南開大學
Nankai University

计算机学院

并行程序设计 **openMP** 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 5 月 15 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

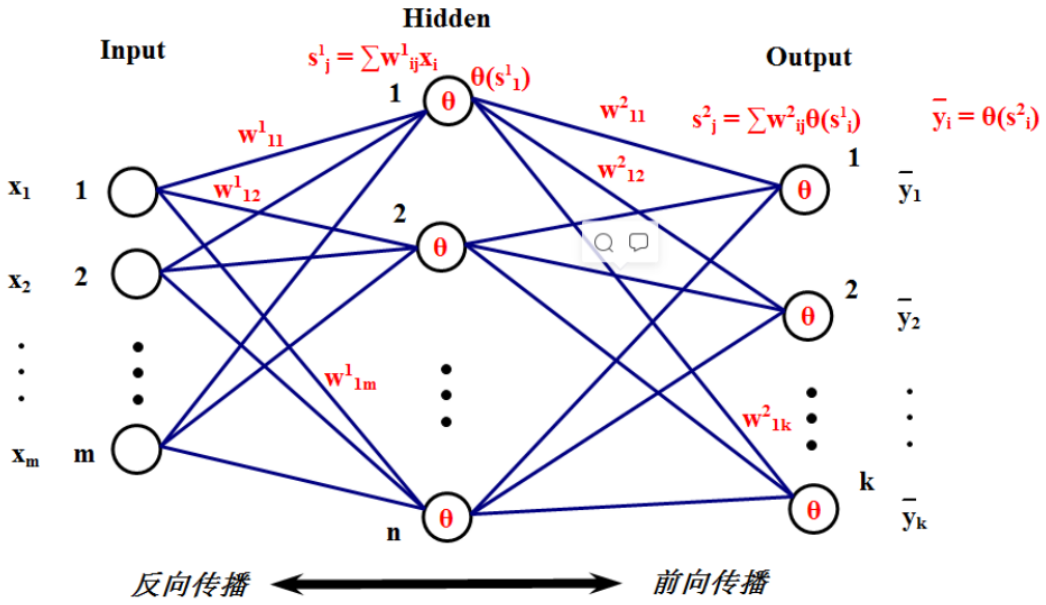

图 1.1: ANN 模型结构

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。

$$\begin{aligned} \boldsymbol{z}^{(l)} &= W^{(l)}\boldsymbol{x}^{(l-1)} + \boldsymbol{b}^{(l)} \\ \boldsymbol{x}^{(l)} &= f\left(\boldsymbol{z}^{(l)}\right) \end{aligned} \tag{1}$$

Algorithm 1 前向传播

```
function PREDICT(Face)
    hiddenLayer ← W1×X + bias
    outputLayer ← f(W2 * hiddenLayer + bias)
    return outputLayer
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。

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

1.2.2 反向传播与参数更新

对于一个训练数据 $\left(\boldsymbol{x}^{(i)}, \boldsymbol{y}^{(i)}\right)$，首先计算其代价函数:

$$\begin{aligned} E_{(i)} &= \frac{1}{2}\left\|\boldsymbol{y}^{(i)} - \boldsymbol{o}^{(i)}\right\| \\ &= \frac{1}{2}\sum_{k=1}^{n_L}\left(y_k^{(i)} - o_k^{(i)}\right)^2 \end{aligned} \tag{3}$$

所有数据的平均代价函数为:

$$E_{total} = \frac{1}{N}\sum_{i=1}^{N} E_{(i)} \tag{4}$$

采用梯度下降法更新参数:

$$\begin{aligned} W^{(l)} &= W^{(l)} - \mu\frac{\partial E_{\text{total}}}{\partial W^{(l)}} \\ &= W^{(l)} - \frac{\mu}{N}\sum_{i=1}^{N}\frac{\partial E_{(i)}}{\partial W^{(l)}} \\ \boldsymbol{b}^{(l)} &= \boldsymbol{b}^{(l)} - \mu\frac{\partial E_{\text{total}}}{\partial \boldsymbol{b}^{(l)}} \\ &= \boldsymbol{b}^{(l)} - \frac{\mu}{N}\sum_{i=1}^{N}\frac{\partial E_{(i)}}{\boldsymbol{b}^{(l)}} \end{aligned} \tag{5}$$

现计算其中的偏导数，记 $\delta_i^{(l)} \equiv \frac{\partial E}{\partial z_i^{(l)}}$，根据求导的链式法则，用向量形式可表示为:

$$\begin{aligned} \delta_i^{(L)} &= -\left(y_i - a_i^{(L)}\right) f'\left(z_i^{(L)}\right) && (1 \le i \le n_L) \\ \frac{\partial E}{\partial w_{ij}^{(L)}} &= \delta_i^{(L)} a_j^{(L-1)} && (1 \le i \le n_L, 1 \le j \le n_{L-1}) \end{aligned} \tag{6}$$

其中的 L 为神经网络的层数，为了简化，程序中取 L=2

Algorithm 2 反向传播训练参数

```
1: function TRAIN
2:     while 未达到迭代次数或收敛条件 do
3:         TrainVec()
4:         //更新 Weights 与 Bias
5:         for each layer reversely do
6:             Weights− = studyRate ∗ DeltaWeights
7:             Bias− = studyRate ∗ DeltaBias
8:             TrainVec()
9:         end for
10:    end while
11: end function
12:
13: function TRAINVEC
14:    //正向传播，计算 LayerOutput
15:    for each layer do
16:        LayerOutput ← sigmoid(LayerInput ∗ Weight + Bias)
17:    end for
18:    //求导，即计算 DeltaBias 与 DeltaWeights
19:    for each layer reversely do
20:        DeltaBias = (−0.1) ∗ (Label − LayerOutput) ∗ LayerOutput ∗ (1 − LayerOutput)
21:        DeltaWeights = DeltaBias ∗ LayerInput
22:    end for
23: 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。这里，由于迭代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为 128 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。接下来，改变不同训练样本数，即训练样本类别数 * 每个类别训练样本数，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，程序的时间复杂度为：

$$T = O(numEpoch*(numInput*numHidden1+numHidden1*numOutput)*numClass*numSamplePerClass) \tag{7}$$

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。

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 minibatch 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，进行公式的计算。问题中共有三大类循环可以优化。

1. 前向传播的循环, 可以在同一层内按照 weights 矩阵的不同列进行划分，为避免数据访问的冲突，每个线程计算不同列，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 openMP

具体任务划分方式与 pthread 相同，直接使用 openMP 相关 API 进行并行程序的实现。

openMP

```
void ANN_openMP::train (const int num_sample, float** _trainMat, float** _labelMat)
{

    printf ("begin training\n");
    float thre = 1e-2;
    float *avr_X = new float[num_each_layer[0]];
    float *avr_Y = new float[num_each_layer[num_layers + 1]];


    for (int epoch = 0; epoch < num_epoch; epoch++)
    {
        if (epoch % 50 == 0)
        {
            //  printf ("round%d:\n", epoch);
        }
        int index = 0;

```

```
18          while (index < num_sample)
19          {
20              for (int X_i = 0; X_i < num_each_layer[0]; X_i++) avr_X[X_i] = 0.0;
21              for (int Y_i = 0; Y_i < num_each_layer[num_layers + 1]; Y_i++)
                    avr_Y[Y_i] = 0.0;
22
23              for (int j = 0; j < num_each_layer[num_layers + 1]; j++)
24              {
25                  layers[num_layers]->delta[j] = 0.0;
26              }
27
28              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
29              {
30                  //前向传播
31                  #pragma omp parallel num_threads(NUM_THREADS)
32                  {
33                      #pragma omp for
34                      for (int i = 0; i < num_each_layer[1]; i++)
35                      {
36                          layers[0]->output_nodes[i] = 0.0;
37
38                          for (int j = 0; j < num_each_layer[0]; j++)
39                          {
40                              layers[0]->output_nodes[i] += layers[0]->weights[i][j]
                                    * _trainMat[index][j];
41                          }
42                          layers[0]->output_nodes[i] += layers[0]->bias[i];
43                          layers[0]->output_nodes[i] =
                                layers[0]->activation_function
                                (layers[0]->output_nodes[i]);
44                      }
45                  }
46                  for (int i_layer = 1; i_layer <= num_layers; i_layer++)
47                  {
48                      #pragma omp parallel num_threads(NUM_THREADS)
49                      {
50                          #pragma omp for
51                          for (int i = 0; i < num_each_layer[i_layer + 1]; i++)
52                          {
53                              layers[i_layer]->output_nodes[i] = 0.0;
54
55                              for (int j = 0; j < num_each_layer[i_layer]; j++)
56                              {
57                                  layers[i_layer]->output_nodes[i] +=
                                        layers[i_layer]->weights[i][j] * layers[i_layer
                                        - 1]->output_nodes[j];
58                              }
```

```
59                          layers[i_layer]->output_nodes[i] +=
                                layers[i_layer]->bias[i];
60                          layers[i_layer]->output_nodes[i] =
                                layers[i_layer]->activation_function
                                (layers[i_layer]->output_nodes[i]);
61
62                      }
63
64                  }
65
66              }
67
68              //计算loss, 即最后一层的delta,即该minibatch中所有loss的平均值
69              for (int j = 0; j < num_each_layer[num_layers + 1]; j++)
70              {
71                  //均方误差损失函数,batch内取平均
72                  layers[num_layers]->delta[j] +=
                        (layers[num_layers]->output_nodes[j] - _labelMat[index][j])
                        * layers[num_layers]->derivative_activation_function
                        (layers[num_layers]->output_nodes[j]);
73                  //交叉熵损失函数
74                  //layers[num_layers]->delta[j] +=
                        (layers[num_layers]->output_nodes[j] - _labelMat[index][j]);
75              }
76              // printf ("finish cal error\n");
77              for (int X_i = 0; X_i < num_each_layer[0]; X_i++) avr_X[X_i] +=
                    _trainMat[index][X_i];
78              for (int Y_i = 0; Y_i < num_each_layer[num_layers + 1]; Y_i++)
                    avr_Y[Y_i] += _labelMat[index][Y_i];
79          }
80
81          //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
82          if (index % batch_size == 0)
83          {
84              for (int j = 0; j < num_each_layer[num_layers + 1]; j++)
85              {
86                  if (batch_size == 0) printf ("wrong!\n");
87                  layers[num_layers]->delta[j] /= batch_size;
88                  avr_Y[j] /= batch_size;
89              }
90              for (int X_i = 0; X_i < num_each_layer[0]; X_i++) avr_X[X_i] /=
                    batch_size;
91
92          }
93          else
94          {
95              for (int j = 0; j < num_each_layer[num_layers + 1]; j++)
96              {
97                  if (index % batch_size == 0) printf ("wrong!\n");
```

```
98                  layers[num_layers]->delta[j] /= (index % batch_size);
99                  avr_Y[j] /= (index % batch_size);
100             }
101             for (int X_i = 0; X_i < num_each_layer[0]; X_i++) avr_X[X_i] /=
                    (index % batch_size);
102         }
103         //计算loss func, 仅用于输出
104         if ( index >= num_sample)
105         {
106             static int tt = 0;
107             float loss = 0.0;
108             for (int t = 0; t < num_each_layer[num_layers + 1]; ++t)
109             {
110                 loss += (layers[num_layers]->output_nodes[t] - avr_Y[t]) *
                        (layers[num_layers]->output_nodes[t] - avr_Y[t]);
111             }
112             //  printf ("第%d次训练: %0.12f\n", tt++, loss);
113         }
114
115         //反向传播更新参数
116
117         //计算每层的delta,行访问优化
118         for (int i = num_layers - 1; i >= 0; i--)
119         {
120             #pragma omp parallel num_threads(NUM_THREADS)
121             {
122                 float *error = new float[num_each_layer[i + 1]];
123                 #pragma omp for
124                 for (int j = 0; j < num_each_layer[i + 1]; j++)
125                 {
126                     error[j] = 0.0;
127                     for (int k = 0; k < num_each_layer[i + 2]; k++)
128                     {
129                         error[j] += layers[i + 1]->weights[k][j] * layers[i +
                                1]->delta[k];
130                     }
131                     layers[i]->delta[j] = error[j] *
                            layers[num_layers]->derivative_activation_function
                            (layers[i]->output_nodes[j]);
132                 }
133
134                 delete[]error;
135             }
136         }
137         //反向传播, weights和bias更新
138         #pragma omp parallel num_threads(NUM_THREADS)
139         {
140             #pragma omp for
141             for (int k = 0; k < num_each_layer[1]; k++)
```

```
142                 {
143
144                     for (int j = 0; j < num_each_layer[0]; j++)
145                     {
146                         layers[0]->weights[k][j] -= study_rate * avr_X[j] *
                                layers[0]->delta[k];
147                     }
148                     layers[0]->bias[k] -= study_rate * layers[0]->delta[k];
149                 }
150             }
151             for (int i = 1; i <= num_layers; i++)
152             {
153                 #pragma omp parallel num_threads(NUM_THREADS)
154                 {
155                     #pragma omp for
156                     for (int k = 0; k < num_each_layer[i + 1]; k++)
157                     {
158
159                         for (int j = 0; j < num_each_layer[i]; j++)
160                         {
161                             layers[i]->weights[k][j] -= study_rate * layers[i -
                                    1]->output_nodes[j] * layers[i]->delta[k];
162                         }
163                         layers[i]->bias[k] -= study_rate * layers[i]->delta[k];
164                     }
165                 }
166             }
167             //printf ("finish bp with index:%d\n",index);
168         }
169     }
170
171     // display();
172     printf ("finish training\n");
173     delete[]avr_X;
174     delete[]avr_Y;
175 }
```

3 实验和结果分析

因为笔记本为 AMD 架构，无法使用 Vtune 工具。测试了不同算法设计、openMP+SIMD、不同规模下、不同平台下的程序运行时间。

3.1 不同问题规模下串行算法与并行算法的对比

不同问题规模详细的定义如表1所示。其中，numInput、numOutput、numHidden1、numLayers、numClass、numSamplePerClass、numEpoch、batchSize 依次为输入层结点数、输出层结点数、隐藏层结点数、隐藏层层数、训练样本类别数、每个类别训练样本数、ANN 迭代次数、一次参数更新使用

表 1: 问题规模

	size1	size2	size3	size4	size5
numInput	1024	512	256	128	64
numOutput	1024	512	256	128	64
numHidden1	1024	512	256	128	64
numLayers	1	1	1	1	1
numClass	4	4	4	4	4
numSamplePerClass	16	16	16	16	16
numEpoch	128	128	128	128	128
batchSize	1	1	1	1	1

表 2: 不同问题规模下的程序运行时间 (单位：s)

	size1	size2	size3	size4	size5
串行	611.273785160	150.650142500	30.40502581	9.460616700	2.484375260
pthread_sem	165.5424523	40.28941561	10.21026395	2.87439668	1.08398819
openMP	154.081129780	38.885507360	7.831970010	2.593021310	0.936434260

的样本数目。在华为鲲鹏平台上的实际运行时间如表2所示。其对应的加速比如图3.2所示。

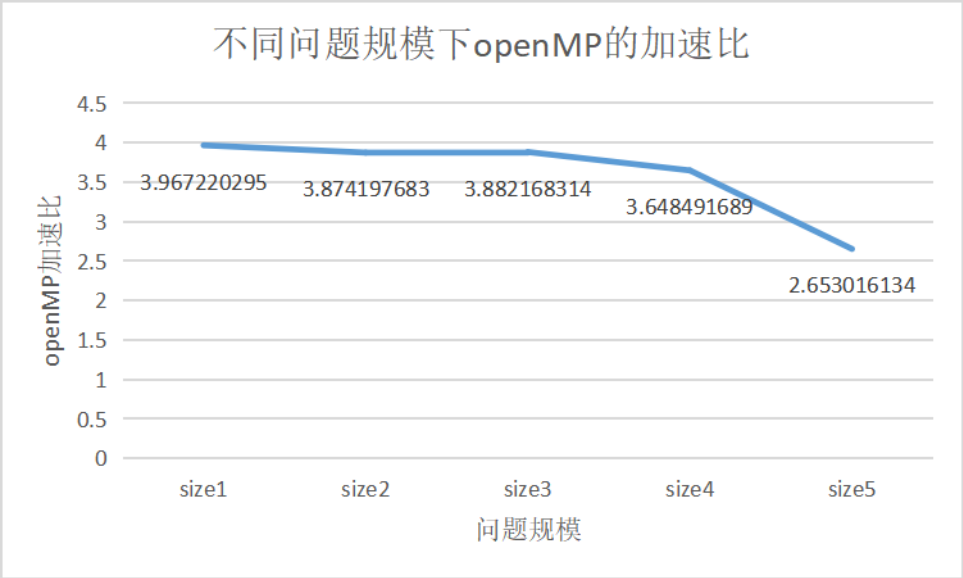

图 3.2: 不同问题规模下的并行程序加速比

根据并行部分的特性，这里隐藏层层数均为 1，ANN 迭代次数均为 128，由于迭代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为 128 不变。测试了不同输入层结点数、输出层结点数、隐藏层结点数的程序运行时间。

理论上来说，程序的时间复杂度为：

$$T = O(t*(numInput*numHidden1+numHidden1*numOutput)*numClass*numSamplePerClass) \tag{8}$$

其中对总训练时间影响较大的是循环中每层的 numInput*numOutput，决定总时间的是其中较小的值，故每次不同的规模中各层的维度设置相同。

表 3: 不同线程数的程序运行时间 (单位：s)

问题规模	size3							
分配线程数	1	2	3	4	5	6	7	8
串行	36.861629050	37.070465060	36.879099810	30.40502581	30.35391538	30.42887226	30.36589808	30.43123709
pthread_sem	39.04147453	19.60434686	13.41327387	10.21026395	8.38806622	7.30424905	6.44540705	6.43263342
openMP	36.766177690	18.535041120	12.62769445	9.54947765	7.89707267	6.6262456	5.81433008	5.28885976

通过多线程将其中的一层循环进行划分，理论上可以达到接近线程数的加速比。实际实验结果：

- 在问题规模较大时，与理论上基本吻合，基本达到了与线程数相当的加速比，结果是十分理想的;
- 在规模较小时，由于线程创建的开销，加速比较小，并行的优势还不足以体现，这也符合预期。除了线程创建的开销，还包括构建线程输入参数的开销。在 openMP 程序的设计中，线程的创建被声明在遍历每个样本的循环之内，故这里应该造成的一定的开销。

关于 pthread 与 openMP 的讨论：

- 二者在最终保证正确性的前提下，在较大数据规模下均达到了比较理想的加速比，但 pthread 代码的编写代价远远高于了 openMP。
- pthread 采用了静态的线程分配方式，在 train 函数的最开始进行线程的创建，在循环中间大量使用同步机制确保结果的正确性；而 openMP 中，我直接在循环中创建线程，并使用 openMP 线程回收时的隐式同步。结果表明，这两种方式的运行时间并没有明显的差异。原因可能如下：
  1. 矩阵运算按块划分，各线程任务量相当，十分容易达到负载均衡。
  2. 相比于内层循环大量的计算，外层循环中线程创建和回收的开销可以忽略。

3.2 不同线程数对比

测试了不同线程数时，朴素算法、pthreadsem、openMP 的程序运行时间与加速比，如表3所示，其中的问题规模定义如上节所述。其加速比如图3.3所示。

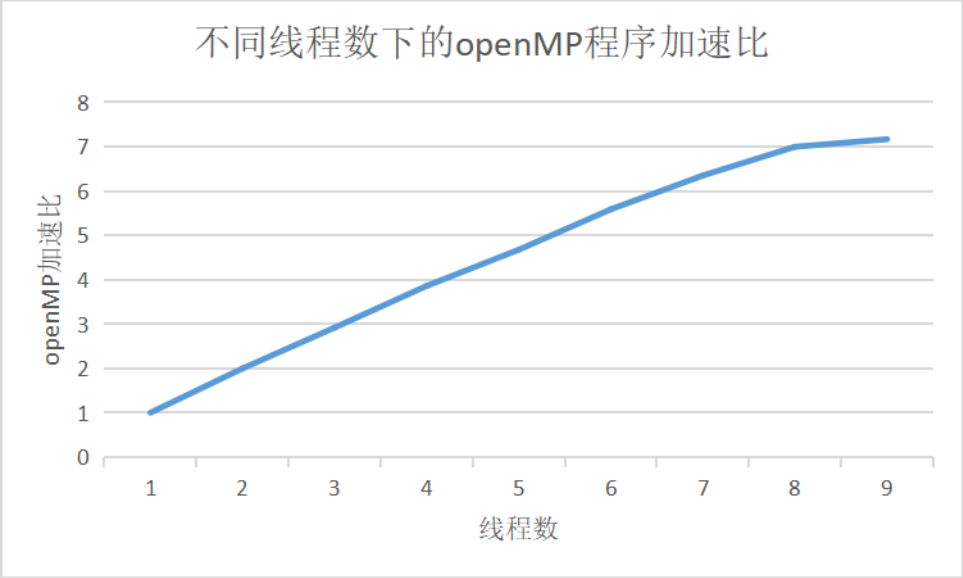

图 3.3: 不同问题规模下的并行程序加速比

表 4: 不同平台下的程序运行时间与加速比（单位：s）

问题规模	size1	
平台	x86	ARM
串行	41.6971	611.2737852
openMP	16.8548	154.0811298
openMP	2.47390061	3.967220295
pthread_sem	10.21026395	165.5424523
pthread_sem	4.083841535	3.692550018
openMP+SIMD	8.87588	123.3412443
openMP+SIMD	4.697798979	4.955956044

表 5: 不同数据划分的运行时间（s）

	串行	矩阵按列分配	矩阵按行分配
运行时间	41.6971	16.854	14.173

可以发现，在不同线程数下，使用 openMP 基本达到了接近线程数的加速比，其结果还是比较理想的。最大分配 8 个线程，子线程数为 8、9 时结果相近。

3.3 不同平台下对比

分别对比 x86 与 ARM 平台下的程序运行时间，理论上 ARM 应慢于 x86，但加速比应当接近，结果如表5所示，符合预期。其中 pthread 的线程数是子线程数，openMP 的线程数是全部线程数，所以差异可以理解。openMP+SIMD 使用 sse 和 noen 实现，4 个浮点数进行向量化，在 openMP 的基础上理论上有额外 4 倍的加速比，但与 pthread 中的实验相同，并未有显著的差异，原因应该也与上一次相似。在 x86 平台中程序运行较快，所以对于相同的问题规模，加速比更小，开销占比更大。

3.4 不同编程策略的并行算法的对比

以下实验均在 size1，4 线程，x86 平台下进行测试。

3.4.1 不同的任务分配方式

对于反向传播的循环，可以按行或按列划分任务，其 train 函数执行时间如表5所示。要额外注意的是如果按列划分，要单独考虑增加同步和私有变量的声明以保证结果的正确性，bias 计算被串行执行。按行划分要快于按列划分，这是可能是因为按行划分是线程内利用了空间局部性。

4 总结

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。实现了对于反向传播算法更新、求导的 SIMD 与多线程并行算法，达到了一定加速比。

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

参考文献