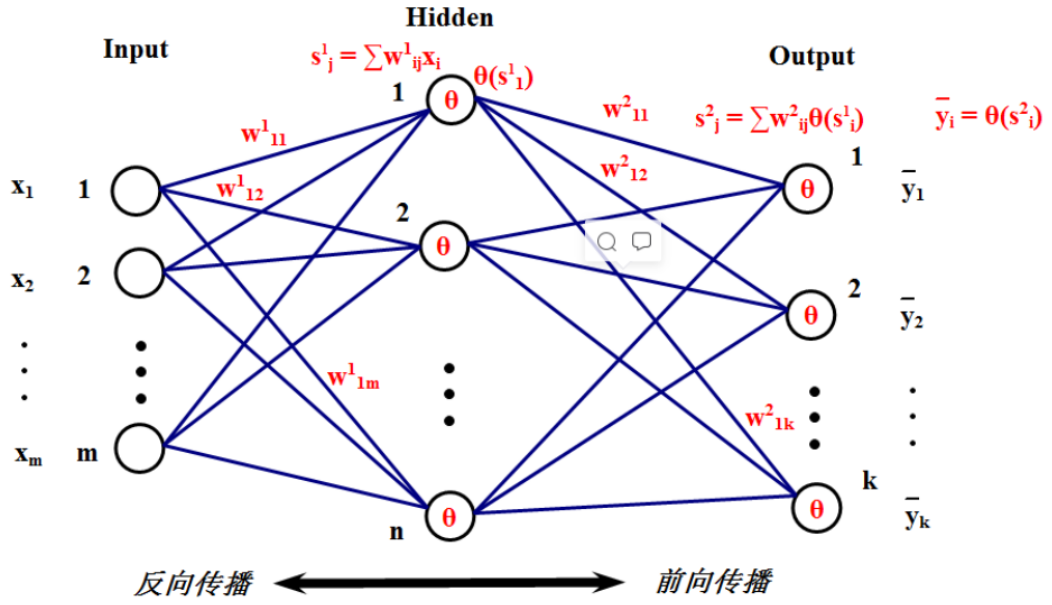


图 1.1: ANN 模型结构

其中，输入层维度为 m，输出层维度为 k，为了简化，假设仅有一个隐藏层，维度为 n。模型的参数分为两部分，W_1 为一个 m*n 的权值矩阵，用于计算输入层到隐藏层，W_2 为一个 n*k 的权值矩阵，用于计算隐藏层到输出层。此外，需要维护一个大小相同的导数矩阵，用于更新权值。

ANN 计算过程中主要有前向计算与反向传播的参数求导及其更新等过程，其算法描述如下。

1.1 前向传播

前向传播即由输入样本矩阵计算出对应的输出标签矩阵，其中每一层的计算即参数矩阵相乘，再加之偏置。由第 0 层至最后一层依次计算。设第 l 层的维度为 N_{l-1}, 第 l 层的计算如下:

$$\begin{aligned} \boldsymbol{z}_{(l)} &= W_{(l)} \boldsymbol{o}_{(l-1)} + \boldsymbol{b}_{(l)} \\ \boldsymbol{o}_{(l)} &= f\left(\boldsymbol{z}_{(l)}\right) \end{aligned} \tag{1}$$

其中 $o_{(l)} \in R^{N_l}$ 为第 l 层的输出，第 0 层的输出就是 ANN 的输入样本 x。$W_{(l)} \in R^{N_l} \times R^{N_{l-1}}$ 为第 l 层的权重矩阵，$b_{(l)} \in R^{N_l}$ 为第 l 层的偏置。

Algorithm 1 前向传播

```
function PREDICT
    for each layer do
        layers_{i+1}.outputNodes ← layers_i.weights * layers_i.outputNodes + layers_i.Bias
        ayers_{i+1}.outputNodes ← activationFunction(ayers_{i+1}.outputNodes)
    end for
end function
```

其中的 f 函数, 即激活函数, 可能有多种选择, 它的加入是为增加非线性性。常见的激活函数包括 sigmoid 和 tanh, 本实验中采用 sigmoid 函数。

$$f(x) = Sigmoid(x) = \frac{1}{1+e^{-x}} \tag{2}$$

其导数为:

$$f'(x) = Sigmoid'(x) = Sigmoid(x) * (1 - Sigmoid(x)) \tag{3}$$

1.2 损失函数

对于一个训练数据 $\left(\boldsymbol{x}^{(i)}, \boldsymbol{y}^{(i)}\right)$，全链接层有多个输出节点 $o_1^{(i)}, o_2^{(i)}, o_3^{(i)}, \ldots, o_K^{(i)}$ ，每个输出节点对应不同真实标签 $y_1^{(i)}, y_2^{(i)}, y_3^{(i)}, \ldots, y_K^{(i)}$ 均方误差可以表示为:

$$L^{(i)} = \frac{1}{2} \sum_{i=1}^{K} \left(o_i^{(i)} - y_i^{(i)}\right)^2 \tag{4}$$

一个 batch 内所有数据的平均代价函数为:

$$L_{total} = \frac{1}{batchSize} \sum_{i=1}^{batchSize} L^{(i)} \tag{5}$$

1.3 反向传播与参数更新

每次迭代，由最后一层向前依次计算，采用梯度下降法更新参数:

$$\begin{aligned} W_{(l)} &= W_{(l)} - studyRate * \frac{\partial L_{\text{total}}}{\partial W_{(l)}} \\ \boldsymbol{b}_{(l)} &= \boldsymbol{b}_{(l)} - studyRate * \frac{\partial L_{\text{total}}}{\partial \boldsymbol{b}_{(l)}} \end{aligned} \tag{6}$$

现计算其中的偏导数，记 l 层损失函数对于偏置的偏导数为:

$$\delta_{(l)} = \frac{\partial L_{total}}{\partial b_{(l)}}$$

设 L 为神经网络的层数。最后一层（l=L）的偏导数可直接由 L_{total} 得到:

$$\delta_{i,(L)} = -\frac{1}{batchSize} \sum_{k=1}^{batchSize} \left(y_i^{(k)} - o_{i,(L)}^{(k)}\right) f'\left(o_{i,(L)}^{(k)}\right) (1 \le i \le N_L) \tag{7}$$

对于其他每层, 根据求导的链式法则，经过推导可表示为:

$$\begin{aligned}\delta_{i,(l)} = \frac{\partial L_{total}}{\partial b_{(l)}} &= f'(o_{i,(l)}) * W_{ij,(l+1)} * \delta_{j,(l+1)} \, (1 \le i \le N_l, 1 \le j \le N_{l+1}) \\ \frac{\partial L_{total}}{\partial w_{ij,(l)}} &= \delta_{i,(l)} o_{j,(l-1)} \, (1 \le i \le N_l, 1 \le j \le N_{l-1})\end{aligned} \tag{8}$$

Algorithm 2 反向传播与参数更新

```
1: function CALULATE DELTA
2:     for each layer reversely do
3:         error ← layers_{i+1}.weights * layers_{i+1}.delta
4:         layers_i.delta ← error * erivative_activation_function(layers_i.output_nodes)
5:     end for
6: end function
7:
8: function BACK PROPAGATION
9:     for each layer reversely do
10:        layers_i.weights− = studyRate * layers_i.delta * layers_{i+1}.outputNodes
11:        layers_i.bias− = studyRate * layers_i.delta
12:    end for
13: end function
```

1.4 训练

在整个训练过程中，重复 numEpoch 次，每次遍历全部样本。对于每 batchSize 个样本，进行 batchSize 次前向传播，并计算总 loss。每 batchSize 个样本，进行一次导数计算和反向传播，更新参数。

Algorithm 3 训练过程

```
1: function TRAIN
2:     for each epoch do
3:         while samples are not traversed do
4:             for each sample in batch do
5:                 predict()
6:             end for
7:             calulate loss()
8:             calulate delta()
9:             back propagation()
10:        end while
11:    end for
12: end function
```

2 相关研究综述

随着深度学习模型的快速发展和应用，越来越多的大模型被应用到了各个领域，使用大量的模型参数解决复杂的问题。训练大模型的主要瓶颈是对大量 GPU 内存的强烈需求，远远高于单个 GPU 上的内存。大模型在训练过程中，GPU 内存主要用在模型权重参数存储（例如数百亿个浮点数），中间计算输出存储（例如梯度和优化器状态，例如 Adam 中的动量和变化），这使得 GPU 成本十分昂贵。除此之外，训练一个大模型通常时通常需要与一个大型训练语料库配对，因此单个过程可能需要很长时间。因此，并行性在训练大模型时十分必要。而 ANN 是许多复杂模型的基础组成部分，对 ANN 的并行优化在一定程度上能对绝大多数模型有着很好的加速效果。目前也有很多对 ANN 的并行优化研究。

2.1 向量化架构与多线程架构

1992 年，Wojtek Przytula[19] 等人基于图论方法，提出了将神经网络工作映射到网格连接 SIMD 阵列的系统方法。该方法用稀疏矩阵向量运算来表示多层网络模型，映射适用的计算机类别包括大多数实验性和商用网状连接 SIMD 处理器阵列。

2.2 多进程、多机架构

1995 年，Malluhi 等人提出了一种在超立方体大规模并行机上映射人工神经网络（ANN）的技术[9]。提出了附加树网格（MAT），用于快速实现 ANN，给出了一个将 MAT 结构嵌入超立方体拓扑的递归过程。这一过程被用作在 hy percube 系统上高效映射 ANN 计算的基础。此外，还考虑了带反向传播的多层前馈（FFBP）和 opfield ANN 模型。设 n 是最大层的大小，实了 O（logn）时间的算法，允许多个输入模式的流水线，从而进一步提高了性能。

2004 年，Udo Seiffert[14] 进行了大规模并行计算机硬件和多层感知器与反向传播并行计算机硬件综述，发现 MPI 是并行计算机硬件的一个很好的折衷方案。2008 年，G.Dahl et al.[3] 等人构造了八位奇偶校验、模式并行训练（PPT）和 FANN 工具 +MPI 库，PPT，8 节点版本（65 秒 vs 689）实现了 10.6 的最佳加速。2008 年，Lyle N. Long et al.[8] 等人实现了识别字符集和 SPANN[面向对象（C++）+MPI 库]。SPANN 可以扩展到训练神经元，并允许使用数十亿个权重。它用于小型串行计算机，并允许人工神经网络在一台极为并行的计算机上进行训练，从而显著降低了通信成本。2009 年，R. Rabenseifner et at[12] 实现了集群系统上并行编程、MPI 通信和 OpenMP 的分层硬件设计。2010 年，V. Turchenko et al.[17] 使用 OpenMP 和 MPICH2 进行并行批处理模式培训和并行编程,MPI–Allreduce。2011 年，H.Jin et al.[6] 进行了多区域 NAS 并行基准测试和 NAS 并行基准测试（NPB-MZ）。2011 年，A.Dobnikar et al.[13] 等人实现了基于 Pthreads 与 OpenMP 的并行批训练和顺序并行反向传播学习算法。通过从可用的内核创建尽可能多的服务器线程，可以实现最佳的加速，而使用 OpenMP 并行化训练算法的效果最佳。但是，如果使用的线程多于内核，则从一个线程切换到另一个线程所需的时间非常长，因此并行化的效率相当低。2016 年，Chanthini, P and Shyamala,K[2] 对 ANN 的 MPI 并行进行了总结和展望。

2.3 GPU 相关架构

2010 年，Sierra-Canto 等人 [16] 在 NVIDIA 开发的并行计算体系结构 CUDA 上实现反向传播算法。使用 CUBLAS（基本线性代数子程序库（BLAS）的 CUDA 实现）简化了过程, 使用两个标准基准数据集对该实现进行了测试，并行训练算法的运行速度是顺序训练算法的 63 倍。

在此之后,针对不同的特定任务,人们在 ANN 的基础上进行了特定的优化。包括 Altaf AhmadHuqqani[5] 等人人脸识别神经网络的多核和 GPU 并行化,Junliang Wang 等人 [18] 大数据驱动晶圆生产计划周期时间并行预测等。

2.4 Megatron 框架

近期，NVIDIA 提出了一个基于 PyTorch 的分布式训练框架 Megatron[15, 11, 7]，它可以用来训练超大 Transformer 语言模型，可以通过综合应用数据并行，Tensor 并行和 Pipeline 并行来复现拥有 1750 亿参数的超大规模模型 GPT3[1]。Megatron 中很重要的一部分就是对 ANN 的前向传播和反向传播进行并行优化。使用硬件加速器来横向扩展（scale out）深度神经网络训练主要有两种模式：数据并行，模型并行。

2.4.1 数据并行

数据并行模式会在每个 worker 之上复制一份模型，这样每个 worker 都有一个完整模型的副本。输入数据集是分片的，一个训练的小批量数据将在多个 worker 之间分割；worker 定期汇总它们的梯度，以确保所有 worker 看到一个一致的权重版本。对于无法放进单个 worker 的大型模型，人们可以在模型之中较小的分片上使用数据并行。

数据并行扩展通常效果很好，但有两个限制：1）超过某一个点之后，每个 GPU 的 batch size 变得太小，这降低了 GPU 的利用率，增加了通信成本；2）可使用的最大设备数就是 batchsize，这限制了可用于训练的加速器数量。

2.4.2 模型并行

人们会使用一些内存管理技术，如激活检查点（activation checkpointing）来克服数据并行的这种限制，也会使用模型并行来对模型进行分区来解决这两个挑战，使得权重及其关联的优化器状态不需要同时驻留在处理器上。模型并行模式会让一个模型的内存和计算分布在多个 worker 之间，以此来解决一个模型在一张卡上无法容纳的问题，其解决方法是把模型放到多个设备之上。

模型并行分为两种：流水线并行和张量并行，就是把模型切分的方式。流水线并行（pipeline model parallel）是把模型不同的层放到不同设备之上，比如前面几层放到一个设备之上，中间几层放到另外一个设备上，最后几层放到第三个设备之上。张量并行则是层内分割，把某一个层做切分，放置到不同设备之上，也可以理解为把矩阵运算分配到不同的设备之上，比如把某个矩阵乘法切分成为多个矩阵乘法放到不同设备之上。

2.4.3 技术组合

如图2.2所示，Megatron 展示了一个如何结合流水线、张量和数据并行，名为 PTD-P 的技术，这项技术将以良好的计算性能（峰值设备吞吐量的 52%）在 1000 个 GPU 上训练大型语言模型。PTD-P 利用跨多 GPU 服务器的流水线并行、多 GPU 服务器内的张量并行和数据并行的组合，在同一服务器和跨服务器的 GPU 之间具有高带宽链接的优化集群环境中训练具有一万亿个参数的模型，并具有优雅的扩展性。

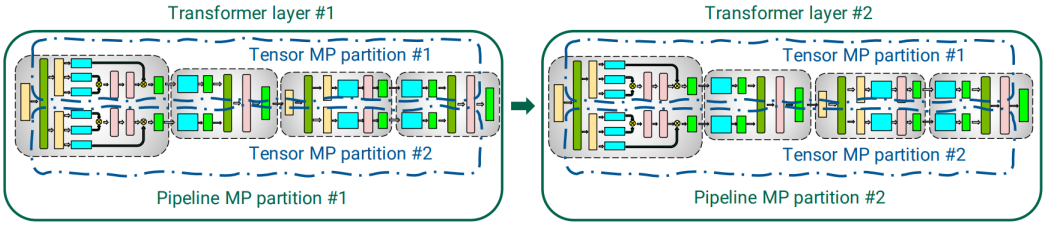


图 2.2: 张量并行与流水线并行组合 [11]

3 算法的设计与思想

在之前的历次作业中，使用的是随机的训练数据和标签来测试程序的运行时间，同时为了公平比较，采用的都是随机梯度下降的方式。在这种情况下没有具体的预测目标，并不评判其结果正确率的好坏而只是单纯地比较运行时间。在这次期末报告中，为了测试并行方法在 ANN 解决实际任务中的优化效果，将之前全手工实现的 ANN 模型用在了鸢尾花和乳腺癌两个数据集上，测试模型的实际运行时间和在不同数据集上的表现结果。

同时，考虑到之前使用的随机梯度下降存在着单个样本梯度方向不稳定，噪声较大的现象，本次的算法使用了小批量梯度下降法，随机选取一定批次的训练样本来计算梯度更新参数，这样在保证最终参数计算的准确率的同时，也能有较快的训练速度。对于一次参数的更新，一个 batch 内不同的样本可以进行并行的前向传播。

以下全部是自己设计的实现方法。

3.1 数据集介绍

在本次实验中将 ANN 应用到了鸢尾花卉种类预测和乳腺癌良恶性预测两个实际的分类任务中，在不同的任务中执行各种并行优化，研究其实际的加速效果，具有更高的参考价值。现介绍使用的两个实际数据集和分类任务。

3.1.1 Iris 鸢尾花卉数据集

Iris 也称鸢尾花卉数据集，是一类多重变量分析的数据集[1]。数据集包含 150 个数据样本，分为 3 类，每类 50 个数据，每个数据包含 4 个属性。可通过花萼长度，花萼宽度，花瓣长度，花瓣宽度 4 个属性预测鸢尾花卉属于（Setosa，Versicolour，Virginica）三个种类中的哪一类。

3.1.2 乳腺癌数据集

乳腺癌数据集[2]中一共有 699 个样本，共 11 列数据，如表1所示，每个样本有 10 属性征和 1 个对应的标签，所有数据特征值中包含 16 个缺失值，用“?”标出，标签值有两种情况，值为 2 表示乳腺癌为良性，值为 4 表示乳腺癌为恶性。

3.2 小批量梯度下降与并行优化

在之前的历次作业中，因不涉及具体任务，使用的都是随机梯度下降方法，使用每个样本计算梯度更新参数，这样会有较快的收敛速度。但在实际的任务当中，单个样本计算出来的梯度往往会有比较大

[1]Iris 鸢尾花卉数据集下载地址https://www.heywhale.com/mw/dataset/58a942bc7159a710d916af11

[2]乳腺癌数据集下载地址https://archive.ics.uci.edu/ml/machine-learning-databases/breast-cancer-wisconsin/

表 1: 乳腺癌数据集属性及标签信息

属性名称	属性含义
Sample code number	索引 ID
Clump Thickness	肿瘤厚度
Uniformity of Cell Size	细胞大小均匀性
Uniformity of Cell Shape	细胞形状均匀性
Marginal Adhesion	边缘粘附力
Single Epithelial Cell Size	单上皮细胞大小
Bare Nuclei	裸核
Bland Chromatin	染色质的颜色
Normal Nucleoli	核仁正常情况
Mitoses	有丝分裂情况
Class	分类情况，2 为良性，4 为恶性

的噪声，无法很好地拟合整体数据分布的梯度方向，参数优化过程不太稳定。因此在本次期末报告的鸢尾花种类预测和乳腺癌良恶性预测的任务上，使用了小批量梯度下降（Mini-batch Gradient Descent）的方法。它在有较快训练速度的同时还能够保证最后参数训练结果的准确率。

小批量梯度下降每次选取一定数量（batchSize）的训练样本作为一个批次（batch），对批次中的每个样本单独计算一次梯度和参数更新值，之后对批次中所有样本参数更新值求平均最为最后实际的一次梯度更新的值。在这种情况下，相同批次中的每个样本的前向传播和反向传播的过程都是独立的，因此可以很方便地进行并行优化，大致上可以有数据划分并行和流水线并行两类思路。

3.2.1 数据划分并行

可以将一个批次的数据划分成若干个等量的小批次分给不同的进程（线程），让它们分别执行小批次的前向传播和反向传播过程，最后将更新的结果汇总到其中某一个进程（线程），进行该批次实际的更新操作。另外，考虑到静态等量划分可能存在的数据负载不均、执行时间有差异的情况，可以使用一个主进程（线程）动态地为其他进程（线程）细粒度循环分配数据。

在使用 MPI 多进程数据划分并行的情况下，主进程需要在每个 batch 开始阶段向其他进程同步参数，而其他进程需要在每次计算完成后向向主进程发送参数更新值。这样的话总共的数据传输量为两倍的参数量，每个 batch 数据传输复杂度为 $O(pln^2)$，其中 p 是总共的进程数量，l 是 ANN 层数，n 是所有层中的最大节点数量，在静态分配的情况下每个 batch 总共的数据传输次数为 $3(p-1)$，在粒度为 1 的动态数据分配的情况下，数据传输次数为 $2(p-1)+batchSize$。

本次期末作业中，实现了数据划分的并行方式，每进行一次参数更新，多个不同的样本可以同时并行的进行前向传播和导数计算。

3.2.2 流水线并行

在数据划分并行中，不同的进程（线程）负责不同的数据，它们都会完成数据整个的前向传播和反向传播过程。而在流水线并行中，一个模型的各层会在多个进程（线程）上做切分。一个批次被分割成较小的微批（microbatches），并在这些微批上进行流水线式执行。图3.4展示了一种模型不同层的划分方式，其中第 0、1、2 层和 3、4、5 层划分给了不同的进程（线程）。在流水线模型并行中，训练会在一个进程（线程）上执行一组操作，然后将输出传递到流水线中下一个进程（线程）执行另一组不同操作。原生（naive）流水线会有这样的问题：一个输入在反向传播中看到的权重更新并不是其前向传

播中所对应的。所以，流水线方案需要确保输入在前向和反向传播中看到一致的权重版本，以实现明确的同步权重更新语义。

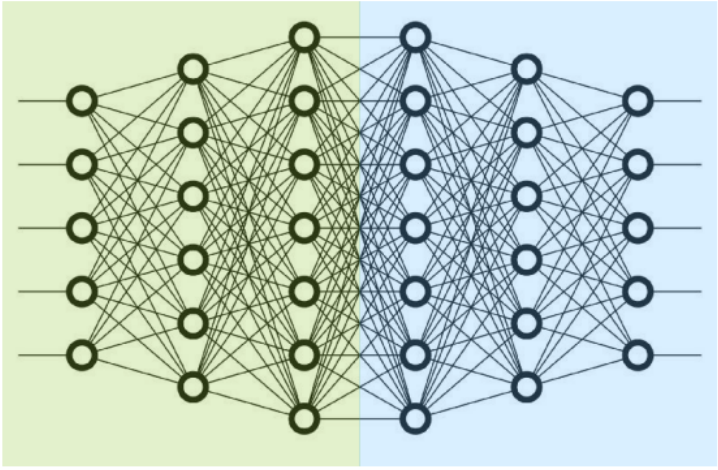


图 3.3: 流水线并行模型不同层划分示意图 [10]

如图在每个批次的开始和结束时，设备是空闲的。我们把这个空闲时间称为流水线 bubble，并希望它尽可能的小。根据注入流水线的微批数量，多达 50% 的时间可能被用于刷新流水线。微批数量与模型层数的比例越大，流水线刷新所花费的时间就越少。因此，为了实现高效率，通常需要较大的 batchSize。

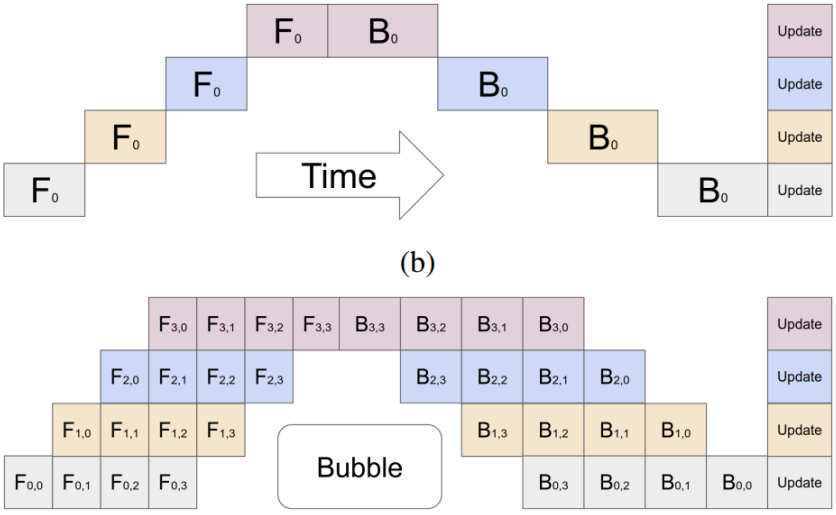


图 3.4: 流水线及 bubble 示意图 [4]

在使用 MPI 进行多进程流水线并行的情况下，主线程不再需要将所有的参数同步给其他进程，而只需要将不同层的权重分别同步给负载该层流水线计算任务的进程，考虑到整个模型的参数是 MPI 数据传输的主要负荷，这样可以在每个 batch 更新中只传输一次整体参数，参数传输的复杂度为 $O(ln^2)$，其中 l 是 ANN 层数，n 是所有层中的最大节点数量，相较于数据划分并行有着明显的性能提升。但是在样本数据传输上，每个进程需要负责所有样本的一部分网络层计算，因此需要从上一层进程中获取所有的样本输出，并给下一层进程发送自己的计算结果，这样的话总共的样本数据结果传输的复杂度

为 $O(npbatchSize)$，相较于数据划分并行的 $O(nbatchSize)$ 有着更高的中间样本结果传输复杂度。通过以上的分析可以看出，随着进程数量的增加，流水线并行参数传输不受影响，但样本中间数据传输时间会线性提升；数据划分并行参数传输时间会线性提升，但样本数据传输不受影响。二者各有优劣。

3.3 层内计算与并行优化

在 ANN 中，无论是前向传播、反向传播还是梯度更新，它们都需要对层内的每个节点的结果进行计算，同时层内的不同节点之间的计算是相互独立的。对前向传播（反向传播）来说，第 n 层的计算需要用到 n-1（n+1）层的输出，所以说不同隐藏层之间的计算实际上是串行的。其主要思想如图3.5所示，在同一层内，将不同节点输出的计算任务分配给不同的进程（线程），它们互不干扰。

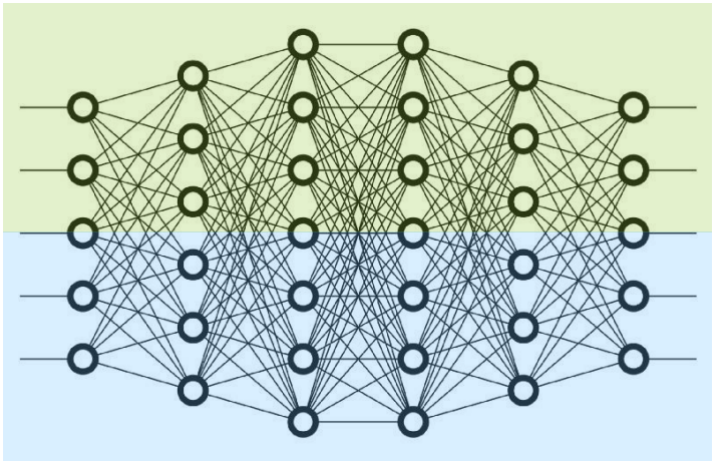
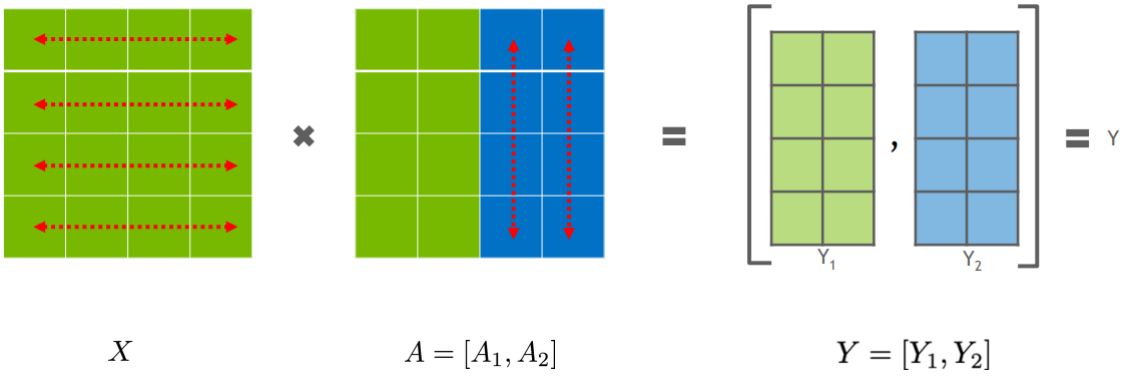


图 3.5: 层内不同节点计算任务划分 [10]

以前向传播为例，由第一章的公式1可以看出，当前层的计算主要需要用到上一层的输出和当前层的权重与偏置，而对节点计算任务的分配实质上是对权重矩阵和上一层输出矩阵乘法运算的分配操作。举例来说，假如有权重矩阵 A 和输入矩阵 X，它们的矩阵乘积操作可以进行如图3.6所示，可以将输入矩阵 A 与权重矩阵 X 的不同列相乘，并将它们的结果拼接起来得到最后的结果。



X $A = [A_1, A_2]$ $Y = [Y_1, Y_2]$

图 3.6: 矩阵乘法分解操作示意图

在使用 MPI 对层内计算进行并行优化时，主进程需要向其他进程发送它们需要负责计算的矩阵行列，在这里是上一层的输出以及需要计算节点对应的的权重和偏置，每个进程在计算得到自己负责节点的输出后，需要将结果发送给主进程，主进程在汇总得到所有节点的计算结果后才会继续下一层的

计算。由此可以得知整体的数据传输开销为 $O(ln^2)$，其中 l 是 ANN 层数，n 是所有层中的最大节点数量。

之前历次作业中，即为对特定层不同节点计算的任务分配，本次实验中，将网络结构整体进行了改变，也同样实现了特定层不同节点计算的任务分配的并行算法。

3.4 算法评估与并行正确性

在之前的历次实验中，使用的都是固定随机初始化的输入矩阵，观察 lossfunction 是否收敛与收敛速度。但在这次的期末研究中，使用了具体任务的数据集，进行鸢尾花分类和乳腺癌良恶性判断，它们都属于分类任务，那么就可以使用分类准确率作为评价指标，以二分类为例，分类结果有四种情况：

1. 预测值为正 (Positive)，并且真实值也为正 (Positive)，预测为真 (True)，True Positive (TP)。
2. 预测值为负 (Negative)，但是真实值为正 (Positive)，预测失败 (False)，False Negative (FN)。
3. 预测值为正 (Positive)，但是真实值为负 (Negative)，预测失败 (False)，False Positive (FP)。
4. 预测值为负 (Negative)，并且真实值也为负 (Negative)，预测为真 (True)，True Negative (TN)。

那么准确率的计算公式为：

$$Accuracy = \frac{n_{\text{correct}}}{n_{\text{total}}} = \frac{TP + TN}{TP + FN + FP + TN} \tag{9}$$

神经网络的运算对浮点数精度的要求不高，实验允许在并行优化后，由于计算次序的不同而导致的浮点计算精度误差，但是在使用同样的参数和算法的情况下，需要并行优化后的代码与串行计算版本在准确率评价指标上有着相同的结果，这样才能保证并行的正确性，并行优化才有意义。

3.5 实验问题规模的设置

在之前的实验中，考虑到只需要测试并行对串行程序的优化效果，固定了隐藏层层数为 1，ANN 迭代次数 numEpoch 为 128，不同训练样本数（即训练样本类别数 * 每个类别训练样本数）为 64。由于上述几个变量仅等同于重复的函数调用，可看作问题时间的等倍放大，所以之前的实验保持其不变。通过改变不同输入层结点数、输出层结点数、隐藏层结点数，来动态调整问题计算规模，测试程序运行时间与并行优化效果。

但是在这次实验中有了具体的任务，训练样本数取决于数据集的大小，隐藏层层数、迭代次数变成了会影响实验分类结果的重要参数，将他们固定不变不太合适，也需要通过不断调整找到合适的值。另外，由于本次实验使用了小批次梯度下降，训练样本的批次大小也是一个很重要的参数，它会影响并行优化中数据划分的粒度，实验中也将其作为一个可探索的参数设置。本次实验目标就变成了，固定输入层和输出层维度、训练样本数量，调整隐藏层层数、隐藏层节点数、迭代次数、批次大小，在研究不同问题规模下的并行优化效果的同时，也找到问题的最佳解决方法。

3.6 程序时间复杂度分析

理论上来说，假设网络层节点数最大为 n，网络层数为 l，则程序的时间复杂度为：

$$T = O(numEpoch * numSamples * n^2 * l) \tag{10}$$

理论上来说，4 个数的向量化最大带来 4 倍的加速比，加上数据打包解包等开销，加速比可能会略小于 4。n 个线程最多带来 n 倍的加速比，加上同步开销与线程初始化与创建的开销，加速比可能会略小于 n。对于 MPI 来说，假设创建了 n 个进程，在对等节点的模式下，n 个进程均会参加运算，理论上来说会有 n 倍的加速比，但受限于数据传输与同步的开销，加速比会远小于 n，尤其是在多个实际节点上运算时，数据的网络传输将极大地限制程序的加速效果；而在主从模式下，0 号进程用于分发和接收数据，相应地，只有 n-1 个进程参与实际的运算，整体加速比将小于等于 n-1。

4 算法实现

本次实验因为是在鸢尾花和乳腺癌两个具体的数据集上进行的，所以首先探究了不同的模型结构和参数设置对模型的表现结果的影响，然后选取表现较好的参数作为后续待并行优化的版本。之后实验综合所学的并行优化知识，分别使用指令集并行 SIMD、线程并行 OpenMP 和进程并行 MPI 来对串行算法进了并行优化。并且最后还综合了不同的优化方案得到了最终在这两个数据集上性能和结果都较优的解决方案。

4.1 数据处理与串行解决方案

该任务的串行解决方案首先对任务数据集进行了清洗和读入，在每个 epoch 中，将数据分为多个批次来梯度下降更新参数，数据处理和串行版本的代码如下：

数据处理与串行代码

```
#include <iostream>
#include"ANN_4.h"
#include"ANN_parallel.h"
#include<windows.h>
#include <cstdlib>
#include <ctime>
#include <fstream>
#include <iomanip>
#include <sstream>
#include <string>

#include"global.h"
using namespace std;


void creat_samples()
{
    TRAIN_MAT = new float* [NUM_SAMPLE];                    //生成训练样本
    for (int i = 0; i < NUM_SAMPLE; ++i)
    {
        TRAIN_MAT[i] = new float[NUM_EACH_LAYER[0]];
    }
    LABEL_MAT = new float* [NUM_SAMPLE]; //生成标签矩阵
    for (int i = 0; i < NUM_SAMPLE; ++i)
    {
```

```
26          LABEL_MAT[i] = new float[NUM_EACH_LAYER[NUM_LAYERS + 1]];
27          for (int j = 0; j < NUM_EACH_LAYER[NUM_LAYERS + 1]; ++j)
28          {
29              LABEL_MAT[i][j] = 0;
30          }
31      }
32      MAX_NUM_LAYER = NUM_EACH_LAYER[0];
33      for (int i = 1; i <= NUM_LAYERS + 1; i++) {
34          MAX_NUM_LAYER = max(NUM_EACH_LAYER[i], MAX_NUM_LAYER);
35      }
36      //MAX_NUM_LAYER = 256;
37      printf("finished creating samples\n");
38
39  }
40  void print_samples() {
41      for (int i = 0; i < min(5, NUM_SAMPLE); i++)
42      {
43          for (int j = 0; j < min(5, NUM_EACH_LAYER[0]); j++) printf("%f ",
                TRAIN_MAT[i][j]); printf("\n");
44          for (int j = 0; j < min(5, NUM_EACH_LAYER[NUM_LAYERS+1]); j++) printf("%f
                ", LABEL_MAT[i][j]);
45          printf("\n");
46      }
47  }
48  void delet_samples()
49  {
50      printf("begin delete samples ");
51      //释放内存
52      for (int i = 0; i < NUM_SAMPLE; ++i)
53          delete[] TRAIN_MAT[i];
54      delete[] TRAIN_MAT;
55      for (int i = 0; i < NUM_SAMPLE; ++i)
56          delete[] LABEL_MAT[i];
57      delete[] LABEL_MAT;
58      printf("finish delete\n");
59  }
60
61  void read_samples1(const char* filename) {
62      ifstream inFile(filename, ios::in);
63      if (!inFile)
64      {
65          cout << "打开文件失败! " << endl;
66          exit(1);
67      }
68      NUM_SAMPLE = 0;
69      string line;
70      getline(inFile, line);
71      NUM_EACH_LAYER[0] = 4;
72      NUM_EACH_LAYER[NUM_LAYERS + 1] = 3;
```

```
73
74      while (getline(inFile, line))//getline(inFile, line)表示按行读取CSV文件中的数据
75      {
76          string field;
77          istringstream sin(line); //将整行字符串line读入到字符串流sin中
78          for (int j = 0; j < NUM_EACH_LAYER[0]; j++) {
79              getline(sin, field, ',');
80              TRAIN_MAT[NUM_SAMPLE][j] = atof(field.c_str());
81          }
82          getline(sin, field);
83          LABEL_MAT[NUM_SAMPLE][atoi(field.c_str())] = 1;
84          NUM_SAMPLE++;
85      }
86      inFile.close();
87  }
88  void read_samples(const char* filename) {
89      ifstream inFile(filename, ios::in);
90      if (!inFile)
91      {
92          cout << "打开文件失败!" << endl;
93          exit(1);
94      }
95      NUM_SAMPLE = 0;
96      string line;
97      NUM_EACH_LAYER[0] = 9;
98      NUM_EACH_LAYER[NUM_LAYERS+1] = 2;
99
100     while (getline(inFile, line))//getline(inFile, line)表示按行读取CSV文件中的数据
101     {
102         string field;
103         istringstream sin(line); //将整行字符串line读入到字符串流sin中
104         getline(sin, field, ',');
105         for (int j = 0; j < NUM_EACH_LAYER[0]; j++) {
106             getline(sin, field, ',');
107             TRAIN_MAT[NUM_SAMPLE][j] = atof(field.c_str());
108         }
109         getline(sin, field);
110         LABEL_MAT[NUM_SAMPLE][atoi(field.c_str())/2-1] = 1;
111         NUM_SAMPLE++;
112     }
113     inFile.close();
114 }
115 int main(const int argc, char* argv[])
116 {
117     int provided;
118     MPI_Init_thread(&argc, &argv, MPI_THREAD_FUNNELED,
        &provided);//MPI_THREAD_MULTIPLE
119     int myid=0, numprocs;
120     MPI_Comm_rank(MPI_COMM_WORLD, &myid);
```

```
121
122
123     int thesize[10] = { 32,64,96,128,256, 512,768,1024,1280 };
124
125     for (int d_i = 0; d_i < 4; d_i++)
126     {
127         for (int l_i = 1; l_i < NUM_LAYERS + 1; l_i++) NUM_EACH_LAYER[l_i] =
                thesize[d_i];
128
129         //if (myid == 0) {
130         creat_samples();
131             //read_samples1("D:/Mycodes/VSProject/ANN_final/ANN_final/iris_training.csv");
132             read_samples("D:/Mycodes/VSProject/ANN_final/ANN_final/breast-cancer-wisconsin.data"
133             //print_samples();
134         //}
135          if (myid == 0) cout << "----------------------begin " << thesize[d_i] <<
                "--------------------" << endl;
136         for (int l_i = 0; l_i < NUM_LAYERS + 2; l_i++) cout << NUM_EACH_LAYER[l_i]
               << ' '; cout << endl;
137         //MPI_Barrier(MPI_COMM_WORLD);
138        // cout << endl;
139         ANN_parallel ann((int*)NUM_EACH_LAYER, 30, 12, NUM_LAYERS, 0.01);
140         ann.shuffle(NUM_SAMPLE, TRAIN_MAT, LABEL_MAT);
141        // ANN_parallel ann2((int*)NUM_EACH_LAYER, 50, 1, NUM_LAYERS, 0.005);
142        // ann2.shuffle(NUM_SAMPLE, TRAIN_MAT, LABEL_MAT);
143         int train_num = NUM_SAMPLE * 0.8;
144         MPI_Barrier(MPI_COMM_WORLD);
145         //ann.train_MPI(train_num, TRAIN_MAT, LABEL_MAT);
146         MPI_Barrier(MPI_COMM_WORLD);
147         //ann.train_MPI_openMP_SIMD(train_num, TRAIN_MAT, LABEL_MAT);
148         if (myid == 0) {
149             ann.train(train_num, TRAIN_MAT, LABEL_MAT);
150             ann.train_SIMD(train_num, TRAIN_MAT, LABEL_MAT);
151             ann.train_openMP_SIMD(train_num, TRAIN_MAT, LABEL_MAT);
152             ann.train_openMP(train_num, TRAIN_MAT, LABEL_MAT);
153             //ann.get_results(NUM_SAMPLE - train_num, &TRAIN_MAT[train_num],
                    &LABEL_MAT[train_num]);
154             delet_samples();
155         }
156     }
157     MPI_Finalize();
158     //printf("!!");
159     std::fflush(stdout);
160
161
162     return 0;
163 }
164
165
```

```cpp
#include "ANN_4.h"

ANN_4::ANN_4(int* _num_each_layer, int _num_epoch, int _batch_size, int
    _num_layers, float _study_rate)
{
    num_layers = _num_layers;
    study_rate = _study_rate;
    num_each_layer = new int[_num_layers + 2];
    num_each_layer[0] = _num_each_layer[0];
    for (int i = 1; i <= num_layers + 1; i++)
    {
        num_each_layer[i] = _num_each_layer[i];
        layers.push_back(new Layer(num_each_layer[i - 1],
            num_each_layer[i],_tanh));
    }
    //display();
    num_epoch = _num_epoch;
    batch_size = _batch_size;
}

ANN_4::~ANN_4()
{
    //printf ("begin free ANN_4");
    delete[]num_each_layer;
    for (int i = 0; i < layers.size(); i++) delete layers[i];
    // printf ("free ANN_4\n");
}
void ANN_4::shuffle(const int num_sample, float** _trainMat, float** _labelMat)
{
    //init
    int* shuffle_index = new int[num_sample];
    float** trainMat_old = new float* [num_sample];
    float** labelMat_old = new float* [num_sample];
    for (int i = 0; i < num_sample; i++)
    {
        trainMat_old[i] = new float[num_each_layer[0]];
        labelMat_old[i] = new float[num_each_layer[num_layers + 1]];
    }
    for (int i = 0; i < num_sample; i++)
    {
        for (int j = 0; j < num_each_layer[0]; j++)
            trainMat_old[i][j] = _trainMat[i][j];
        for (int j = 0; j < num_each_layer[num_layers + 1]; j++)
            labelMat_old[i][j] = _labelMat[i][j];
    }

    //shuffle
    for (int i = 0; i < num_sample; i++) shuffle_index[i] = i;
    random_shuffle(shuffle_index, shuffle_index + num_sample);
```

```
213     for (int i = 0; i < num_sample; i++)
214     {
215         for (int j = 0; j < num_each_layer[0]; j++) _trainMat[i][j] =
                trainMat_old[shuffle_index[i]][j];
216         for (int j = 0; j < num_each_layer[num_layers + 1]; j++) _labelMat[i][j] =
                labelMat_old[shuffle_index[i]][j];
217     }
218
219     //delete
220     for (int i = 0; i < num_sample; i++)
221     {
222         delete[] trainMat_old[i];
223         delete[] labelMat_old[i];
224     }
225     delete[] trainMat_old;
226     delete[] labelMat_old;
227     delete[] shuffle_index;
228
229     printf("finish shuffle\n");
230 }
231 void ANN_4::get_results(const int num_sample, float** _trainMat, float**
        _labelMat) {
232     float tp = 0, fp = 0, tn = 0, fn = 0;
233     float *Y;
234     for (int index = 0; index < num_sample; index++) {
235         this->predict(_trainMat[index]);
236         Y = layers[num_layers]->output_nodes;
237         float maxn = -1,maxi = 0;
238         for (int j = 0; j < num_each_layer[num_layers + 1]; j++) {
239             if (Y[j] > maxn) {
240                 maxn = Y[j];
241                 maxi = j;
242             }
243         }
244         if (_labelMat[index][(int)maxi] == 1) {
245             tp++;
246         }
247         else {
248             fp++;
249             cout << maxn << maxi;
250             this->show_predictions(_trainMat[index]);
251         }
252     }
253     cout << "tp:" << tp  << endl;
254     cout << "fp:" << fp << endl;
255     cout << "accuracy:" << tp / (tp + fp) << endl;
256 }
257
258 void ANN_4::train(const int num_sample, float** _trainMat, float** _labelMat)
```

```
{
    std::printf("begin training\n");
    float thre = 1e-2;

    for (int epoch = 0; epoch < num_epoch; epoch++)
    {
        isNotConver_(num_sample, _trainMat, _labelMat, thre);
        // if (epoch % 50 == 0) printf ("round%d:\n", epoch);
        for (int batch_i = 0; batch_i < (num_sample + batch_size - 1) /
            batch_size; batch_i++) {
            for (int i = num_layers; i >= 0; i--)
                for (int j = 0; j < num_each_layer[i + 1]; j++) {
                    for (int k = 0; k < num_each_layer[i]; k++) {
                        layers[i]->weights_delta[j][k] = 0;
                    }
                    layers[i]->bias_delta[j] = 0;
                }
            int t_batch_size = min((batch_i + 1) * batch_size, num_sample) -
                batch_i * batch_size;
            if (t_batch_size == 0)cout << "t_batch_szie wrong!!" << endl;
            for (int index = batch_i * batch_size; index < batch_i * batch_size +
                t_batch_size; index++) {
                //前向传播
                float output_node;
                for (int i = 0; i < num_each_layer[1]; i++)
                {
                    output_node = 0.0;
                    for (int j = 0; j < num_each_layer[0]; j++)
                    {
                        output_node += layers[0]->weights[i][j] *
                            _trainMat[index][j];
                    }
                    output_node += layers[0]->bias[i];
                    layers[0]->output_nodes[i] =
                        layers[0]->activation_function(output_node);//输出-1到1
                }
                for (int i_layer = 1; i_layer <= num_layers; i_layer++)
                {
                    for (int i = 0; i < num_each_layer[i_layer + 1]; i++)
                    {
                        output_node = 0.0;
                        for (int j = 0; j < num_each_layer[i_layer]; j++)
                        {
                            output_node += layers[i_layer]->weights[i][j] *
                                layers[i_layer - 1]->output_nodes[j];
                        }
                        output_node += layers[i_layer]->bias[i];
                        layers[i_layer]->output_nodes[i] =
                            layers[i_layer]->activation_function(output_node);
```

```
301                     }
302                 }
303                 //计算delta
304                 float* temp_bias_delta = new float[MAX_NUM_LAYER];
305                 for (int j = 0; j < num_each_layer[num_layers + 1]; j++)
306                 {
307                     //均方误差损失函数
308                     temp_bias_delta[j] = (layers[num_layers]->output_nodes[j] -
                            _labelMat[index][j]) *
                            layers[num_layers]->derivative_activation_function(layers[num_layers]->o
                            //输出0到1
309                     layers[num_layers]->bias_delta[j] += temp_bias_delta[j];
310                     //交叉熵损失函数
311                     //layers[num_layers]->delta[j] +=
                            (layers[num_layers]->output_nodes[j] - _labelMat[index][j]);
312                     for (int k = 0; k < num_each_layer[num_layers]; k++)
313                         layers[num_layers]->weights_delta[j][k] +=
                                temp_bias_delta[j] * layers[num_layers -
                                1]->output_nodes[k];
314                 }
315                 //计算每层的delta,行访问优化
316                 for (int i = num_layers - 1; i >= 0; i--)
317                 {
318                     float* error = new float[num_each_layer[i + 1]];
319
320                     for (int j = 0; j < num_each_layer[i + 1]; j++) error[j] = 0.0;
321                     for (int k = 0; k < num_each_layer[i + 2]; k++)
322                     {
323                         for (int j = 0; j < num_each_layer[i + 1]; j++)
324                         {
325                             error[j] += layers[i + 1]->weights[k][j] *
                                    temp_bias_delta[k];
326                         }
327                     }
328                     for (int j = 0; j < num_each_layer[i + 1]; j++)
329                     {
330                         temp_bias_delta[j] = error[j] *
                                layers[num_layers]->derivative_activation_function(layers[i]->output
331                         layers[i]->bias_delta[j] += temp_bias_delta[j];
332                         for (int k = 0; k < num_each_layer[i]; k++)
333                             if (i == 0) layers[0]->weights_delta[j][k] +=
                                    temp_bias_delta[j] * _trainMat[index][k];
334                             else layers[i]->weights_delta[j][k] +=
                                    temp_bias_delta[j] * layers[i - 1]->output_nodes[k];
335                     }
336                     delete[] error;
337                 }
338
339                 delete[] temp_bias_delta;
```

```
340             }
341
342             //反向传播, weights和bias更新
343             for (int i = 0; i <= num_layers; i++)
344             {
345                 for (int k = 0; k < num_each_layer[i + 1]; k++)
346                 {
347                     for (int j = 0; j < num_each_layer[i]; j++)
348                     {
349                         layers[i]->weights[k][j] -= study_rate *
                                layers[i]->weights_delta[k][j] / t_batch_size;
350                     }
351                     layers[i]->bias[k] -= study_rate * layers[i]->bias_delta[k] /
                            t_batch_size;
352                 }
353             }
354         }
355     }
356     std::printf("finish training\n");
357 }
358
359 bool ANN_4::isNotConver_(const int _sampleNum, float** _trainMat, float**
        _labelMat, float _thresh)
360 {
361     float lossFunc = 0.0;
362     for (int k = 0; k < _sampleNum; ++k)
363     {
364         predict(_trainMat[k]);
365         float loss = 0.0;
366         for (int t = 0; t < num_each_layer[num_layers + 1]; ++t)
367         {
368             loss += (layers[num_layers]->output_nodes[t] - _labelMat[k][t]) *
                    (layers[num_layers]->output_nodes[t] - _labelMat[k][t]);
369         }
370         lossFunc += (1.0 / 2) * loss;
371     }
372
373     lossFunc = lossFunc / _sampleNum;
374
375     static int tt = 0;
376     tt++;
377      printf ("第%d次训练: %0.12f\n", tt, lossFunc);
378
379     if (lossFunc > _thresh) return true;
380     return false;
381 }
382
383 void ANN_4::predict(float* in)
384 {
```

```
385     layers[0]->_forward(in);
386     for (int i = 1; i <= num_layers; i++)
387     {
388         layers[i]->_forward(layers[i - 1]->output_nodes);
389     }
390 }
391 void ANN_4::display()
392 {
393     for (int i = 0; i <= num_layers; i++)
394     {
395         layers[i]->display();
396     }
397 }
398
399
400 void ANN_4::show_predictions(float* X)
401 {
402     predict(X);
403     static int t = 0;
404     printf("in%d:", t);
405     for (int i = 0; i < min(5, num_each_layer[0]); i++) printf("%f,", X[i]);
406     printf("  out%d:", t);
407     for (int i = 0; i < min(5, num_each_layer[num_layers + 1]); i++)
408         printf("%f,", layers[num_layers]->output_nodes[i]);
409     t++;
410     printf("\n");
411     //return layers[num_layers]->output_nodes;
412 }
413
414 #include "Layer.h"
415
416 Layer::Layer(int _num_input, int _num_output, activationType
        _activation_type=_tanh)
417 {
418     //ctor
419     num_input = _num_input;
420     num_output = _num_output;
421     weights = new float* [num_output];
422     for (int i = 0; i < num_output; i++) weights[i] = new float[num_input];
423     weights_delta = new float* [num_output];
424     for (int i = 0; i < num_output; i++) weights_delta[i] = new float[num_input];
425
426     bias = new float[num_output];
427     output_nodes = new float[num_output];
428     //error = new float[num_output];
429     bias_delta = new float[num_output];
430
431     for (int i = 0; i < num_output; i++)
432     {
```

```
433         for (int j = 0; j < num_input; j++)
434         {
435             weights[i][j] = ((rand() % 20000) - 10000) / 10000.0;
436         }
437         bias[i] = ((rand() % 20000) - 10000) / 10000.0;
438     }
439     //activation_type = tanh;
440     activation_type = _activation_type;
441 }
442
443 Layer::~Layer()
444 {
445     //dtor
446     delete[] bias;
447     delete[] output_nodes;
448     for (int i = 0; i < num_output; i++) {
449         delete[] weights[i];
450         delete[] weights_delta[i];
451     }
452     delete[]weights_delta;
453    // delete[]error;
454     delete[]bias_delta;
455 }
456 void Layer::display()
457 {
458     printf("%d,%d:\n", num_input, num_output);
459     for (int i = 0; i < num_input; i++)
460     {
461         for (int j = 0; j < num_output; j++)
462         {
463             printf("%f ", weights[j][i]);
464         }
465         printf("\n");
466     }
467 }
468 void Layer::_forward(float* input_nodes)
469 {
470     for (int i = 0; i < num_output; i++)
471     {
472         output_nodes[i] = 0.0;
473         for (int j = 0; j < num_input; j++)
474         {
475             output_nodes[i] += weights[i][j] * input_nodes[j];
476         }
477         output_nodes[i] += bias[i];
478         output_nodes[i] = activation_function(output_nodes[i]);
479         if (output_nodes[i] == NAN)printf("!!!!!!!");
480     }
481 }
```

```
482 float Layer::activation_function(float x)
483 {
484
485     switch (activation_type)
486     {
487     case _sigmoid:
488         return 1 / (1 + exp(-1 * x));
489     case _tanh:
490         return (exp(x) - exp(-1 * x)) / (exp(x) + exp(-1 * x));
491     default:
492         return x;
493     }
494 }
495 float Layer::derivative_activation_function(float x)
496 {
497     switch (activation_type)
498     {
499     case _sigmoid:
500         return activation_function(x) * (1 - activation_function(x));
501         // return x*(1-x);
502     case _tanh:
503         return 1 - activation_function(x) * activation_function(x);
504     default:
505         return x;
506     }
507 }
```

4.2 SIMD

SIMD 将基础数据向量化，每一条指令同时计算 4 个（SSE）或 8 个（AVX）数据。具体地，以 SSE 优化前向传播为例，对于每个隐层节点的计算，可以将一个节点中遍历的向量的每四个数据打包成一个 ___m128 变量, 然后调用 SSE 的 API 进行具体的计算。其在 ANN 中的优化主要体现在 5 个主要循环计算部分，的矩阵行或列的任务分配。由于代码过长，与最终版本类似，具体可以参见最终版本。

4.3 OpenMP

OpenMP 是一种用于共享内存并行系统的多线程程序设计方案，OpenMP 提供对并行算法的高层抽象描述，特别适合在多核 CPU 机器上的并行程序设计。OpenMP 采用 fork-join 的执行模式，开始的时候只存在一个主线程，当需要进行并行计算的时候，派生出若干个分支线程来执行并行任务。当并行代码执行完成之后，分支线程会合，并把控制流程交给单独的主线程。其在 ANN 中的优化主要体现在 5 个主要循环计算部分，的矩阵行或列的任务分配。由于代码过长，与最终版本类似，具体可以参见最终版本。

4.4 MPI

OpenMP 提供了线程级别的并行，可以在一台主机上执行多个线程同时完成一个任务以达到并行加速的效果。但更通用的并行是需要能够在多台主机上整合多个节点的计算能力执行同一个任务，这显然不是一个单独的进程能够实现的了。在这种情况下，MPI 提供了进程级别的并行，不同的进程可以在不同的物理节点上运行，这样能够充分利用整个网络上的计算资源。由于涉及到多进程，不同的进程有着不同的地址和存储空间，无法直接相互访问，那么如何在不同进程之间合理而高效地传输数据就成了 MPI 并行的关键。此外，为了保证程序运行的正确性，进程之间的同步也非常重要。

之前的 MPI 实验细粒度得考虑了隐层中不同节点的计算，将不同节点的计算划分给不同的进程，同时使用一个特定的进程来进行结果的接收和同步。但是在本次实验中，算法使用了小批量梯度下降方法，那么在同一批次中不同样本之间的损失函数和梯度的计算是相互独立的，可以在更高的样本粒度上对数据进行划分，将同一批次中的样本分配给不同的进程计算。本次实验分别使用了对等模式和主从模式两种 MPI 编程模式来比较样本粒度并行的加速效果。

4.4.1 对等模式

在对等模式下，所有的进程都参与 ANN 中“前向传播”、“反向传播”、“更新权重”三个部分的计算。在 ANN 小批量梯度更新中，不同批次之间是串行的。那么对任务的分配也就可以聚焦到同一批次的不同样本上。需要注意的是，每个进程只负责批次中一部分样本的梯度计算，但之后的参数更新需要用到该批次中所有样本的计算结果，所以需要一个特定的进程来进行结果的接收和同步。由于代码过长，与最终版本类似，具体可以参见最终版本。

4.4.2 主从模式

在对等模式 ANN 并行中，每个进程根据自己的编号和总体进程数目静态地确定自己的计算任务，最后由 0 号进程进行结果的收集和同步。但是这样的划分任务粒度较大，无法将计算任务按需分配给各个进程，可能产生由于负载不均或者不同进程计算速度不一而导致的空闲等待问题。针对这个问题，可以使用主从模式对任务进行动态分配。在主从模式中，0 号进程不再参与运算，而是将任务划分成更加细粒度单个样本，为其他进程进行实时的任务分配与结果收集。

具体地，0 号进程首先给其他节点各发送一个样本，之后进入循环等待接收状态，一旦接收到了某个进程发送过来的计算结果，则根据消息中的 MPI_TAG 判断该结果对应的是哪个样本并进行存储。接收完毕后，如果仍有未计算的样本，则给该空闲进程发送该样本。重复以上步骤直到该批次内的所有样本点都被计算完毕。最后 0 号进程会将整合好的计算结果广播给其他节点，进入下一批次的计算。由于代码过长，与最终版本类似，具体可以参见最终版本。

4.5 MPI + OpenMP + SIMD

MPI 提供了进程级别的并行，OpenMP 提供了线程级别的并行，SIMD 提供了指令级别的并行，它们的粒度各不相同，可以很好地结合起来。需要注意的是在 MPI 程序中使用多线程时需要注意 MPI 定义的四种安全级别：

- MPI_THREAD_SINGLE：应用中只有一个线程
- MPI_THREAD_FUNNELED：多线程，但只有主线程会进行 MPI 调用（调用 MPI_Init_thread 的那个线程）

- MPI_THREAD_SERIALIZED：多线程，但同时只有一个线程会进行 MPI 调用
- MPI_THREAD_MULTIPLE：多线程，且任何线程任何时候都会进行 MPI 调用（有一些限制避免竞争条件）

在该问题当中，每个进程将分配到到若干个需要计算的样本，这个时候每个进程可以使用 OpenMP 创建多个线程，将这些样本前向传播、反向传播、梯度更新等实际计算任务分配给不同的线程去计算，因为在线程的计算中并不会进行 MPI 调用，因此程序属于 MPI_THREAD_FUNNELED 的安全级别。在最终的版本中，使用 MPI 对等模式静态块划分进行同一 batc 和、内不同样本的分配，使用 OPENMP 与 SIMD 进行一个样本的前向传播、导数更新、反向传播循环内不同神经网络节点的运算划分。MPI、OpenMP 和 SIMD 方法结合的代码如下：

MPI + OpenMP + SIMD

```
void ANN_parallel::train_MPI_openMP_SIMD(const int num_sample, float** _trainMat,
    float** _labelMat)
{
  std::printf("begin training\n");
  long long head, tail, freq;// timers
  QueryPerformanceFrequency((LARGE_INTEGER*)&freq);
  QueryPerformanceCounter((LARGE_INTEGER*)&head); // start time

  float thre = 1e-2;
  int myid, numprocs;
  MPI_Comm_rank(MPI_COMM_WORLD, &myid);
  MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
  for (int epoch = 0; epoch < num_epoch; epoch++)
  {
    //if (myid == 0) isNotConver_(num_sample, _trainMat, _labelMat, thre);
    //fflush(stdout);
    // if (epoch % 50 == 0) printf ("round%d:\n", epoch);
    for (int batch_i = 0; batch_i < (num_sample + batch_size - 1) / batch_size;
        batch_i++) {
      if (myid == 0) {
        for (int i = num_layers; i >= 0; i--) {
          for (int j = 0; j < num_each_layer[i + 1]; j++) {
            memset(layers[i]->weights_delta[j], 0, num_each_layer[i]);
          }
          memset(layers[i]->bias_delta, 0, num_each_layer[i + 1]);
        }
      }
      //cout << myid << "is waiting in bari1" << endl;
      //MPI_Barrier(MPI_COMM_WORLD);
      //cout << myid << "after bari1" << endl;
      int t_batch_size = min((batch_i + 1) * batch_size, num_sample) - batch_i *
          batch_size;
      if (t_batch_size == 0)cout << "t_batch_szie wrong!!" << endl;

      int my_size = (t_batch_size + numprocs - 1) / numprocs;
      //
```

```
34        for (int index = batch_i * batch_size + my_size * myid;
35          index < min(batch_i * batch_size + t_batch_size, batch_i * batch_size +
                my_size * (myid + 1));
36          index++) {
37          //cout << myid << "index" <<index<< endl;
38          //数据发送
39          for (int i_layer = 0; i_layer <= num_layers; i_layer++) {
40            MPI_Bcast(layers[i_layer]->bias, num_each_layer[i_layer + 1], MPI_FLOAT,
                  0, MPI_COMM_WORLD);
41            for (int i = 0; i < num_each_layer[i_layer + 1]; i++)
42              MPI_Bcast(layers[i_layer]->weights[i], num_each_layer[i_layer],
                    MPI_FLOAT, 0, MPI_COMM_WORLD);
43          }
44          //cout <<myid<< "begin predict" << endl;
45                  //前向传播
46  #pragma omp parallel num_threads(NUM_THREADS)
47  #pragma omp for
48          for (int i = 0; i < num_each_layer[1]; i++)
49          {
50            layers[0]->output_nodes[i] = 0.0;
51            //printf("%d,%d ",i,class_p->num_each_layer[1]);
52            int j = 0;
53            __m128 ans = _mm_setzero_ps();
54            for (; j + 4 < num_each_layer[0]; j += 4)
55            {
56              __m128 t1, t2;
57              //将内部循环改为4位的向量运算
58              t1 = _mm_loadu_ps(layers[0]->weights[i] + j);
59              t2 = _mm_loadu_ps(_trainMat[index] + j);
60              t1 = _mm_mul_ps(t1, t2);
61
62              ans = _mm_add_ps(ans, t1);
63            }
64            // 4个局部和相加
65            ans = _mm_hadd_ps(ans, ans);
66            ans = _mm_hadd_ps(ans, ans);
67            //标量存储
68            float z;
69            _mm_store_ss(&z, ans);
70            for (j -= 4; j < num_each_layer[0]; j++) {
71              z += layers[0]->weights[i][j] * _trainMat[index][j];
72            }
73            layers[0]->output_nodes[i] += z;
74            layers[0]->output_nodes[i] += layers[0]->bias[i];
75            layers[0]->output_nodes[i] =
                  layers[0]->activation_function(layers[0]->output_nodes[i]);
76          }
77          for (int layers_i = 1; layers_i <= num_layers; layers_i++)
78          {
```

```
79  #pragma omp parallel num_threads(NUM_THREADS)
80  #pragma omp for
81            for (int i = 0; i < num_each_layer[layers_i + 1]; i++)
82            {
83              layers[layers_i]->output_nodes[i] = 0.0;
84              int j = 0;
85              __m128 ans = _mm_setzero_ps();
86              for (; j + 4 < num_each_layer[layers_i]; j += 4)
87              {
88                __m128 t1, t2;
89                //将内部循环改为4位的向量运算
90                t1 = _mm_loadu_ps(layers[layers_i]->weights[i] + j);
91                t2 = _mm_loadu_ps(layers[layers_i - 1]->output_nodes + j);
92                t1 = _mm_mul_ps(t1, t2);
93
94                ans = _mm_add_ps(ans, t1);
95              }
96              // 4个局部和相加
97              ans = _mm_hadd_ps(ans, ans);
98              ans = _mm_hadd_ps(ans, ans);
99              //标量存储
100             float z;
101             _mm_store_ss(&z, ans);
102             for (j -= 4; j < num_each_layer[layers_i]; j++) {
103               z += layers[layers_i]->weights[i][j] * layers[layers_i -
                      1]->output_nodes[j];
104             }
105             layers[layers_i]->output_nodes[i] += z;
106             layers[layers_i]->output_nodes[i] += layers[layers_i]->bias[i];
107             layers[layers_i]->output_nodes[i] =
                    layers[layers_i]->activation_function(layers[layers_i]->output_nodes[i]);
108           }
109         }
110         //cout <<myid<< "begin cal delta" << endl;
111         //计算delta
112         MPI_Status status;
113         //cout << MAX_NUM_LAYER << endl;
114         //for (int i = 0; i < num_layers + 2; i++)cout << num_each_layer[i] << ' ';
115         //float* temp_bias_delta = new float[MAX_NUM_LAYER];
116         float* temp_bias_delta = (float*)malloc(MAX_NUM_LAYER * sizeof(float));
117         //float* temp_weights_delta = new float[MAX_NUM_LAYER];
118         float* temp_weights_delta = (float*)malloc(MAX_NUM_LAYER * sizeof(float));
119
120         int pro_recv = numprocs;
121         if (batch_i * batch_size + t_batch_size < batch_i * batch_size + my_size *
                numprocs) pro_recv--;
122
123         for (int j = 0; j < num_each_layer[num_layers + 1]; j++)
124         {
```

```
125             //均方误差损失函数
126             temp_bias_delta[j] = (layers[num_layers]->output_nodes[j] -
                    _labelMat[index][j]) *
                    layers[num_layers]->derivative_activation_function(layers[num_layers]->output_node
                    //输出0到1
127             //layers[num_layers]->bias_delta[j] += temp_bias_delta[j];
128             //cout << myid << "cal delta@" << j << endl;
129             //交叉熵损失函数
130             //layers[num_layers]->delta[j] += (layers[num_layers]->output_nodes[j] -
                    _labelMat[index][j]);
131
132             //cout << myid << "cal delta!" << j << endl;
133             if (myid != 0)
134               MPI_Send(temp_weights_delta, num_each_layer[num_layers], MPI_FLOAT, 0,
                      100 + j, MPI_COMM_WORLD);
135             else {
136               int k = 0;
137               __m128 t2 = _mm_set1_ps(temp_bias_delta[j]);
138 #pragma omp parallel num_threads(NUM_THREADS)
139 #pragma omp for private(k)
140               for (k = 0; k< num_each_layer[num_layers]-4; k += 4)
141               {
142                 __m128 t1, t3, product;
143                 t1 = _mm_loadu_ps(layers[num_layers]->weights_delta[j] + k);
144                 t3 = _mm_loadu_ps(layers[num_layers - 1]->output_nodes + k);
145                 //product=sr*t2*t3, 向量对位相乘
146                 product = _mm_mul_ps(t3, t2);
147                 t1 = _mm_add_ps(t1, product);
148                 _mm_storeu_ps(layers[num_layers]->weights_delta[j] + k, t1);
149               }
150               for (k -= 4; k < num_each_layer[num_layers]; k++)
151                 layers[num_layers]->weights_delta[j][k] += temp_bias_delta[j] *
                        layers[num_layers - 1]->output_nodes[k];
152
153               //for (int k = 0; k < num_each_layer[num_layers]; k++)
154               //  layers[num_layers]->weights_delta[j][k] += temp_weights_delta[k];
155               for (int pro_i = 1; pro_i < pro_recv; pro_i++) {
156                 MPI_Recv(temp_weights_delta, num_each_layer[num_layers], MPI_FLOAT,
                        MPI_ANY_SOURCE, 100 + j, MPI_COMM_WORLD, &status);
157                 int k = 0;
158                 __m128 t2 = _mm_set1_ps(temp_bias_delta[j]);
159 #pragma omp parallel num_threads(NUM_THREADS)
160 #pragma omp for private(k)
161                 for (k = 0; k< num_each_layer[num_layers]-4; k += 4)
162                 {
163                   __m128 t1, t3, product;
164                   t1 = _mm_loadu_ps(layers[num_layers]->weights_delta[j] + k);
165                   t3 = _mm_loadu_ps(layers[num_layers - 1]->output_nodes + k);
166                   //product=sr*t2*t3, 向量对位相乘
```

```
167                 product = _mm_mul_ps(t3, t2);
168                 t1 = _mm_add_ps(t1, product);
169                 _mm_storeu_ps(layers[num_layers]->weights_delta[j] + k, t1);
170               }
171               for (k -= 4; k < num_each_layer[num_layers]; k++)
172                 layers[num_layers]->weights_delta[j][k] += temp_bias_delta[j] *
                      layers[num_layers - 1]->output_nodes[k];
173
174               //for (int k = 0; k < num_each_layer[num_layers]; k++)
175               //  layers[num_layers]->weights_delta[j][k] += temp_weights_delta[k];
176             }
177           }
178           //MPI_Reduce(temp_weights_delta, layers[num_layers]->weights_delta[j],
                  num_each_layer[num_layers], MPI_FLOAT, MPI_SUM, 0, MPI_COMM_WORLD);
179           // cout << myid << "cal delta#" << j << endl;
180         }
181         if (myid != 0)
182           MPI_Send(temp_bias_delta, num_each_layer[num_layers + 1], MPI_FLOAT, 0,
                  2, MPI_COMM_WORLD);
183         else {
184 #pragma omp parallel num_threads(NUM_THREADS)
185 #pragma omp for
186           for (int k = 0; k < num_each_layer[num_layers + 1]; k++)
187             layers[num_layers]->bias_delta[k] += temp_bias_delta[k];
188           for (int pro_i = 1; pro_i < pro_recv; pro_i++) {
189             MPI_Recv(temp_bias_delta, num_each_layer[num_layers + 1], MPI_FLOAT,
                    MPI_ANY_SOURCE, 2, MPI_COMM_WORLD, &status);
190 #pragma omp parallel num_threads(NUM_THREADS)
191 #pragma omp for
192             for (int k = 0; k < num_each_layer[num_layers + 1]; k++)
193               layers[num_layers]->bias_delta[k] += temp_bias_delta[k];
194           }
195         }
196         //MPI_Reduce(temp_bias_delta, layers[num_layers]->bias_delta,
                num_each_layer[num_layers + 1], MPI_FLOAT, MPI_SUM, 0, MPI_COMM_WORLD);
197         //cout << myid << "finish cal delta"<< num_layers << endl;
198
199         //计算每层的delta,行访问优化
200         for (int i = num_layers - 1; i >= 0; i--)
201         {
202           float* error = new float[num_each_layer[i + 1]];
203
204           for (int j = 0; j < num_each_layer[i + 1]; j++) error[j] = 0.0;
205 #pragma omp parallel num_threads(NUM_THREADS)
206 #pragma omp for
207           for (int k = 0; k < num_each_layer[i + 2]; k++)
208           {
209             int j = 0;
210             __m128 t2 = _mm_set1_ps(temp_bias_delta[k]);
```

```
211             for (; j + 4 < num_each_layer[i + 1]; j += 4)
212             {
213               __m128 t1, t3, product;
214               t1 = _mm_loadu_ps(error + j);
215               t3 = _mm_loadu_ps(&layers[i + 1]->weights[k][j]);
216               product = _mm_mul_ps(t3, t2);
217               t1 = _mm_add_ps(t1, product);
218               _mm_storeu_ps(error + j, t1);
219             }
220             for (j -= 4; j < num_each_layer[i + 1]; j++) {
221               error[j] += layers[i + 1]->weights[k][j] * temp_bias_delta[k];
222             }
223           }
224           for (int j = 0; j < num_each_layer[i + 1]; j++)
225           {
226             temp_bias_delta[j] = error[j] *
                  layers[num_layers]->derivative_activation_function(layers[i]->output_nodes[j]);
227             layers[i]->bias_delta[j] += temp_bias_delta[j];
228             int k = 0;
229             __m128 t2 = _mm_set1_ps(temp_bias_delta[j]);
230 #pragma omp parallel num_threads(NUM_THREADS)
231 #pragma omp for private(k)
232             for (k = 0; k< num_each_layer[i]-4; k += 4)
233             {
234               __m128 t1, t3, product;
235               if (i == 0) t3 = _mm_loadu_ps(&_trainMat[index][k]);
236               else t3 = _mm_loadu_ps(&layers[i - 1]->output_nodes[k]);
237               product = _mm_mul_ps(t3, t2);
238               _mm_storeu_ps(&temp_weights_delta[k], product);
239             }
240
241             for (k -= 4; k < num_each_layer[i]; k++)
242               if (i == 0) temp_weights_delta[k] = temp_bias_delta[j] * layers[i -
                    1]->output_nodes[k];
243               else temp_weights_delta[k] = temp_bias_delta[j] *
                    _trainMat[index][k];
244
245             if (myid != 0)
246               MPI_Send(temp_weights_delta, num_each_layer[i], MPI_FLOAT, 0, 100 +
                    MAX_NUM_LAYER + i * MAX_NUM_LAYER + j, MPI_COMM_WORLD);
247             else {
248               int k = 0;
249 #pragma omp parallel num_threads(NUM_THREADS)
250 #pragma omp for private(k)
251               for (k=0; k< num_each_layer[i]-4; k += 4)
252               {
253                 __m128 t1, t3;
254                 t3 = _mm_loadu_ps(&temp_weights_delta[k]);
255                 t1 = _mm_loadu_ps(&layers[i]->weights_delta[j][k]);
```

```
256                 t1 = _mm_add_ps(t3, t1);
257                 _mm_storeu_ps(&layers[i]->weights_delta[j][k], t1);
258               }
259               for (k -= 4; k < num_each_layer[i]; k++)
260                 layers[i]->weights_delta[j][k] += temp_weights_delta[k];
261
262               for (int pro_i = 1; pro_i < pro_recv; pro_i++) {
263                 //cout << sizeof(temp_weights_delta) << endl;
264                 MPI_Recv(temp_weights_delta, num_each_layer[i], MPI_FLOAT,
                        MPI_ANY_SOURCE, 100 + MAX_NUM_LAYER + i * MAX_NUM_LAYER + j,
                        MPI_COMM_WORLD, &status);
265                 int k = 0;
266 #pragma omp parallel num_threads(NUM_THREADS)
267 #pragma omp for private(k)
268                 for (k=0; k< num_each_layer[i]-4; k += 4)
269                 {
270                   __m128 t1, t3;
271                   t3 = _mm_loadu_ps(&temp_weights_delta[k]);
272                   t1 = _mm_loadu_ps(&layers[i]->weights_delta[j][k]);
273                   t1 = _mm_add_ps(t3, t1);
274                   _mm_storeu_ps(&layers[i]->weights_delta[j][k], t1);
275                 }
276                 for (k -= 4; k < num_each_layer[i]; k++)
277                   layers[i]->weights_delta[j][k] += temp_weights_delta[k];
278               }
279             }
280             //MPI_Reduce(temp_weights_delta, layers[i]->weights_delta[j],
                    num_each_layer[i], MPI_FLOAT, MPI_SUM, 0, MPI_COMM_WORLD);
281           }
282           delete[] error;
283
284           if (myid != 0)
285             MPI_Send(temp_bias_delta, num_each_layer[i + 1], MPI_FLOAT, 0, 3 + i,
                    MPI_COMM_WORLD);
286           else {
287 #pragma omp parallel num_threads(NUM_THREADS)
288 #pragma omp for
289             for (int k = 0; k < num_each_layer[i + 1]; k++)
290               layers[i]->bias_delta[k] += temp_bias_delta[k];
291             for (int pro_i = 1; pro_i < pro_recv; pro_i++) {
292               MPI_Recv(temp_bias_delta, num_each_layer[i + 1], MPI_FLOAT,
                      MPI_ANY_SOURCE, 3 + i, MPI_COMM_WORLD, &status);
293 #pragma omp parallel num_threads(NUM_THREADS)
294 #pragma omp for
295               for (int k = 0; k < num_each_layer[i + 1]; k++)
296                 layers[i]->bias_delta[k] += temp_bias_delta[k];
297             }
298           }
299           //MPI_Reduce(temp_bias_delta, layers[i]->bias_delta, num_each_layer[i +
```

```
                1], MPI_FLOAT, MPI_SUM, 0, MPI_COMM_WORLD);
300           //cout << myid << "finish cal delta" <<i<< endl;
301         }
302         std::free(temp_weights_delta);
303         std::free(temp_bias_delta);
304         //cout << myid << "finish cal delta" << endl;
305       }
306
307     //cout << myid << "is waiting in bari2" << endl;
308
309     MPI_Barrier(MPI_COMM_WORLD);
310     //cout << myid << "after bari2" << endl;
311     if (myid == 0) {
312       //反向传播, weights和bias更新
313       for (int i = 0; i <= num_layers; i++)
314       {
315 #pragma omp for
316         for (int k = 0; k < num_each_layer[i + 1]; k++)
317         {
318           __m128 r = _mm_set1_ps(study_rate / t_batch_size);
319           int j = 0;
320           for (; j + 4 < num_each_layer[i]; j += 4) {
321             __m128 t1, t3, product;
322             t1 = _mm_loadu_ps(&layers[i]->weights[k][j]);
323             t3 = _mm_loadu_ps(&layers[i]->weights_delta[k][j]);
324             product = _mm_mul_ps(t3, r);
325             t1 = _mm_sub_ps(t1, product);
326             _mm_storeu_ps(&layers[i]->weights[k][j], t1);
327           }
328           for (j -= 4; j < num_each_layer[i]; j++) {
329             layers[i]->weights[k][j] -= study_rate *
                  layers[i]->weights_delta[k][j] / t_batch_size;
330           }
331           layers[i]->bias[k] -= study_rate * layers[i]->bias_delta[k] /
                t_batch_size;
332         }
333       }
334     }
335     }
336   }
337
338   QueryPerformanceCounter((LARGE_INTEGER*)&tail); // end time
339   if (myid == 0) {
340     std::cout << myid << "mpi+simd+openmp:" << (tail - head) * 1.0 / freq << "s"
          << endl;
341     std::printf("finish training\n");
342   }
343
344 }
```

5 实验和结果分析

实验部分分析了问题规模对串行和并行算法的影响,分析了不同进程(线程)数目对 MPI(OpenMP)程序运行时间的影响，分析了 SIMD 方法对程序运行时间的影响。此外，MPI、OpenMP 和 SIMD 三种方法还可以相互结合，实验探讨了串行、SIMD+OpenMP 以及 MPI+OpenMP+SIMD 的运行时间差异。

由于服务器所限，所有实验均在 x86、windows 架构下进行。由于本机是 AMD 架构，无法使用 Vtune 进行分析，故仅进行了计时和讨论。此外，本机为 6 核 6 线程，即加速的上限。

5.1 串行分析

5.1.1 问题规模分析

理论上来说，不同隐层之间节点数量是否相同不太会影响模型的表现结果，不失一般性，假设所有隐层的节点数量均为 n，实验探究了隐藏层层数与隐藏层节点数对模型表现和运行时间的影响。实验结果如表2、3、4、5所示。其中 epoch 数均设置为 30，batchsize 均设置为 12，学习率设置为 0.01. 由实验结果可以看出，在相同 epoch 数的情况下，准确率先增加后下降。增加的原因是，随着隐层数核隐层维度的增加，模型的表征能力增加。下降的原因是，由于 epoch 数所限，在模型参数量过大的情况下，模型尚未收敛。此外，在实验过程中发现，在参数量较大时，模型出现了梯度爆炸的现象，这可能与具体的模型、超参数的选择有关。根据实验结果，在接下来的实验中，取隐层层数为 2，隐层维度均为 32，在 2 个数据集上进行实验。

表 2: 鸢尾花数据集上不同隐层参数下准确率

隐藏层层数	n=32	n=64	n=96	n=128
1	0.875	0.916667	1	0.916667
2	0.958333	0.916667	1	1
3	0.666667	1	0.875	0.791667
4	1	0.916667	0.708333	0.625

表 3: 乳腺癌数据集上不同隐层参数下准确率

隐藏层层数	n=32	n=64	n=96	n=128
1	0.928571	0.921429	0.885714	0.9
2	0.678571	0.907143	0.914286	0.885714
3	0.771429	0.842857	0.878571	0.714286
4	0.842857	0.692857	0.664286	0.635714

表 4: 鸢尾花数据集上不同隐层参数下运行时间（单位：s）

隐藏层层数	n=32	n=64	n=96	n=128
1	0.136677	0.223789	0.329134	0.438298
2	0.597874	2.07027	4.28894	7.39954
3	1.15133	3.86819	8.37028	14.5643
4	1.66993	5.60371	12.5653	22.2738

表 5: 乳腺癌数据集上不同隐层参数下运行时间（单位：s）

隐藏层层数	n=32	n=64	n=96	n=128
1	0.801159	1.503	2.28771	2.89591
2	4.95341	13.715	30.1261	47.8367
3	6.31847	22.74	49.3724	86.0874
4	9.01768	33.356	71.9907	125.879s

5.1.2 批大小分析

在小批次梯度下降中，一个批次中的所有样本都会进行前向传播和反向传播，但每个批次只会执行一次参数更新。所以理论上来说，批次的大小也会影响程序的运行时间。实验设置 ANN 隐藏层数为 2，每层节点数相同为表中 n 所示，epoch 设置为 30，使用不同的批大小测试了程序的运行时间，实验结果如表6所示，可以看到在相同的规模下，随着 batchsize 增大，程序运行时间减少，这是因为网络的全部参数更新为每个 batch 一次，所以符合预期。而实际上由于前向传播、导数更新每个循环内部的时间复杂度都为 O(n*n)，故在总体上应当仅体现为常数的变化，且仅在 batch 规模较小时比较明显，实验结果也证明了这一点。在接下来的实验中，对不同的样本进行并行化，故取 batchsize 为 12 不变。

表 6: 乳腺癌数据集上不同批大小下运行时间（单位：s）

n	bs=1	bs=8	bs=16	bs=32	bs=64	bs=128
32	5.3384	3.58139	3.46938	3.53268	3.56893	3.44663
64	18.4349	12.5895	12.0596	11.5883	11.6511	11.5024
96	39.2763	26.2946	25.3394	24.7713	24.3951	24.2793
128	68.718	45.0818	43.8394	42.7091	42.1679	41.9678

5.2 SIMD

SIMD 优化使用了 SSE，将矩阵运算中的每四个数据打包成一个向量，使用 SSE 提供的接口来进行向量的乘法和加法等运算。SIMD 方法在不同 ANN 规模下，在乳腺癌数据集上的运行时间如表7所示，在鸢尾花数据集上的运行时间如表8所示，可以看到，与上述串行时间相比，单使用 SSE 4 路向量化在两个数据集上，均达到了接近 3 倍的加速比。由于程序中串行部分的影响，这是比较合理的。

表 7: 鸢尾花数据集上 SIMD 的运行时间（单位：s）

隐藏层层数	n=32	n=64	n=96	n=128
2	0.40185	1.04903	11.9668	3.24992

5.3 OpenMP

为了测试 OpenMP 中线程数量对程序加速效果的影响，实验固定了 ANN 隐层层数为 2，测试了在两个数据集上，不同线程数和不同问题规模下的运行时间，其结果如表9,10所示，与串行程序进行对比，达到了接近线程数的加速比，而随着线程数的增加，加速比更加接近线程数。值得注意的是，由于本机为 6 线程，故最大实现 6 倍的加速比，故可以看到，当线程数设置为 8 时，程序性能明显下降，这可能是由于额外的线程开销所致。

表 8: 乳腺癌数据集上 SIMD 的运行时间（单位：s）

隐藏层层数	n=32	n=64	n=96	n=128
2	2.31557	6.27715	11.9668	19.3921

表 9: 乳腺癌数据集上 OpenMP 的运行时间（单位：s）

问题规模 n	2 线程	4 线程	6 线程	8 线程
32	2.30052	2.10342	1.9776	2.88899
64	6.95326	5.21216	4.57849	6.87771
96	14.1842	9.99308	7.94555	12.2592
128	23.7609	16.222	12.604	18.5547

将 openMP 与 SIMD 结合，测试了在两个数据集上，不同线程数和不同问题规模下的运行时间，其结果如表11,12所示，SIMD 最多实现 4 倍加速，当线程数为 4 时，理论上最多得到 16 倍的加速比，实际结合得到约 7 倍的加速比；当线程数为 6 时，理论上最多得到 24 倍的加速比，实际结合得到约 12 倍的加速比。结合 SIMD 与 openMP 单独的实验可知，由于二者单独都无法达到理想的加速比，故将其结合后结果是合理的。

5.4 MPI

MPI 方法与 OpenMP 方法类似，会受到进程数量的影响，因此实验同样固定 ANN 隐藏层层数为 2，epoch 数为 30，测试 MPI 方法在不同进程数和问题规模下的运行时间。

5.4.1 对等模式

对等模式下所有进程都参与计算任务，并且是根据自身的进程 ID 获取对应的计算任务。对等模式 MPI 在两个数据集上的运行时间如表13,14所示，可以看到，结果并不十分理想，加速比远远达不到进程数，而在规模较小时甚至与串行算法接近。

于是结合代码进行分析，发现在多进程间进行数据传输时，确实用到了较大的额外开销。针对上述问题，进行了进一步实验，测试了单个样本进行一次前向传播与导数计算所用时间，并与串行算法单个样本的计算时间进行比较。以鸢尾花数据集为例，结果如表15所示。可以发现，增加了进程间数据交换的单样本计算，计算量相较于串行增加了将近 3 倍，故即使再将其分配到多个进程，加速比也无法达到预期，这就解释了 MPI 结果出现的原因。

表 10: 鸢尾花数据集上 OpenMP 的运行时间（单位：s）

问题规模 n	2 线程	4 线程	6 线程	8 线程
32	0.428727	0.508615	0.422552	0.584846
64	1.13032	0.885499	0.965929	1.2234
96	2.40648	1.65642	1.60137	1.99696
128	4.02745	2.61956	2.11826	3.23934

表 11: 乳腺癌数据集上 OpenMP+SIMD 的运行时间（单位：s）

问题规模 n	2 线程	4 线程	6 线程	8 线程
32	1.38595	1.14974	0.9875	1.2708
64	3.52445	2.29064	1.9119	2.55639
96	6.39568	4.13609	3.07235	4.13102
128	10.1481	6.30379	4.46709	6.10468

表 12: 鸢尾花数据集上 OpenMP+SIMD 的运行时间（单位：s）

问题规模 n	2 线程	4 线程	6 线程	8 线程
32	0.257012	0.230649	0.318293	0.37648
64	0.566553	0.391693	0.401552	0.439661
96	1.13786	0.644662	0.52826	0.708284
128	1.7318	1.04083	0.768676	1.09097

表 13: 乳腺癌数据集上对等模式 MPI 的运行时间（单位：s）

问题规模 n	2 进程	4 进程
32	3.82212	3.07936
64	12.1525	9.75421
96	23.9293	21.0579
128	37.8392	32.3566

表 14: 鸢尾花数据集上对等模式 MPI 的运行时间（单位：s）

问题规模 n	2 进程	4 进程	6 进程
32	0.410821	0.326436	0.888699
64	1.30687	1.00205	1.56343
96	2.72229	2.10132	2.04111
128	4.59648	3.96232	3.64631

表 15: 鸢尾花数据集单个样本运行时间（单位：s）

问题规模 n	MPI（6 进程）	MPI（4 进程）	MPI（2 进程）	串行
32	0.002041	0.0004266	0.0002675	0.000185
64	0.0019868	0.0013532	0.0008575	0.0006394
96	0.0037551	0.0027039	0.0017642	0.0013854
128	0.0073411	0.0051354	0.0030596	0.0023772

5.4.2 主从模式

主从模式下，0 号进程不再参与计算任务，而是为其他的进程动态分配需要计算的样本，并循环地接收结果。主从模式 MPI 在两个数据集上的运行时间如表16,17所示，可以发现样本主从模式的分配方式达到的结果与对等模式相似，但略有下降，可能是如下两个原因：主进程不参与计算，使每个进程计算的任务量增加。细粒度的数据传输的额外开销。因此，在最终结合的实验中，采用对等模式划分进行多线程、向量化的结合。

表 16: 乳腺癌数据集上主从模式 MPI 的运行时间（单位：s）

问题规模 n	2 进程	4 进程
32	3.9864	3.9865
64	12.6535	9.9631
96	23.9343	21.5529
128	38.3424	33.4266

表 17: 鸢尾花数据集上主从模式 MPI 的运行时间（单位：s）

问题规模 n	2 进程	4 进程	6 进程
32	0.49642	0.34123	0.87333
64	1.39766	1.3203	1.59772
96	2.8862	2.3032	2.0578
128	4.97662	4.0772	3.7088

5.5 MPI+OpenMP+SIMD 混合编程

实验的最后部分将多种并行优化结合，给出一个较优的 ANN 并行解决方案。实验设置 MPI 进程数为 4，采用每个进程 6 线程的设置。对于 SIMD，实验使用 SSE 指令。串行与混合编程的实验结果如表18所示，可以看到相比于多线程与 SIMD 的结合，速度进一步略有提升，达到了 ANN 训练目前的最优效果。

表 18: 串行与最终混合编程方式的运行时间 (单位：s)

问题规模 n	串行	MPI+OpenMP+SIMD
32	5.0341	0.7603
64	13.4523	1.4148
96	30.0098	2.5672
128	47.98822	3.4411

6 总结

本文主要调研、研究和实现了 ANN 问题的并行优化算法。首先，对于 ANN 问题本身，它是由前向传播、反向传播和参数更新几个步骤组成的密集计算型任务，比较适合并行优化。本文具体介绍了它的串行算法设计，并且分析了他的时空复杂度。然后，本文介绍了前人的算法，并将 MPI、openMP、SIMD 应用于 ANN 问题的求解，通过实验验证了在一定范围内程序运行时间和线程数、进程数量的

线性关系，这也证实了向量化、多线程、多进程在提升程序效率上的实用性。最后本文将进程粒度的 MPI、线程粒度的 OpenMP 和指令集粒度的 SIMD 结合起来，对 ANN 问题探究出了一套综合较优的并行方案。

并行的技术不是仅限于 MPI、OpenMP、Pthread、SIMD 和 CUDA，盲目地套用往往会适得其反，从整个的实验过程来看，必须要针对实际问题的特点，分析限制运行时间的关键部分和因素，对症下药，采取合适的策略才能得到好的加速效果。

项目源代码链接 https://github.com/AldebaranL/Parallel-programming-Homework

参考文献

[1] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020.

[2] P Chanthini and K Shyamala. A survey on parallelization of neural network using mpi and open mp. *Indian Journal of Science and Technology*, 9(19), 2016.

[3] George Dahl, Alan McAvinney, Tia Newhall, et al. Parallelizing neural network training for cluster systems. In *Proceedings of the IASTED international conference on parallel and distributed computing and networks*, pages 220–225. ACTA Press, 2008.

[4] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Dehao Chen, Mia Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V Le, Yonghui Wu, et al. Gpipe: Efficient training of giant neural networks using pipeline parallelism. *Advances in neural information processing systems*, 32, 2019.

[5] Altaf Ahmad Huqqani, Erich Schikuta, Sicen Ye, and Peng Chen. Multicore and gpu parallelization of neural networks for face recognition. *Procedia Computer Science*, 18:349–358, 2013.

[6] Haoqiang Jin, Dennis Jespersen, Piyush Mehrotra, Rupak Biswas, Lei Huang, and Barbara Chapman. High performance computing using mpi and openmp on multi-core parallel systems. *Parallel Computing*, 37(9):562–575, 2011.

[7] Vijay Korthikanti, Jared Casper, Sangkug Lym, Lawrence McAfee, Michael Andersch, Mohammad Shoeybi, and Bryan Catanzaro. Reducing activation recomputation in large transformer models. *arXiv preprint arXiv:2205.05198*, 2022.

[8] Lyle N Long and Ankur Gupta. Scalable massively parallel artificial neural networks. *Journal of Aerospace Computing, Information, and Communication*, 5(1):3–15, 2008.

[9] Qutaibah M. Malluhi, Magdy A. Bayoumi, and TRN Rao. Efficient mapping of anns on hypercube massively parallel machines. *IEEE Transactions on Computers*, 44(6):769–779, 1995.

[10] Raul Puri Mostofa Patwary. Gtc 2020: Megatron-lm: Training multi-billion parameter language models using model parallelism. [EB/OL]. https://developer.nvidia.com/gtc/2020/video/s21496 Accessed June 30, 2022.

[11] Deepak Narayanan, Mohammad Shoeybi, Jared Casper, Patrick LeGresley, Mostofa Patwary, Vijay Korthikanti, Dmitri Vainbrand, Prethvi Kashinkunti, Julie Bernauer, Bryan Catanzaro, et al. Efficient large-scale language model training on gpu clusters using megatron-lm. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–15, 2021.

[12] Rolf Rabenseifner, Georg Hager, and Gabriele Jost. Hybrid mpi/openmp parallel programming on clusters of multi-core smp nodes. In *2009 17th Euromicro international conference on parallel, distributed and network-based processing*, pages 427–436. IEEE, 2009.

[13] Olena Schuessler and Diego Loyola. Parallel training of artificial neural networks using multithreaded and multicore cpus. In *International Conference on Adaptive and Natural Computing Algorithms*, pages 70–79. Springer, 2011.

[14] Udo Seiffert. Artificial neural networks on massively parallel computer hardware. *Neurocomputing*, 57:135–150, 2004.

[15] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-lm: Training multi-billion parameter language models using model parallelism. *arXiv preprint arXiv:1909.08053*, 2019.

[16] Xavier Sierra-Canto, Francisco Madera-Ramirez, and Victor Uc-Cetina. Parallel training of a back-propagation neural network using cuda. In *2010 Ninth International Conference on Machine Learning and Applications*, pages 307–312. IEEE, 2010.

[17] Volodymyr Turchenko, Lucio Grandinetti, George Bosilca, and Jack J Dongarra. Improvement of parallelization efficiency of batch pattern bp training algorithm using open mpi. *Procedia Computer Science*, 1(1):525–533, 2010.

[18] Junliang Wang, Jungang Yang, Jie Zhang, Xiaoxi Wang, and Wenjun Zhang. Big data driven cycle time parallel prediction for production planning in wafer manufacturing. *Enterprise information systems*, 12(6):714–732, 2018.

[19] K Wojtek Przytula, Viktor K Prasanna, and Wei-Ming Lin. Parallel implementation of neural networks. *Journal of VLSI signal processing systems for signal, image and video technology*, 4(2):111–123, 1992.