南開大學
Nankai University

计算机学院

并行程序设计 **MPI** 作业

ANN-bp 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 6 月 17 日

摘要

本文首先介绍期末大作业的选题与本次作业的选题。接下来，介绍算法的设计与实现。最后，介绍不同平台、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-bp; SIMD; openMP

目录

1 问题描述

1.1 期末选题

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。BP，Back Propagation，反向传播，是 ANN 网络的计算算法。

ANN 可以有很多应用场景，在期末大作业中，拟将其运用于简单的特征分类实际问题中，如根据花期等特征对植物的分类，环境污染照片的分类，或基于已知词嵌入的文本情感分类等。

- 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

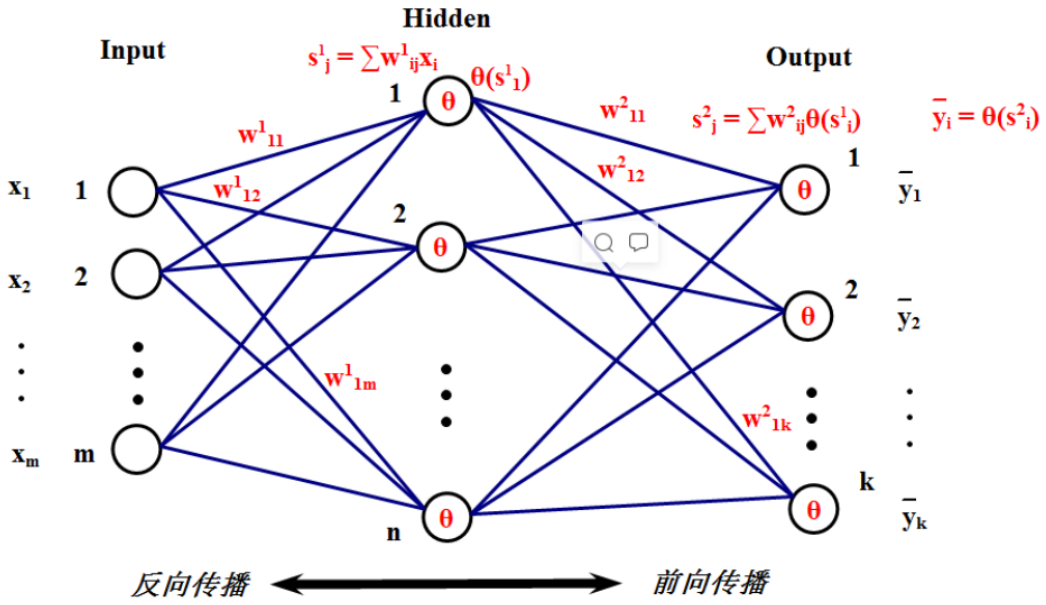

图 1.1: ANN 模型结构

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。

1.2 本次题目选题

本次实验将针对 ANN 计算过程中前向计算与反向传播的参数求导及其更新过程进行优化，这部分的算法描述如下。

1.2.1 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:

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

1.2.2 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:

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

1.2.3 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

```
function CALULATE DELTA
    for each layer reversely do
        error ← layers_{i+1}.weights * layers_{i+1}.delta
        layers_i.delta ← error * erivative_activation_function(layers_i.output_nodes)
    end for
end function

function BACK PROPAGATION
    for each layer reversely do
        layers_i.weights− = studyRate * layers_i.delta * layers_{i+1}.outputNodes
        layers_i.bias− = studyRate * layers_i.delta
    end for
end function
```

1.2.4 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

```
function TRAIN
    for each epoch do
        while samples are not traversed do
            for each sample in batch do
                predict()
            end for
            calulate loss()
            calulate delta()
```

```
            back propagation()
        end while
    end for
end function
```

2 算法的设计与实现

2.1 实验分析与设计

2.1.1 程序正确性的判断

为判断程序的正确性，采用固定的输入矩阵，并观察 lossfunction 是否收敛与收敛速度。此外，由于神经网络的运算对浮点数精度的要求不高，全部采用 float 型进行存储，对于不同运算次序精度对结果的不同影响，在此问题中也无须考虑。

2.1.2 实验问题规模的设置

根据并行部分的特性，这里隐藏层层数均为 1，ANN 迭代次数 numEpoch 均为 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）均为 64。这里，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，也使计时的差异更加明显，但因为其与并行部分无关，故在本次实验中保持不变。使用随机梯度下降，即，使用每个样本数据更新参数。改变不同输入层结点数、输出层结点数、隐藏层结点数，测试程序运行时间。

2.1.3 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，则程序的时间复杂度为：

$$T = O(numEpoch * numSamples * n^2) \tag{9}$$

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。

2.1.4 程序并行化的设计思路

为保证相同的问题规模，不采用 loss 函数是否收敛为终止条件，而设置特定的迭代次数做为终止条件。由于迭代的不同 batch 更新参数有前后依赖关系，并且在一个 batch 的训练中不同层前向传播与反向传播的层间也有依赖关系，故这里数据划分的方式采用在每层的矩阵运算中对矩阵的行或列划分。问题中共有三大类，共 5 个循环，可以优化。

1. 前向传播的循环, 可以在同一层内按照 weights 矩阵的不同列或行进行划分。若进行列划分，可直接避免数据访问的冲突，每个线程计算不同的行；若进行行划分，需要在每层计算后需要同步。

2. 计算导数 (更新量) 的循环，由于不存在访问冲突的问题，可以选择在同一层内按照 weights 矩阵的不同行或列进行划分，在每层计算后需要同步。

3. 更新 weights 的循环，类似的，在同一层内按照 weights 矩阵的不同列进行划分，在每层计算后需要同步。

2.2 MPI

pthread 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，这些都是本次实验需要考虑的重点。

2.2.1 朴素静态块划分

朴素静态块划分算法，通过 0 号进程进行数据的调度，0 号进程不参与计算，代码如下:

朴素静态块划分

```
void ANN_MPI::predict_MPI_static1()
{
    //静态块划分的朴素方法
    int myid, numprocs;
    MPI_Status status;
    MPI_Comm_rank(MPI_COMM_WORLD, &myid);
    MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
    for (int i_layer = 1; i_layer <= num_layers; i_layer++)
    {
        int i_size = (num_each_layer[i_layer + 1] + numprocs - 2) / (numprocs -
            1);//0号进程不参与计算

        if (myid == 0)
        {
            //数据发送
            for (int procs_i = 1; procs_i < numprocs; procs_i++)
            {
                MPI_Send(layers[i_layer - 1]->output_nodes,
                    num_each_layer[i_layer], MPI_FLOAT, procs_i, 98,
                    MPI_COMM_WORLD);
                for (int i = (procs_i - 1) * i_size; i <
                    min(num_each_layer[i_layer + 1], procs_i * i_size); i++)
                {
                    MPI_Send(layers[i_layer]->weights[i], num_each_layer[i_layer],
                        MPI_FLOAT, procs_i, 99 + i, MPI_COMM_WORLD);
                }
            }
```

```
23              //结果接收
24              for (int procs_i = 1; procs_i < numprocs; procs_i++)
25              {
26                  MPI_Recv(layers[i_layer]->output_nodes + (procs_i - 1) * i_size,
                        i_size, MPI_FLOAT, procs_i, 97, MPI_COMM_WORLD, &status);
27              }
28          }
29          else
30          {
31              //数据接收
32              MPI_Recv(layers[i_layer - 1]->output_nodes, num_each_layer[i_layer],
                    MPI_FLOAT, 0, 98, MPI_COMM_WORLD, &status);
33
34              //从节点计算
35              for (int i = (myid - 1) * i_size; i < min(num_each_layer[i_layer + 1],
                    myid * i_size); i++)
36              {
37                  MPI_Recv(layers[i_layer]->weights[i], num_each_layer[i_layer],
                        MPI_FLOAT, 0, 99 + i, MPI_COMM_WORLD, &status);
38                  layers[i_layer]->output_nodes[i] = 0.0;
39                  for (int j = 0; j < num_each_layer[i_layer]; j++)
40                  {
41                      layers[i_layer]->output_nodes[i] +=
                            layers[i_layer]->weights[i][j] * layers[i_layer -
                            1]->output_nodes[j];
42                  }
43                  layers[i_layer]->output_nodes[i] += layers[i_layer]->bias[i];
44                  layers[i_layer]->output_nodes[i] =
                        layers[i_layer]->activation_function(layers[i_layer]->output_nodes[i]);
45              }
46
47              //从节点将结果发送回主节点（0号）
48              MPI_Send(layers[i_layer]->output_nodes + (myid - 1) * i_size, i_size,
                    MPI_FLOAT, 0, 97, MPI_COMM_WORLD);
49          }
50      }
51  }
```

2.2.2 对等模式任务块划分

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 中，第 n 层的计算需要用到 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。那么对任务的分配也就可以聚焦到特定层不同节点的计算上。需要注意的是，每个进程只负责一部分隐层节点的计算，但之后的计算需要用到该隐层所有节点的计算结果，所以需要一个特定的进程来进行结果的接收和同步。

具体地,首先在 main 函数中调用 MPI_Init () 初始化进程。对每个进程,可以分别调用 MPI_Comm_rank() 和 MPI_Com_size () 函数获取自身的进程 id 以及全部进程数量，由这两个参数即可确定进程需要处

理的数据大小及范围。对于参与运算的权重和隐层输出数组，程序指定 0 号进程调用 MPI_Bcast() 函数广播给其他进程。同样地，0 号进程还负责从其他进程接收隐层不同节点的计算结果用于后续的计算。对等模式 ANN 并行的核心代码如下：

对等模式任务划分

```
void ANN_MPI::predict_MPI_static1()
{
    //对等模式任务划分
 ...
    for (int i_layer = 1; i_layer <= num_layers; i_layer++)
    {
        int i_size = (num_each_layer[i_layer + 1] + numprocs - 1) /
            (numprocs);//0号进程参与计算
        //数据发送
        MPI_Bcast(layers[i_layer - 1]->output_nodes, num_each_layer[i_layer],
            MPI_FLOAT, 0, MPI_COMM_WORLD);
        for (int i = 0; i < num_each_layer[i_layer + 1]; i++)
            MPI_Bcast(layers[i_layer]->weights[i], num_each_layer[i_layer],
                MPI_FLOAT, 0, MPI_COMM_WORLD);
        //计算
        ...
        //结果发送和接收
        ...
    }
    ...
}
```

2.2.3 对等模式任意结果接收

在上一节对等模式任务块划分中，0 号进程采用了常用的按照进程编号顺序的方式来接收结果，这样虽然能够可以比较好地定位接受数据的缓冲区，但是可能会因为各个进程的负载不均而导致编号靠后的进程产生较长的空闲等待。因此在 0 号进程接收数据时可以将源进程号改为 MPI_ANY_SOURCE，也就是任意进程，以“先计算完的先接收”的方式从各个进程接收运算结果，再根据解析出来的实际进程号来确定结果应该存储的位置。优化后的随机结果接收代码如下：

对等模式随机结果接收

```
void ANN_MPI::predict_MPI_static2()
{
    //静态块划分，0号进程接收数据时来自MPI_ANY_SOURCE，无需顺序进行
...
    for (int i_layer = 1; i_layer <= num_layers; i_layer++)
    {
        ...
        //数据发送
        ...
        //计算
        ...
```

```
        //结果发送和接收
        if (myid == 0)
        {
            float* temp_nodes = new float[num_each_layer[i_layer + 1]];
            for (int temp_procs_i = 1; temp_procs_i < numprocs; temp_procs_i++)
            {
                //0号进程接收数据时来自MPI_ANY_SOURCE, 无需顺序进行
                MPI_Recv(temp_nodes, i_size, MPI_FLOAT, MPI_ANY_SOURCE, 97,
                    MPI_COMM_WORLD, &status);
                memcpy(layers[i_layer]->output_nodes + status.MPI_SOURCE * i_size,
                    temp_nodes, sizeof(float) * i_size);
            }
            delete[] temp_nodes;
            // cout<<myid<<"finish recv"<<endl;
        }
        else
        {
            //从节点将数据发送回主节点（0号）
            MPI_Send(layers[i_layer]->output_nodes + myid * i_size, i_size,
                MPI_FLOAT, 0, 97, MPI_COMM_WORLD);
            // cout<<myid<<"finish send"<<endl;
        }
    }
}
```

2.2.4 主从模式动态任务分配

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隐层节点，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行权重矩阵（对应一个隐层节点的计算），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隐层节点并进行存储。接收完毕后，如果仍有未计算的隐层节点，则给该空闲进程发送一行相应的权重矩阵。重复以上步骤直到所有的隐层节点都被计算完毕。最后 0 号进程会将整合好的计算结果广播给其他节点，进入下一层的计算。

关于不同的任务分配粒度，这里均采用一行为粒度。原因是若为多行，无法保证每行的发送、计算、接收为原子操作，无法判断从进程是否完成，当接收到一行新的结果，只能再发送一行，粒度就退化为 1 行。

主从模式动态任务分配的核心代码如下：

主从模式动态任务分配

```
void ANN_MPI::predict_MPI_dynamic()
{
    //动态分配任务,主从式,0号进程不参与计算
```

```
    int myid, numprocs;
    MPI_Status status;
    MPI_Comm_rank(MPI_COMM_WORLD, &myid);
    MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
    for (int i_layer = 1; i_layer <= num_layers; i_layer++)
    {
        //数据发送
        //粒度必须为1行, 因为多行的接收不能保证为原子操作
        //广播layers[i_layer - 1]->output_nodes,
        MPI_Bcast(layers[i_layer - 1]->output_nodes, num_each_layer[i_layer],
            MPI_FLOAT, 0, MPI_COMM_WORLD);
        if (myid == 0)
        {
            int i;
            for (i = 0; i < numprocs - 1; i++)
            {
                MPI_Send(layers[i_layer]->weights[i], num_each_layer[i_layer],
                    MPI_FLOAT, i + 1, i, MPI_COMM_WORLD); //发送一行, tag为行号
            }
            float temp_node;
            int finish = 0;
            while (finish < numprocs - 1)
            {
                //第i-1行已完成
                MPI_Recv(&temp_node, 1, MPI_FLOAT, MPI_ANY_SOURCE, MPI_ANY_TAG,
                    MPI_COMM_WORLD, &status);

                if (status.MPI_TAG < num_each_layer[i_layer + 1])
                    layers[i_layer]->output_nodes[status.MPI_TAG] = temp_node;

                if (i < num_each_layer[i_layer + 1])  //根据tag判断是否完成全部的行
                {
                    //尚未完成, 发送新的一行, i
                    MPI_Send(layers[i_layer]->weights[i], num_each_layer[i_layer],
                        MPI_FLOAT, status.MPI_SOURCE, i, MPI_COMM_WORLD);
                    i++;
                }
                else
                {
                    //已经完成, 发送任意数据, 使从进程退出循环
                    MPI_Send(layers[i_layer]->weights[0], num_each_layer[i_layer],
                        MPI_FLOAT, status.MPI_SOURCE, num_each_layer[i_layer + 1] +
                        1, MPI_COMM_WORLD);
                    finish++;
                }
            }
        }
        else
        {
```

```
47              float* temp_nodes = new float[num_each_layer[i_layer]];
48              int i = myid;
49              while (i < num_each_layer[i_layer + 1])
50              {
51                  //接收一行, tag为行号
52                  MPI_Recv(temp_nodes, num_each_layer[i_layer], MPI_FLOAT, 0,
                        MPI_ANY_TAG, MPI_COMM_WORLD, &status);
53                  i = status.MPI_TAG;
54                  if (i >= num_each_layer[i_layer + 1]) break;
55
56                  //计算
57  ...
58
59                  //从节点将数据发送回主节点（0号）
60                  MPI_Send(&output_nodes, 1, MPI_FLOAT, 0, i, MPI_COMM_WORLD);
61              }
62              delete[] temp_nodes;
63          }
64      }
65  }
```

2.2.5 非阻塞通信

在以上的代码实现中，0 号进程和其他进程进行数据传输采用的都是阻塞通信，这就意味着在进程在调用 MPI_Send() 或 MPI_Recieve() 之后，必须要等到数据成功发送或接收，才能接着往后执行。但实际上在 MPI 程序中，数据的传输往往非常耗时，尤其是对于数据接收来说，必须等源进程将数据准备好之后才能接收，会导致长时间的 CPU 空闲，降低程序效率。针对这个问题，MPI 提供了非阻塞通信方式 MPI_Isend() 和 MPI_Irecieve() 。对于发送方，可以在调用函数后重用发送缓冲区，对于接收方，可以在调用函数后进行后续的代码执行，等待数据成功接收到缓冲区后再处理数据。

ANN 的计算主要集中在前向传播、反向传播和权重更新这三步，同样数据传输也主要是在这几步中。考虑到各个进程在发送完数据后对缓冲区的重用没有要求，且非阻塞发送对性能影响不大，因此主要关注非阻塞接收。在前向传播中，中间每层结果的计算都需要用到上一层的输出结果和对应权重矩阵，因此在成功接收到这些值之前，无法进行后续的计算，显然无法使用非阻塞通信。但需要注意的是，在 0 号进程接收到最后的输出层结果后，紧接着是计算损失函数值，计算过程中并不需要一次性用到所有的输出层结果，这时可以采用非阻塞接收以提高程序效率。

具体地，构建一个 MPI_Request 指针数组 req，0 号进程使用 req 循环从其他进程中非阻塞接收最终输出层结果，之后针对 req 循环调用 MPI_Waitany()，对成功接收到的节点输出结果计算损失函数值，直到所有的节点结果都被接收和计算完毕。

2.2.6 组通信规约

ANN 中 0 号进程在收集中间层计算结果时是从所有其他进程结果数组中的不同位置中获取相同大小的数据到自己的结果数组中，这与组通信中的规约操作非常类似，只要将结果中与进程计算无关的数据初始化为 0，然后采用 MPI_Reducec() 操作对 0 号进程的结果数组进行一个加法的规约操作，这样就可以用非常简单的代码实现结果的收集。但实际上，这样每次都需要传输整个结果数组，会导

致较大的数据传输开销，并且会有额外的加法操作影响, 因此对效率不会有提升。

2.2.7 Scatter 和 Gather

0 号进程在收集计算结果时其实并没有进行计算操作，因此使用加法规约操作会有些多余的运算。对于这种从从其他进程接收不同的数据，可以使用 MPI_Gather() 函数，同样的，在 0 号进程向其他进程发送不同数据时，可以使用 MPI_Scatter() 函数，这样也就不需要再循环发送接收了。其核心代码如下:

Scatter 和 Gather

```
void ANN_MPI::predict_MPI_gather()
{
    //多进程规约
...
    for (int i_layer = 1; i_layer <= num_layers; i_layer++)
    {
        //数据发送
        int position = 0;
        int buff_size = num_each_layer[i_layer + 1] * num_each_layer[i_layer] *
            sizeof(float);
        float* buffer_for_packed = new float[buff_size];
        float* recv_buffer = new float[buff_size];

        //由进程0打包wieghts
        if (myid == 0)
        {
            for (int i = 0; i < num_each_layer[i_layer + 1]; i++)
            {
                MPI_Pack(layers[i_layer]->weights[i], num_each_layer[i_layer],
                    MPI_FLOAT, buffer_for_packed, buff_size, &position,
                    MPI_COMM_WORLD);
            }
        }

...
        //Scatter打包好的weights
        MPI_Scatter(buffer_for_packed, num_each_layer[i_layer] * i_size,
            MPI_PACKED, recv_buffer, num_each_layer[i_layer] * i_size, MPI_PACKED,
            0, MPI_COMM_WORLD);

        //广播layers[i_layer - 1]->output_nodes
        MPI_Bcast(layers[i_layer - 1]->output_nodes, num_each_layer[i_layer],
            MPI_FLOAT, 0, MPI_COMM_WORLD);

        //从进程需要将weights解包并进行后续计算
        if (myid != 0)
        {
            position = 0;
```

```
            for (int i = myid * i_size; i < min(num_each_layer[i_layer + 1], (myid
                + 1) * i_size); i++)
            {
                MPI_Unpack(recv_buffer, buff_size, &position,
                    layers[i_layer]->weights[i], num_each_layer[i_layer],
                    MPI_FLOAT, MPI_COMM_WORLD);
            }
        }
        delete[] buffer_for_packed;
        delete[] recv_buffer;

        //计算
...

        //数据收集, gather到0号进程
        float* buff_output = new float[num_each_layer[i_layer + 1]];
        MPI_Gather(layers[i_layer]->output_nodes + myid * i_size, my_i_size,
            MPI_FLOAT, buff_output, my_i_size, MPI_FLOAT, 0, MPI_COMM_WORLD);
        if (myid == 0)
            memcpy(layers[i_layer]->output_nodes, buff_output,
                num_each_layer[i_layer + 1]);
        delete[] buff_output;
    }
}
```

2.2.8 All-to-All 广播

在以上代码中，其他进程分别将自己计算的结果发送给 0 号进程，如果后续需要计算下一层，0 号进程再讲整合好的结果广播给其他进程，使所有进程都有一份计算结果数据。这其实就相当于所有进程对计算结果做了一个 All-to-All 的广播。具体地，可以使用 MPI_Allgather() 函数，相当于组内每个进程都执行一次收集。其具体代码如下：

All-to-All 广播

```
void ANN_MPI::predict_MPI_alltoall()
{
    //all-to-all
...

    //第一次需要进行广播
    MPI_Bcast(layers[0]->output_nodes, num_each_layer[0], MPI_FLOAT, 0,
        MPI_COMM_WORLD);
...
        //数据打包、发送、解包
...

        //计算
        for (int i = myid * i_size; i < min(num_each_layer[i_layer + 1], (myid +
            1) * i_size); i++)
```

```
14          {
15  ...
16              //数据收集并发送给所有进程
17              MPI_Allgather(&layers[i_layer]->output_nodes[i], 1, MPI_FLOAT,
                    &layers[i_layer]->output_nodes[i], 1, MPI_FLOAT, MPI_COMM_WORLD);
18          }
19
20      }
21  }
```

2.2.9 单边通信

在所有每一隐层计算完毕后，程序希望的是将所有进程的计算结果写入 0 号进程的结果数组中，于是很自然地想将 0 号进程的结果作为 RMA 共享内存曝露给其他进程，这样其他进程就可以通过 MPI 提供的 RMA 窗口访问接口直接将计算的结果写入 0 号进程的结果数组。这种单边通信不需要 0 号进程的直接参与。在每一层的实际计算中，结果都是存储在 layer.output_nodes 数组中，于是创建 2 个窗口，即分别当前层和上一层的 output_nodes 数组共享，不同进程按块划分，直接在相应位置对其进行读写操作。为了保证 0 号进程读取 result 时所有进程均已写结果完毕，在写结果之后和读结果之前加入 MPI_Barrier() 函数进行同步。单边通信的核心代码如下：

单边通信

```
1   void ANN_MPI::predict_MPI_rma()
2   {
3       //rma单边通信，将每层的output_nodes共享
4   ...
5
6       //声明窗口句柄
7       MPI_Win win_previous_layer;
8       MPI_Win win_current_layer;
9       for (int i_layer = 1; i_layer <= num_layers; i_layer++)
10      {
11  ...
12          //将上一层和当前层的output_nodes共享
13          MPI_Win_create(layers[i_layer - 1]->output_nodes, num_each_layer[i_layer]
                * sizeof(float), sizeof(float), MPI_INFO_NULL, MPI_COMM_WORLD,
                &win_previous_layer);
14          MPI_Win_create(layers[i_layer]->output_nodes, num_each_layer[i_layer + 1]
                * sizeof(float), sizeof(float), MPI_INFO_NULL, MPI_COMM_WORLD,
                &win_current_layer);
15
16          //发送weights数据
17  ...
18
19          //计算
20  ...
21          //不再需要结果发送和接收，但为保证上一层的节点已经全部计算完成，需要进行同步。
22          MPI_Barrier(MPI_COMM_WORLD);
```

```
    }
    MPI_Win_free(&win_previous_layer);
    MPI_Win_free(&win_current_layer);
}
```

2.3 MPI + OpenMP

MPI 定义了四种安全级别:

- 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, 如果采用块划分的方式, 每个进程将分配到到若干个需要计算的隐层节点, 这个时候每个进程可以使用 OpenMP 创建多个线程, 将这些隐层节点分配给不同的线程去进行实际的计算, 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。MPI + OpenMP 的核心代码如下所示:

MPI + OpenMP

```
void ANN_MPI::predict_MPI_threads()
{
    //与predict_MPI_static2相同, 增加了多线程, MPI+openMP
...
    for (int i_layer = 1; i_layer <= num_layers; i_layer++)
    {
...
        //数据发送
...
        //计算
#pragma omp parallel num_threads(NUM_THREADS)
        {
            //#pragma omp parallel for
            for (int i = myid * i_size; i < min(num_each_layer[i_layer + 1], (myid
                + 1) * i_size); i++)
            {
...
#pragma omp parallel for reduction(+:sum)
                for (int j = 0; j < num_each_layer[i_layer]; j++)
                {
...
                }
...
```

```
            }
        }
...
    }
}
```

2.4 MPI + OpenMP + SIMD

在 MPI + OpenMP 的基础上，还可以加上与线程无关的 SIMD 提高程序并行效率。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 SSE 的 API 进行具体的计算。MPI + OpenMP + SIMD 的核心代码如下:

MPI + OpenMP + SIMD

```
void ANN_MPI::predict_MPI_threads_SIMD()
{
    //与predict_MPI_static2相同，增加了多线程和向量化，MPI+openMP+SSE(SIMD)
    int myid, numprocs;
    MPI_Status status;
    MPI_Comm_rank(MPI_COMM_WORLD, &myid);
    MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
    for (int i_layer = 1; i_layer <= num_layers; i_layer++)
    {
        int i_size = (num_each_layer[i_layer + 1] + numprocs - 1) /
            (numprocs);//0号进程参与计算
        //数据发送
        MPI_Bcast(layers[i_layer - 1]->output_nodes, num_each_layer[i_layer],
            MPI_FLOAT, 0, MPI_COMM_WORLD);
        for (int i = 0; i < num_each_layer[i_layer + 1]; i++)
            MPI_Bcast(layers[i_layer]->weights[i], num_each_layer[i_layer],
                MPI_FLOAT, 0, MPI_COMM_WORLD);
        //计算
#pragma omp parallel num_threads(NUM_THREADS)
        {
#pragma omp for//使用多线程拆分i的循环，将不同节点分配给不同线程
            for (int i = myid * i_size; i < min(num_each_layer[i_layer + 1], (myid
                + 1) * i_size); i++)
            {
                layers[i_layer]->output_nodes[i] = 0.0;

                //使用SIMD拆分内部的j的循环，归约
                int max_j = num_each_layer[i_layer];
                __m128 ans = _mm_setzero_ps();
                for (int j = 0; j < max_j; j += 4)
                {
                    __m128 t1, t2;
                    //将内部循环改为4位的向量运算
```

```
                    t1 = _mm_loadu_ps(layers[i_layer]->weights[i] + j);
                    t2 = _mm_loadu_ps(layers[i_layer - 1]->output_nodes + j);
                    t1 = _mm_mul_ps(t1, t2);

                    ans = _mm_add_ps(ans, t1);
                }
                // 4个局部和相加
                ans = _mm_hadd_ps(ans, ans);
                ans = _mm_hadd_ps(ans, ans);
                //标量存储
                float z;
                _mm_store_ss(&z, ans);
                layers[i_layer]->output_nodes[i] += z;
                layers[i_layer]->output_nodes[i] += layers[i_layer]->bias[i];
                layers[i_layer]->output_nodes[i] =
                    layers[i_layer]->activation_function(layers[i_layer]->output_nodes[i]);
            }
        }
        //结果接收
        if (myid == 0)
        {
            float* temp_nodes = new float[num_each_layer[i_layer + 1]];
            for (int temp_procs_i = 1; temp_procs_i < numprocs; temp_procs_i++)
            {
                MPI_Recv(temp_nodes, i_size, MPI_FLOAT, MPI_ANY_SOURCE, 97,
                    MPI_COMM_WORLD, &status);
                memcpy(layers[i_layer]->output_nodes + status.MPI_SOURCE * i_size,
                    temp_nodes, sizeof(float) * i_size);
            }
            delete[] temp_nodes;
        }
        else
        {
            //从节点将数据发送回主节点（0号）
            MPI_Send(layers[i_layer]->output_nodes + myid * i_size, i_size,
                MPI_FLOAT, 0, 97, MPI_COMM_WORLD);
        }
    }
}
```

3 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，分析了不同进程数目对 MPI 程序运行时间的影响。考虑到 MPI 程序主要受到使用不同任务划分方法的编程模式（对等模式、主从模式）和数据传输方法（任意接收、非阻塞通信、Scatter 和 Gather、All-to-All、单边通信）影响，实验对这两方面的运行时间分别进行了比较分析。此外，MPI 还能与多线程方法 OpenMP 以及指令集 SIMD 相结合，实验探讨了串行、MPI、MPI+OpenMP 以及 MPI+OpenMP+SIMD 的运行时间差异。最后实验还分

析了 ARM 平台和 X86 平台上运行 MPI 程序的差异。需要注意的是，除了基础的对等模式 MPI 以及不同平台运行比较部分，其他部分的实验均是在本机上运行。

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。

3.1 串行程序与复杂度分析

对于 ANN 问题的串行解决方案，在不同的数据规模下进行了实验。考虑到算法复杂度主要受隐层节点数目的影响，可以固定隐层数为 1 并假设输入输出层节点数目与隐层相同且均为 n，调节 n 的大小进行实验，实验结果如表1所示，其中串行 predict 测试的是 MPI 调用计算部分的运行时间，串行 train 测试的是整个训练的运行时间。此外，由于外层循环对程序运行时间不造成影响，只体现为常数倍的增加，故保持 epoch 数与 batchsize 不变。可以看到运行时间与数据规模大致成二次方的关系。以前向传播为例，输入层到隐层和隐层到输出层的时间复杂度均为 $O(n^2)$，与实验结果相符。

表 1: 不同问题规模下的串行程序运行时间 (单位：s)

问题规模 n	串行 predict	串行 train
32	1.27074	5.25483
64	2.97058	12.1254
96	6.75149	27.5.46
128	11.88887	48.6007
256	47.1046	191.145

3.2 基础 MPI 与进程数量影响

理论上来说，在问题确定的情况下，进程数目对运行时间的影响不太会受到不同编程方法的影响，而主要受实际计算资源（运算核心数）的限制。为了不失一般性，将对等式任务块划分方式作为基础的 MPI 程序，在金山云服务器上运行，使用 4 个节点，最多可以分配 8 个进程，其在不同问题规模以及不同进程数下的实验结果如表2所示。由于服务器所限，仅测试了如下几个进程的运行结果。可以看到程序的加速比随进程数增加而增加，符合预期。

根据运行时间，之后的实验均设置为 4 进程。

表 2: 基础 MPI 在不同进程数量下的运行时间 (单位：s)

问题规模 n	2 进程	4 进程	6 进程	8 进程
32	0.642537	0.394709	0.26170	0.19772
64	2.69362	1.11957	0.8081	0.56072
96	6.28966	2.54562	1.86164	1.27987
128	11.08551	4.253	3.070037	2.14112
256	43.342933	37.5902	23.90558	18.80914

3.3 编程模式影响

MPI 标准的编程范式主要有对等模式以及主从模式，在对等模式下，每个进程根据自身的进程 ID 获取相应的数据块，所有的进程均参与运算；在主从模式下，0 号进程负责向其他进程循环发送计算数据，并负责计算结果的接收和整理。相比较来说，对等模式编程更简单，任务划分力度更粗，而主从模式任务划分力度更细，负载更均衡，但可能会有更频繁的数据传输开销。二者的实验结果如表3所示，

可以看到主从模式的加速比远高于对等模式。实际上，加速比已经超过了进程数，这是不合理的，可能是因为在实际实现过程中，由于整体的代码结构全部进行了重构，无论从数据访问还是计算顺序与串行代码都有比较大的差异，所以出现这种情况。对等模式从任意源接收也明显优于朴素的对等模式。这说明相对于数据传输开销，负载均衡十分重要，数据同步和等待开销要远高于数据传输开销。

表 3: 不同编程模式的运行时间 (单位：s)

问题规模 n	串行	对等模式	对等模式从任意源接收	主从模式
32	0.853191	0.394709	0.473361	0.26907
64	2.81737	1.11957	1.11714	0.58496
96	6.44303	2.54562	2.36315	1.19879
128	11.2415	4.253	3.83776	1.90161
256	47.0429	37.5902	15.1867	6.91494

3.4 数据传输方式影响

MPI 程序提供了大量的消息传递接口，为多进程并行提供了数据传输的保障。在实际场景下，不同的进程往往运行在不同的物理节点上，它们之间存储空间存储介质不共享，数据需要通过网络传输，带来较大的时间开销。因此，选择合适的数据方式，增加代码的简洁性或提升数据传输的效率，就显得尤为重要。为此实验对比了任意接收、Scatter 和 Gather、All-to-All、单边通信对程序运行时间的影响，具体的实验结果如表4所示。可以看到几个不同的实现方式的运行时间相近，Scatter 和 Gather 略快于其他。因为几种函数总体数据传输过程相近，而大量的矩阵向量相乘，十分适用 Scatter 和 Gather 的场景，这也是符合预期的。

表 4: 不同数据传输方式的运行时间 (单位：s)

问题规模 n	send&recv	Scatter&Gather	All-to-All	单边通信
32	0.394709	0.38059	0.381084	0.48205
64	1.11957	1.07859	1.08421	1.38292
96	2.54562	2.44435	2.57133	3.16252
128	4.253	4.07616	3.99291	5.2964
256	37.5902	15.76418	16.3242	17.23806

3.5 MPI+OpenMP+SIMD 混合编程

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。采用每个进程 4 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5所示，可以看到 4 线程在多进程的基础上可以带来大于 3 倍的加速比，有十分好的优化效果，而 SIMD 的 4 路向量化再带来约 2 倍的加速比，这也是符合预期的。

3.6 ARM 平台和 x86 平台运行比较

实验测试了 MPI 程序（基础对等模式）在鲲鹏服务器 ARM 平台和 x86 平台上的运行结果。均设置 4 个进程，每个进程 4 个线程。最后的实验结果如表6所示。可以看到 x86 平台速度远快于 ARM，但同样的 MPI 实现方法，同样的规模带来的加速比是相近的。

表 5: 串行与混合编程方式的运行时间 (单位：s)

问题规模 n	串行	MPI	MPI+OpenMP	MPI+OpenMP+SIMD
32	0.853191	0.394709	0.130089	0.098254
64	2.81737	1.11957	0.35847	0.205352
96	6.44303	2.54562	0.78945	0.397239
128	11.2415	4.253	1.319946	0.647851
256	47.0429	37.5902	12.45261	5.319354

表 6: 不同平台的运行时间 (单位：s)

问题规模 n	ARM 平台	x86 平台
32	9.07146	0.394709
64	29.29394	1.11957
96	62.84011	2.54562
128	112.39116	4.253
256	465.61321	7.5902

4 总结

本文主要研究了 MPI 对 ANN 问题算法时间的影响。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。不同于 pthread 和 OpenMP 提供的线程级别的并行，MPI 为我们提供了进程级别的并行，这在充分利用集群计算资源时尤为实用。本文将将 MPI 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进程数量的线性关系，这也证实了 MPI 在提升程序效率上的实用性。更深入地，本文还探讨了不同编程模式和不同数据传输方式对 MPI 程序运行时间的影响，结果表明即使是细微代码的调整和技巧的使用也能给程序带来影响。当然，除了代码，不同的硬件也会产生很大的影响，实验程序在 ARM 和 x86 平台上的运行时间也有着显著的差异。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。

并行的技术远远不是仅限于 MPI、OpenMP、Pthread 和 SIMD，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。在之后的实验中可以考虑对并行的细节进行进一步的优化或者使用其他的并行技术来更好地解决问题。

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

附录 A

由于完整代码将近 2000 行，这里仅给出完整的 train 函数的对等模式块划分代码供参考。

train 函数的对等模式块划分

```
void ANN_MPI::train_MPI_all_static(const int num_sample, float** _trainMat,
    float** _labelMat)
{
    //对此函数中全部关键循环均进行了MPI优化，采用静态分配方式
    printf("begin training\n");  // cout<<' ';
    float thre = 1e-2;
    float* avr_X = new float[num_each_layer[0]];
    float* avr_Y = new float[num_each_layer[num_layers + 1]];


    long long time_mpi = 0;
    long long head, tail, freq;// timers
    QueryPerformanceFrequency((LARGE_INTEGER*)&freq);

    QueryPerformanceCounter((LARGE_INTEGER*)&head); // start time

    int myid, numprocs;
    MPI_Status status;
    MPI_Comm_rank(MPI_COMM_WORLD, &myid);
    MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
    for (int epoch = 0; epoch < num_epoch; epoch++)
    {
        if (epoch % 50 == 0)
        {
            //  printf ("round%d:\n", epoch);
        }
        int index = 0;

        while (index < num_sample)
        {
            for (int X_i = 0; X_i < num_each_layer[0]; X_i++) avr_X[X_i] = 0.0;
            for (int Y_i = 0; Y_i < num_each_layer[num_layers + 1]; Y_i++)
                avr_Y[Y_i] = 0.0;

            for (int j = 0; j < num_each_layer[num_layers + 1]; j++)
            {
                layers[num_layers]->delta[j] = 0.0;
            }

            for (int batch_i = 0; batch_i < batch_size && index < num_sample;
                batch_i++, index++)
                //默认batch_size=1，即采用随机梯度下降法，每次使用全部样本训练并更新参数
            {
                //1.前向传播，对i进行分配
```

```
41
42                  int i_size = (num_each_layer[1] + numprocs - 2) / (numprocs - 1);
43                  if (myid == 0)
44                  {
45                      for (int procs_i = 1; procs_i < numprocs; procs_i++)
46                      {
47                          for (int i = (procs_i - 1) * i_size; i <
                                min(num_each_layer[0 + 1], procs_i * i_size); i++)
48                          {
49                              MPI_Send(layers[0]->weights[i], num_each_layer[0],
                                    MPI_FLOAT, procs_i, 99 + i, MPI_COMM_WORLD);
50                          }
51                      }
52                      for (int procs_i = 1; procs_i < numprocs; procs_i++)
53                      {
54                          MPI_Recv(layers[0]->output_nodes + (procs_i - 1) * i_size,
                                i_size, MPI_FLOAT, procs_i, 97, MPI_COMM_WORLD,
                                &status);
55                      }
56                  }
57                  else
58                  {
59                      for (int i = (myid - 1) * i_size; i < min(num_each_layer[0 +
                            1], myid * i_size); i++)
60                      {
61                          MPI_Recv(layers[0]->weights[i], num_each_layer[0],
                                MPI_FLOAT, 0, 99 + i, MPI_COMM_WORLD, &status);
62                          layers[0]->output_nodes[i] = 0.0;
63                          for (int j = 0; j < num_each_layer[0]; j++)
64                          {
65                              layers[0]->output_nodes[i] += layers[0]->weights[i][j]
                                    * _trainMat[index][j];
66                          }
67                          layers[0]->output_nodes[i] += layers[0]->bias[i];
68                          layers[0]->output_nodes[i] =
                                layers[0]->activation_function(layers[0]->output_nodes[i]);
69                      }
70                      MPI_Send(layers[0]->output_nodes + (myid - 1) * i_size,
                            i_size, MPI_FLOAT, 0, 97, MPI_COMM_WORLD);
71                  }
72
73                  for (int i_layer = 1; i_layer <= num_layers; i_layer++)
74                  {
75                      int i_size = (num_each_layer[i_layer + 1] + numprocs - 2) /
                            (numprocs - 1);
76                      if (myid == 0)
77                      {
78                          for (int procs_i = 1; procs_i < numprocs; procs_i++)
79                          {
```

```
80                          MPI_Send(layers[i_layer - 1]->output_nodes,
                                num_each_layer[i_layer], MPI_FLOAT, procs_i, 98,
                                MPI_COMM_WORLD);
81                          for (int i = (procs_i - 1) * i_size; i <
                                min(num_each_layer[i_layer + 1], procs_i * i_size);
                                i++)
82                          {
83                              MPI_Send(layers[i_layer]->weights[i],
                                    num_each_layer[i_layer], MPI_FLOAT, procs_i,
                                    1000 + i, MPI_COMM_WORLD);
84                          }
85                      }
86                      for (int procs_i = 1; procs_i < numprocs; procs_i++)
87                      {
88                          MPI_Recv(layers[i_layer]->output_nodes + (procs_i - 1)
                                * i_size, i_size, MPI_FLOAT, procs_i, 97,
                                MPI_COMM_WORLD, &status);
89                      }
90                  }
91                  else
92                  {
93                      MPI_Recv(layers[i_layer - 1]->output_nodes,
                            num_each_layer[i_layer], MPI_FLOAT, 0, 98,
                            MPI_COMM_WORLD, &status);
94                      for (int i = (myid - 1) * i_size; i <
                            min(num_each_layer[i_layer + 1], myid * i_size); i++)
95                      {
96                          MPI_Recv(layers[i_layer]->weights[i],
                                num_each_layer[i_layer], MPI_FLOAT, 0, 1000 + i,
                                MPI_COMM_WORLD, &status);
97                          layers[i_layer]->output_nodes[i] = 0.0;
98                          for (int j = 0; j < num_each_layer[i_layer]; j++)
99                          {
100                             layers[i_layer]->output_nodes[i] +=
                                    layers[i_layer]->weights[i][j] * layers[i_layer
                                    - 1]->output_nodes[j];
101                         }
102                         layers[i_layer]->output_nodes[i] +=
                                layers[i_layer]->bias[i];
103                         layers[i_layer]->output_nodes[i] =
                                layers[i_layer]->activation_function(layers[i_layer]->output_nod
104                     }
105                     MPI_Send(layers[i_layer]->output_nodes + (myid - 1) *
                            i_size, i_size, MPI_FLOAT, 0, 97, MPI_COMM_WORLD);
106                 }
107
108             }
109
110             if (myid != 0)
```

```
111                 continue;
112
113             // cout<<"finish pridect"<<endl;
114
115             //计算loss, 即最后一层的delta,即该minibatch中所有loss的平均值
116             for (int j = 0; j < num_each_layer[num_layers + 1]; j++)
117             {
118                 //均方误差损失函数,batch内取平均
119                 layers[num_layers]->delta[j] +=
                        (layers[num_layers]->output_nodes[j] - _labelMat[index][j])
                        *
                        layers[num_layers]->derivative_activation_function(layers[num_layers]->o
120                 //交叉熵损失函数
121                 //layers[num_layers]->delta[j] +=
                        (layers[num_layers]->output_nodes[j] - _labelMat[index][j]);
122             }
123             // printf ("finish cal error\n");
124             for (int X_i = 0; X_i < num_each_layer[0]; X_i++) avr_X[X_i] +=
                    _trainMat[index][X_i];
125             for (int Y_i = 0; Y_i < num_each_layer[num_layers + 1]; Y_i++)
                    avr_Y[Y_i] += _labelMat[index][Y_i];
126         }
127
128         //delta在batch内取平均, avr_X、avr_Y分别为本个batch的输入、输出向量的平均值
129         if (index % batch_size == 0)
130         {
131             for (int j = 0; j < num_each_layer[num_layers + 1]; j++)
132             {
133                 if (batch_size == 0) printf("wrong!\n");
134                 layers[num_layers]->delta[j] /= batch_size;
135                 //for(int
                        i=0;i<5;i++)printf("delta=%f\n",layers[num_layers]->delta[i]);
136                 avr_Y[j] /= batch_size;
137             }
138             for (int X_i = 0; X_i < num_each_layer[0]; X_i++) avr_X[X_i] /=
                    batch_size;
139
140         }
141         else
142         {
143             for (int j = 0; j < num_each_layer[num_layers + 1]; j++)
144             {
145                 if (index % batch_size == 0) printf("wrong!\n");
146                 layers[num_layers]->delta[j] /= (index % batch_size);
147                 avr_Y[j] /= (index % batch_size);
148             }
149             for (int X_i = 0; X_i < num_each_layer[0]; X_i++) avr_X[X_i] /=
                    (index % batch_size);
150         }
```

```
151             // printf("index:%d\n",index);
152             //计算loss func, 仅用于输出
153             if (index >= num_sample)
154             {
155                 static int tt = 0;
156                 float loss = 0.0;
157                 for (int t = 0; t < num_each_layer[num_layers + 1]; ++t)
158                 {
159                     loss += (layers[num_layers]->output_nodes[t] - avr_Y[t]) *
                            (layers[num_layers]->output_nodes[t] - avr_Y[t]);
160                 }
161                 // printf ("第%d次训练: %0.12f\n", tt++, loss);
162             }
163 
164             //反向传播更新参数
165 
166             //2.计算每层的delta,行访问优化, 对j进行数据分配
167             for (int i = num_layers - 1; i >= 0; i--)
168             {
169                 //从0号进程广播
170                 for (int k = 0; k < num_each_layer[i + 2]; k++)
171                 {
172                     MPI_Bcast(layers[i + 1]->weights[k], num_each_layer[i + 1],
                            MPI_FLOAT, 0, MPI_COMM_WORLD);
173                 }
174                 MPI_Bcast(layers[i + 1]->delta, num_each_layer[i + 2], MPI_FLOAT,
                        0, MPI_COMM_WORLD);
175 
176                 float* error = new float[num_each_layer[i + 1]];
177                 //0号进程也参与计算
178                 int i_size = (num_each_layer[i + 1] + numprocs - 1) / (numprocs);
179                 for (int j = myid * i_size; j < min(num_each_layer[i + 1], (myid +
                        1) * i_size); j++)
180                 {
181                     for (int k = 0; k < num_each_layer[i + 2]; k++)
182                     {
183                         error[j] = 0.0;
184                         error[j] += layers[i + 1]->weights[k][j] * layers[i +
                                1]->delta[k];
185                         layers[i]->delta[j] = error[j] *
                                layers[num_layers]->derivative_activation_function(layers[i]->output
186                     }
187                 }
188                 //将数据汇总到0号进程
189                 if (myid == 0)
190                 {
191                     for (int procs_i = 1; procs_i < numprocs; procs_i++)
192                     {
193                         MPI_Recv(layers[i]->delta + procs_i * i_size, i_size,
```

```
                    MPI_FLOAT, procs_i, 96, MPI_COMM_WORLD, &status);
194             }
195         }
196         else
197         {
198             MPI_Send(layers[i]->delta + myid * i_size, i_size, MPI_FLOAT,
                    0, 96, MPI_COMM_WORLD);
199         }
200
201         delete[] error;
202     }
203     //cout<<"finish cal delta"<<endl;
204
205     //3.反向传播, weights和bias更新, 对k进行数据分配
206
207     MPI_Bcast(layers[0]->delta, num_each_layer[1], MPI_FLOAT, 0,
        MPI_COMM_WORLD); //从0号进程广播
208     int i_size = (num_each_layer[1] + numprocs - 1) /
        (numprocs);//0号进程也参与计算
209     for (int k = myid * i_size; k < min(num_each_layer[1], (myid + 1) *
        i_size); k++)
210     {
211         for (int j = 0; j < num_each_layer[0]; j++)
212         {
213             layers[0]->weights[k][j] -= study_rate * avr_X[j] *
                layers[0]->delta[k];
214         }
215         layers[0]->bias[k] -= study_rate * layers[0]->delta[k];
216     }
217     //将数据汇总到0号进程
218     if (myid == 0)
219     {
220         for (int procs_i = 1; procs_i < numprocs; procs_i++)
221         {
222             MPI_Recv(layers[0]->bias + procs_i * i_size, i_size,
                MPI_FLOAT, procs_i, 94, MPI_COMM_WORLD, &status);
223             for (int k = procs_i * i_size; k < min(num_each_layer[1],
                (procs_i + 1) * i_size); k++)
224             {
225                 MPI_Recv(layers[0]->weights[k], num_each_layer[0],
                    MPI_FLOAT, procs_i, 3000 + k, MPI_COMM_WORLD, &status);
226             }
227         }
228     }
229     else
230     {
231         MPI_Send(layers[0]->bias + myid * i_size, i_size, MPI_FLOAT, 0,
            94, MPI_COMM_WORLD);
232         for (int k = myid * i_size; k < min(num_each_layer[1], (myid + 1)
```

```
            * i_size); k++)
        {
            MPI_Send(layers[0]->weights[k], num_each_layer[0], MPI_FLOAT,
                0, 3000 + k, MPI_COMM_WORLD);
        }
    }
    //cout<<"finish first bp"<<endl;
    //同理
    for (int i = 1; i <= num_layers; i++)
    {
        MPI_Bcast(layers[i]->delta, num_each_layer[i + 1], MPI_FLOAT, 0,
            MPI_COMM_WORLD);
        MPI_Bcast(layers[i - 1]->output_nodes, num_each_layer[i],
            MPI_FLOAT, 0, MPI_COMM_WORLD);
        int i_size = (num_each_layer[i + 1] + numprocs - 1) / (numprocs);
        for (int k = myid * i_size; k < min(num_each_layer[i + 1], (myid +
            1) * i_size); k++)
        {
            for (int j = 0; j < num_each_layer[i]; j++)
            {
                layers[i]->weights[k][j] -= study_rate * layers[i -
                    1]->output_nodes[j] * layers[i]->delta[k];
            }
            layers[i]->bias[k] -= study_rate * layers[i]->delta[k];
        }
        if (myid == 0)
        {
            for (int procs_i = 1; procs_i < numprocs; procs_i++)
            {
                MPI_Recv(layers[i]->bias + procs_i * i_size, i_size,
                    MPI_FLOAT, procs_i, 95, MPI_COMM_WORLD, &status);
                for (int k = procs_i * i_size; k < min(num_each_layer[i +
                    1], (procs_i + 1) * i_size); k++)
                {
                    MPI_Recv(layers[i]->weights[k], num_each_layer[i],
                        MPI_FLOAT, procs_i, 2000 + k, MPI_COMM_WORLD,
                        &status);
                }
            }
        }
        else
        {
            MPI_Send(layers[i]->bias + myid * i_size, i_size, MPI_FLOAT,
                0, 95, MPI_COMM_WORLD);
            for (int k = myid * i_size; k < min(num_each_layer[i + 1],
                (myid + 1) * i_size); k++)
            {
                MPI_Send(layers[i]->weights[k], num_each_layer[i],
                    MPI_FLOAT, 0, 2000 + k, MPI_COMM_WORLD);
```

```
269                         }
270                     }
271 
272                 }
273                 //printf ("finish bp with index:%d\n",index);
274             }
275             // display();
276         }
277         QueryPerformanceCounter((LARGE_INTEGER*)&tail); // start time
278         time_mpi += tail - head;
279         printf("finish training\n");
280         std::cout << myid << "mpi_all:" << time_mpi * 1.0 / freq << "s" << endl;
281         delete[]avr_X;
282         delete[]avr_Y;
283 }
```

参考文献