南開大學
Nankai University

计算机学院

并行程序设计实验报告

MPI 编程实验报告

姓名：张铭徐 张惠程

学号：2113615 2112241

专业：计算机科学与技术

2023 年 5 月 26 日

目录

1 实验目的和实验背景介绍

1.1 实验背景知识介绍

1.1.1 卷积操作背景知识介绍

卷积运算 [2] 是深度学习中非常常见的操作之一，用于提取图像、音频等数据中的特征。假设有两个矩阵 I 和 K，它们的维度分别为 $m \times n$ 和 $k \times k$，则它们的卷积运算结果为 S，维度为 $(m-k+1) \times (n-k+1)$。卷积运算的公式如下所示：

$$S_{i,j} = \sum_{u=1}^{k} \sum_{v=1}^{k} I_{i+u-1,j+v-1} K_{u,v} S$$

其中，$S_{i,j}$ 表示卷积运算结果中第 i 行第 j 列的元素，$I_{i+u-1,j+v-1}$ 表示输入矩阵中第 $(i+u-1)$ 行第 $(j+v-1)$ 列的元素，$K_{u,v}$ 表示卷积核矩阵中第 u 行第 v 列的元素。

在深度学习中，卷积运算通常用于卷积神经网络中的卷积层，用于提取图像、音频等数据中的特征。由于神经网络中通常需要进行大量的卷积运算，如果我们能够使用多线程策略对其进行加速，那么整体的训练和收敛速度将会大大提高，极大程度上减少算力的要求。

在计算机视觉的主流任务中，例如图像分类与目标检测等任务，往往都需要提取特征，在此时，卷积层便能起到很好的特征提取的效果，在目前效果比较好的框架中，例如人脸检测的 MTCNN[4]，就使用了三层级联的网络用于进行特征提取与人脸检验，其网络主要应用的，就是卷积和池化的操作，网络的整体架构如下图1.1所示：

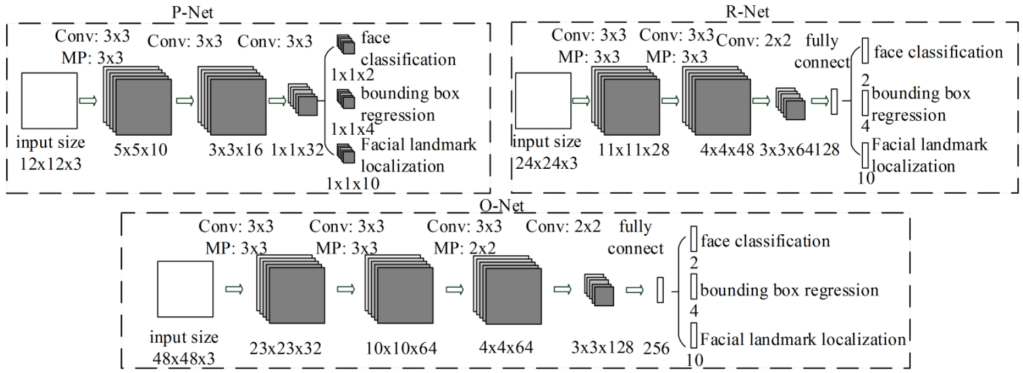

图 1.1: MTCNN's Framework

1.1.2 池化操作背景知识介绍

池化操作（Pooling Operation）[5] 是深度学习中常用的一种操作，用于减小特征图的空间维度，同时保留重要的特征信息。池化操作在卷积神经网络（CNN）等模型中广泛应用，用于降低计算复杂度、减少过拟合，并增强模型的平移不变性。

池化操作主要有两种类型：最大池化（Max Pooling）和平均池化（Average Pooling）。这里我们将重点介绍最大池化操作。

最大池化：给定输入特征图（或称为池化输入）X，最大池化操作通过将特征图划分为不重叠的区域，并在每个区域中选取最大值作为池化输出。具体而言，对于输入特征图的每个区域，最大池化操作选取该区域内的最大值作为对应的输出值。类似的，我们有平均池化的定义：给定输入特征图（或称为池化输入）X，平均池化操作通过将特征图划分为不重叠的区域，并在每个区域中计算特征值的

平均值作为池化输出。具体而言，对于输入特征图的每个区域，平均池化操作计算该区域内特征值的平均值作为对应的输出值。

数学上，最大池化和平均池化操作可以用以下公式表示：

$$\text{Max Pooling}(X)_{i,j,k} = \max_{m,n}(X_{(i\cdot\text{stride}+m),(j\cdot\text{stride}+n),k})$$

$$\text{Average Pooling}(X)_{i,j,k} = \frac{1}{\text{pooling\_size} \times \text{pooling\_size}} \sum_{m,n}(X_{(i\cdot\text{stride}+m),(j\cdot\text{stride}+n),k})$$

其中，$X_{i,j,k}$ 表示输入特征图 X 在第 i 行、第 j 列、第 k 个通道上的值。Max Pooling$(X)_{i,j,k}$ 表示最大池化操作的输出值，它是对应区域内的最大值；Average Pooling$(X)_{i,j,k}$ 表示平均池化操作的输出值，它是对应区域内特征值的平均值；stride 表示池化操作的步长（stride），用于控制池化窗口的移动步幅。通过调整步长，可以控制输出特征图的尺寸；pooling_size 表示池化窗口的大小，即池化操作在每个区域内考虑的特征值数量。

需要注意的是，池化操作在每个通道上是独立进行的，因此在应用池化操作时，针对输入特征图的每个通道分别进行池化。

最大池化操作的效果是通过保留每个区域内的最大值，来提取出特征图的显著特征。通过降低特征图的空间维度，最大池化操作可以减少参数数量，并提高模型的计算效率。平均池化操作通过计算特征图区域内的平均值来提取特征图的平滑特征，能够对输入特征图进行降维和平滑处理。它可以在一定程度上减少噪声的影响，并且也有助于提取图像或特征图的整体统计特征

总结起来，池化操作通过划分输入特征图为不重叠的区域，并选取每个区域内的最大值或者平均值作为输出值，来减小特征图的空间维度。它是深度学习中常用的操作之一，用于增强模型的平移不变性和降低计算复杂度。

1.1.3 Embedding(嵌入操作背景知识介绍)

嵌入层（Embedding Layer）[1] 是深度学习中常用的一种层类型，用于将离散的输入特征映射到低维连续向量空间。嵌入层通过学习每个离散特征的低维度表示，可以捕捉特征之间的语义关系和相似性。

给定一个离散的输入特征 x，例如单词或类别的索引，嵌入层将其映射为一个低维的嵌入向量 e。嵌入向量的维度通常由设计者指定，例如 d 维。嵌入层的参数是一个嵌入矩阵 W，其大小为 $V \times d$，其中 V 是离散特征的总数目。

嵌入操作可以表示为：

$$e = \text{Embedding}(x) = Wx$$

其中，W 是嵌入矩阵，x 是输入特征的索引，e 是对应的嵌入向量。

嵌入层的参数 W 是通过训练过程学习得到的。在训练过程中，通过最小化某个损失函数，模型会更新嵌入矩阵 W 的值，使得嵌入向量能够更好地表示特征之间的语义关系和相似性。

嵌入层可以用于各种深度学习任务，如文本分类、推荐系统、序列建模等。它将离散的输入特征转换为连续的低维嵌入向量，提供了一种更有效地表示和处理离散特征的方法。

总结起来，嵌入层通过学习将离散的输入特征映射到低维连续向量空间，提供了一种有效的特征表示方法。通过嵌入操作，输入特征 x 被映射为对应的嵌入向量 e，并且嵌入矩阵 W 的参数通过训练

过程学习得到。这样的嵌入向量可以用于后续的模型训练和推断。

1.1.4 PCA(主成分分析) 背景知识介绍

主成分分析（Principal Component Analysis，PCA）[3] 是一种常用的数据降维和特征提取技术，用于发现数据集中的主要变化方向。

给定一个包含 n 个样本的数据集，每个样本有 m 个特征，可以构建一个 $n \times m$ 的数据矩阵 X，其中每一行表示一个样本，每一列表示一个特征。PCA 的目标是通过线性变换，将原始数据投影到一个新的坐标系中，使得投影后的数据具有最大的方差。

- 1. 数据中心化：首先，对数据进行中心化操作，即从每个特征维度中减去其均值，以确保数据的均值为零。通过计算每个特征的均值 μ，可以将数据矩阵 X 中的每个元素减去相应的均值，得到中心化后的数据矩阵 $\hat{X}$。

  $$\hat{X}_{ij} = X_{ij} - \mu_j$$

  其中，$\hat{X}_{ij}$ 表示中心化后的数据矩阵 $\hat{X}$ 的元素，X_{ij} 表示原始数据矩阵 X 的元素，μ_j 表示第 j 个特征的均值。

- 2. 协方差矩阵：接下来，计算中心化后的数据矩阵 $\hat{X}$ 的协方差矩阵 C。协方差矩阵用于描述数据特征之间的相关性和方差。

  $$C = \frac{1}{n-1}\hat{X}^T\hat{X}$$

  其中，$\hat{X}^T$ 表示中心化后的数据矩阵 $\hat{X}$ 的转置。

- 3. 特征值分解：对协方差矩阵 C 进行特征值分解，得到特征值和对应的特征向量。特征值表示数据中的主要方差，特征向量表示对应于主要方差的方向。

  $$CV = \lambda V$$

  其中，V 是由特征向量组成的矩阵，λ 是特征值的对角矩阵。

- 4. 选择主成分：根据特征值的大小，选择前 k 个特征向量作为主成分，其中 k 是降维后的维度。这些特征向量对应于最大的特征值，表示数据中的主要变化方向。

- 5. 数据投影：将中心化后的数据矩阵 $\hat{X}$ 投影到选择的主成分上，得到降维后的数据矩阵 Y：

  $$Y = \hat{X}V_k$$

  其中，V_k 是选择的前 k 个特征向量组成的矩阵。

通过 PCA 降维，我们可以将原始数据投影到一个较低维度的子空间中，保留了数据中最重要的变化方向。这样可以减少特征维度，提高计算效率，并且在某些情况下，可以更好地可视化和解释数据。

总结起来，PCA 通过数据中心化、计算协方差矩阵、特征值分解、选择主成分和数据投影等步骤，将高维的原始数据降低到较低维度的表示。通过选择主要方差所对应的特征向量，PCA 帮助我们发现数据集中的主要变化方向，并提供了一种数据降维和特征提取的方法。

1.1.5 MPI 知识简介

MPI，全称为 Message Passing Interface，即消息传递接口，是一种并行编程模型，主要用于分布式内存系统，如集群和超级计算机。在 MPI 模型中，每个进程都有自己的内存空间，进程之间通过发送和接收消息进行通信。这种模型的主要优点是可扩展性强，可以在各种硬件和网络配置上运行。MPI 的主要操作包括：

- 点对点通信：一个进程向另一个进程发送消息，或从另一个进程接收消息。这是最基本的通信模式，可以使用 MPI_Send 和 MPI_Recv 函数来实现。
- 集体通信：一个进程向所有其他进程发送消息，或从所有其他进程接收消息。这是一种更高级的通信模式，可以使用 MPI_Bcast（广播）、MPI_Scatter（分散）、MPI_Gather（收集）等函数来实现。
- 数据分发和收集：一个进程将数据分发给所有其他进程，或从所有其他进程收集数据。这是一种常用的数据处理模式，可以使用 MPI_Scatter 和 MPI_Gather 函数来实现。

MPI 还提供了许多其他的功能，例如：

- 通信域：MPI 允许用户创建自定义的通信域，即一组可以相互通信的进程。这可以让我们更灵活地组织你的并行程序。
- 派生数据类型：MPI 允许定义自己的数据类型，这可以让我们更方便地处理复杂的数据结构。
- 虚拟拓扑：MPI 允许创建虚拟的进程拓扑，例如网格或环形，这可以让我们更方便地编写需要特定拓扑结构的并行程序。
- 非阻塞通信：MPI 提供了非阻塞版本的发送和接收函数，例如 MPI_Isend 和 MPI_Irecv，这可以令通信和计算之间进行重叠，从而提高程序的效率。

深度学习中的许多操作，如卷积，池化和嵌入，都可以通过并行化来加速。MPI 可以在这些操作中发挥作用，因为它可以将任务分配给多个进程，每个进程在自己的数据集上工作，然后将结果合并。

对于一般 MPI 的编程范式，我们整理出了以下的可视化图表1.2：

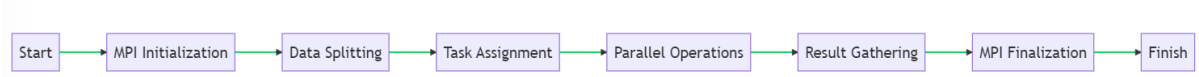

图 1.2: MPI_process_table

1.2 实验平台介绍

本次实验平台在 ARM 平台上进行实验，具体为学校统一配置的鲲鹏服务器，在节点 ss2113615 下进行实验，具体而言，我们编写提交任务脚本，然后将该脚本向实验 master 主节点提交，然后节点自动为我们分配计算资源进行运行，最后在文件目录中找到对应的输出文件。

2 池化 MPI 优化

2.1 朴素算法设计思想

池化操作是神经网络中常用的一种操作，用于减小特征图的空间尺寸并提取特征。类似于 BP 算法，我们可以通过 MPI 优化池化操作来提高其效率，并需要证明算法的正确性。

最大池化没有明确的公式，但可以通过以下方式表示：

给定输入数据 X 和池化区域大小 k，则最大池化的输出 Y 可以表示为：

$$Y_{i,j,c} = \max\left(X_{(i\cdot k):(i\cdot k+k),(j\cdot k):(j\cdot k+k),c}\right)$$

其中，i 和 j 表示输出的行和列索引，c 表示通道索引。$X_{(i\cdot k):(i\cdot k+k),(j\cdot k):(j\cdot k+k),c}$ 表示输入数据的池化区域。

针对上述的讨论过程，我们给出算法流程图1：

Algorithm 1 朴素算法的池化操作

输入: 输入数据
输出: 池化结果

```
for 每个输入通道 do
    for 每个池化窗口 do
        // 计算池化窗口内的最大值
        max_value ← 池化窗口内的最大值
            // 将最大值添加到池化结果
        将 max_value 添加到池化结果中
    end
end
```

2.2 MPI 优化设计思想

我们在考虑设计算法前，最重要的思路就是证明算法的正确性，没有正确性证明的算法，效率再高都无济于事，我们考虑证明 MPI 优化 BP 算法的可能性：首先我们假设：

- 假设输入特征图被正确地分布到各个 MPI 进程，确保每个进程上的特征图是完整且一致的。
- 各个 MPI 进程之间可以进行通信，并且 MPI 操作的结果符合预期。

接下来我们来证明算法的正确性：

- 特征提取的正确性：在每个 MPI 进程上，通过池化操作提取局部特征。由于每个进程上的特征图是完整的，因此在每个进程上提取的局部特征是准确的。
- 特征合并的正确性：通过 MPI 操作，在各个进程之间进行特征的合并，将局部特征合并得到全局特征。MPI 操作的正确性保证了合并操作的准确性。
- 特征传播的正确性：将全局特征传播到所有 MPI 进程，确保所有进程都使用相同的全局特征。

综上所述，通过 MPI 优化池化操作，在正确分布特征图、准确提取局部特征、正确合并特征和传播全局特征的前提下，可以保证最终得到准确的特征表示。这证明了 MPI 优化后的池化操作在神经网络训练过程中的正确性。

3 PCA(主成分分析)MPI 优化

3.1 朴素算法设计思想

Algorithm 2 朴素 PCA 算法

输入: 数据, 主成分数
输出: 降维结果

```
for i ← 0 to num_data do
    // 计算数据的均值
    mean ← 计算数据在每个维度上的均值
     将数据每个维度上的值减去对应维度的均值
     cov_matrix ← 计算数据的协方差矩阵
     eigenvalues, eigenvectors ← 计算协方差矩阵的特征值和特征向量
     selected_eigenvectors ← 选择特征值最大的几个特征向量
     result[i] ← 将数据点 i 投影到主成分上
end
```

朴素版的 PCA 只需要利用我们上面说明的算法原理进行实现即可，在这里我们给出算法流程图，全部的代码和项目文件请参考后文附的 GitHub 项目链接。

3.2 MPI 优化算法设计思想

正如我们在前面算法介绍部分介绍的那样：PCA 是一种常用的降维算法，用于将高维数据映射到低维空间。PCA 的目标是通过线性变换找到数据中的主成分，即方差最大的方向。PCA 通过计算数据的协方差矩阵的特征值和特征向量，选择最大的几个特征值对应的特征向量作为主成分，将数据投影到这些主成分上，实现降维。我们发现，事实上 PCA 的本质仍然是矩阵的运算，既然涉及到了矩阵的运算，我们便显而易见的想要通过并行化技术对计算过程进行加速优化。

Algorithm 3 使用 MPI 的并行 PCA

输入: 数据, 嵌入向量
输出: 结果

```
for i ← start to end do
    mean ← 计算数据在每个维度上的均值
     将数据每个维度上的值减去对应维度的均值
     cov_matrix ← 计算数据的协方差矩阵
     eigenvalues, eigenvectors ← 计算协方差矩阵的特征值和特征向量
     selected_eigenvectors ← 选择特征值最大的几个特征向量
     result[i] ← 将数据点 i 投影到主成分上
end
for i ← 0 to num_data do
    // 全局归约
    MPI_Allreduce(result[i], MPI_SUM)
end
```

上述算法中，我们首先计算数据在每个维度上的均值，并将数据减去均值，以实现数据的中心化。然后计算中心化后数据的协方差矩阵。接着，我们计算协方差矩阵的特征值和特征向量，并选择最大的几个特征值对应的特征向量作为主成分。最后，将数据点投影到选定的主成分上，得到降维后的结

果。在并行化过程中，每个进程根据其负责的工作范围执行计算，并通过 MPI_Allreduce 操作将局部结果与其他进程的结果进行求和，以得到全局一致的结果矩阵。

4 实验结果分析

4.1 PCA 进行 MPI 优化

我们通过 C++ 库中的 chrono 库计算程序的耗时，chrono 功能十分强大；提供了一组类和函数，可以进行时间点和时间间隔的计算、表示和操作。Chrono 库的计时功能可以用于测量代码执行时间、实现定时器功能等。我们在这里使用了如下所示的代码测定程序的开始时间和结束时间，应用差量法测出程序的执行时间。

```
auto start = std::chrono::high_resolution_clock::now();
auto end = std::chrono::high_resolution_clock::now();
```

我们在下面给出了不同数据规模下，不同算法的运行时间对比的数据表，如下表1所示，此外，我们绘制出随数据规模增长，程序运行时间的折线图，如图4.3所示：值得注意的是，我们 PCA 中，n,m 为输入矩阵的规模，我们默认要求 PCA 取方差最大的 k 个值，在这里我们取 $k = 10$。

表 1: PCA 算法测试时间

n,m	Origin	MPI	MPI_OpenMP	SIMD_MPI_OpenMP
2000 200	0.725664	0.0595061	0.0649667	0.0736112
4000 400	1.54789	0.327868	0.366709	0.414305
6000 600	5.26589	0.957157	0.863804	0.84937
8000 800	12.59848	3.56623	2.021	1.98605
10000 1000	46.7366	5.24536	3.57661	3.66901
15000 1500	203.324	38.24	16.092	15.053
20000 2000	806.897	120.34	60.5729	69.4669
30000 3000	1980.26	743.685	147.069	148.581

4.1.1 结果分析

根据我们所提供的表格和折线图，我们可以分析出以下几点：

- PCA_Origin（原始算法）：PCA_Origin 是原始的 PCA 算法实现。我们没有使用任何并行计算或优化技术，故该算法在处理大型数据集时，运行时间较长。并且随着数据集大小的增加，PCA_Origin 的运行时间近似指数增长。
- PCA_MPI (MPI 并行算法)：PCA_MPI 使用 MPI 并行计算框架实现。MPI 允许多个进程在不同的处理器上并行执行任务，从而加快了算法的运行速度。在我们提供的数据表格中，我们可以观察到 PCA_MPI 的运行时间明显优于 PCA_Origin。但是，随着数据集大小的增加，PCA_MPI 的运行时间也逐渐增加，但增长速度相对较慢。
- PCA_MPI_OpenMP（MPI 和 OpenMP 并行算法）：PCA_MPI_OpenMP 结合了 MPI 和 OpenMP 并行计算框架的优势。MPI 用于在多个节点间进行通信和数据分发，而 OpenMP 用于在每个节点内进行多线程并行计算。通过并行化的计算和数据通信，PCA_MPI_OpenMP 显著减少了运行时间。可以观察到，在数据表中，PCA_MPI_OpenMP 相对于 PCA_MPI 进一步减少了运行时间。

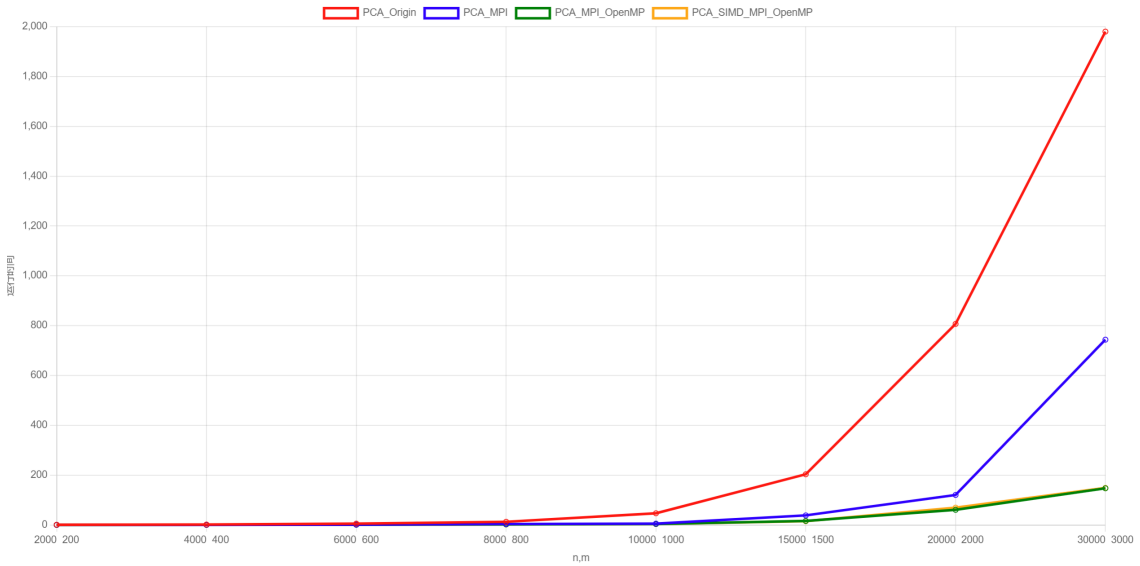

图 4.3: PCA 不同算法程序运行时间

- PCA_SIMD_MPI_OpenMP（MPI、OpenMP 和 SIMD 并行算法）：PCA_AVX_MPI_OpenMP 在 PCA_MPI_OpenMP 的基础上，引入了 SIMD 指令集的优化。SIMD 指令集允许同时处理多个数据元素，从而提高了计算效率。我们可以观察到 PCA_AVX_MPI_OpenMP 在处理较大数据集时具有更好的性能，相比于其他算法，其运行时间更短。

4.2 池化算法 MPI 优化

同样的，我们应用同样的方法，对池化操作的结果进行了整合分析，汇总出了如下表2所示的数据和如图4.4所示的折线图，值得注意的是，我们采用的 Pooling 是最大池化。

表 2: Pooling 算法测试时间

n,m	pooling	Origin	MPI	MPI_OpenMP	SIMD_OpenMP_MPI
1000 1000	2	0.00500842	0.00443384	0.000726752	0.00201667
2000 2000	4	0.0125016	0.0158015	0.00597195	0.00341615
4000 4000	8	0.0844329	0.00666671	0.0119086	0.0137254
8000 8000	16	0.447501	0.0295291	0.0724601	0.0634052
16000 16000	32	2.25992	0.965222	0.302678	0.224352
32000 32000	64	5.74283	1.65425	0.698356	0.507988

4.2.1 结果分析

我们发现对于四种算法的时间折线图的数据分布与我们前面的 PCA 算法的数据分布很相似，是因为都使用了相同的并行优化策略；此外，我们的数据规模也都是成比例增加的，故所得到的折线图与上面的类似。在这里我们考虑一些上面没有讨论过的问题：

- 对于后两种算法的运行时间相近：输入规模可能不足以展现 AVX 指令集的优势。AVX 指令集适用于大规模的数据并行计算，能够同时处理多个数据元素。如果输入规模较小，无法充分利用 AVX 的向量化能力，那么 AVX_OpenMP_MPI 算法的性能优势就不明显。

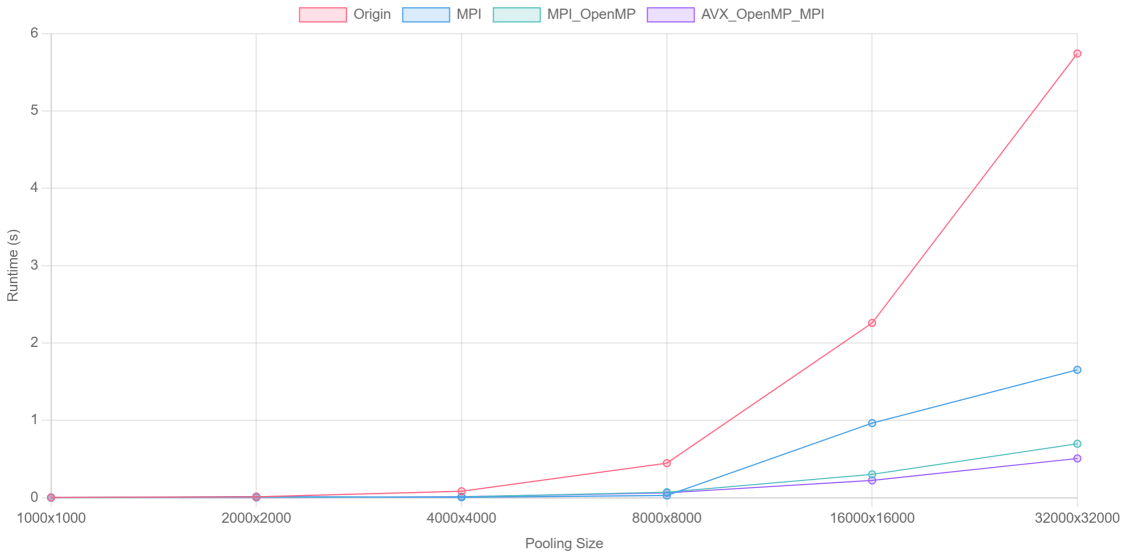

图 4.4: Max-Pooling 算法运行时间图

- 对于低数据分布的情况下朴素算法的运行时间低于其他几种：我们推测是由于数据规模造成的；无论是对于 MPI，SIMD 还是 OpenMP 的几种并行化编程技巧，在最开始时，计算机都需要用时间对其进行初始化操作，而在朴素算法则无需初始化，所以产生了如图所示的性能差异。

5 profiling 分析

基于上述的结果，我们在 X86 的平台上对几种算法进行了 profiling 分析，我们对于几种算法的性能分析时，首先进行了控制变量原则，即控制输入数据的规模相同；具体而言，针对 PCA 算法，我们设置输入数据的规模为：$n = 10000$,$m = 1000$,$k = 10$，对于 Pooling 算法，我们控制输入数据为 $n = 16000$,$m = 16000$,$pooling = 32$，我们使用了 X86 平台上的 VTune 性能分析工具对几种算法的性能进行了详细分析，结果如下表3,4所示：

表 3: PCA 算法性能指标分析

Method	CPU_Time	parallelism	CPU_utilization	LLC_MISS	CPI Rate
origin	37.393	6.20%	5.90%	971707946	1.104
MPI	30.612	6.10%	5.40%	648172118	1.194
MPI_Openmp	22.253	6.00%	4.80%	0	0.228
AVX_MPI_Openmp	20.955	6.10%	3.90%	1650693	0.32

Method	Instruction	L1 bound	L2 bound	L3 bound	memory_bound
origin	153870000000	0.00%	0.10%	16.70%	49.40%
MPI	114369800000	2.20%	0	36.20%	68.20%
MPI_Openmp	438858400000	0	0	0	5.10%
AVX_MPI_Openmp	2985857700000	0.30%	0	0.10%	0

我们首先对给出的几个性能指标做以介绍：

- CPU Time: CPU 执行任务的总时间，单位为秒。

表 4: Pooling 操作性能指标对比

Method	CPU_Time	parallelism	CPU_utilization	LLC_MISS	CPI Rate
origin	19.367	6.10%	4.30%	550231	0.481
MPI	21.275	6.10%	3.50%	22009240	0.482
MPI_Openmp	24.317	6.10%	4.20%	1100462	0.445
MPI_AVX_Openmp	24.158	6.10%	4.10%	550231	0.441

Method	Instruction retired	L1 bound	L2 bound	L3 bound	memory_bound
origin	180177400000	11.9	0	0.1	17.7
MPI	197105400000	11.4	0	0.4	14.4
MPI_Openmp	244699300000	9.4	0.2	0.1	13.2
MPI_AVX_Openmp	24492930000	8.4	0.2	0.1	14.20

- Parallelism: 并行度，表示程序中并行执行的程度。
- CPU_utilization: CPU 利用率，表示 CPU 在执行任务时的利用效率。
- LLC_MISS_COUNT: 最后一级缓存 (LLC) 的缺失次数，表示程序访问缓存的效率。
- CPI Rate: 每指令周期数 (CPI) 的比率，表示指令执行的效率。
- Instruction: 已执行的指令数，表示 CPU 执行的指令数量。
- Memory bound: 程序受限于内存访问的程度。
- L1/L2/L3 bound: 程序受限于缓存的程度。

我们考虑对尝试通过 VT 分析的性能指标来对程序运行时间进行解释：

- CPU 时间直接反映了程序的执行时间，我们在这里很直观的看到，不同的并行优化算法相较于朴素算法，都具有极高的提升，说明我们的并行策略对 PCA 有极高的加速效果。我们从并行度指标可以看出，其他算法对于朴素算法而言，并行度确实有一定的提升。
- CPU 利用率来看，其实几种算法的 CPU 利用率基本一致，甚至于朴素算法在 PCA 算法中是最高的，我们推测这是由于我们算法对于 CPU 的利用率还不够高，数据规模还不够大，无法发挥出 CPU 的计算能力，可以适当增大数据规模。我们后续可以针对一些极限数据做更加详细的测试，我们推测当数据规模足够大时，并行算法的 CPU 利用率应该是最高的。
- 从 cache 的角度来看，我们的并行算法毫无疑问特别成功，针对 PCA 算法，缓存的缺失次数都显著小于平凡算法，此外，越多的并行化策略的使用也会让缓存命中率变高。我们推测原因是：Cache 块的亲和性，即当任务被分配给不同的处理单元/线程时，通常会尽量保证每个处理单元/线程处理的数据在物理上是连续存放的。这种亲和性可以提高缓存的命中率，因为连续存放的数据更有可能存放在同一个 Cache 块中，减少了 Cache 块的换入和换出操作，从而减小了 LLC miss 的可能性。
- 我们从 L1/L2/L3 bound 的角度考虑：我们发现 PCA 的算法优化程序越高，即采用了越多的并行策略，其缓存的 bound 越小，即缓存和内存对于程序的程序运行时间的影响越小，说明我们的算法能够很好的利用计算机的并行架构。但是我们看到，对于 AVX_MPI_OpenMP 并行优化的算法反而有 LLC 的 miss，而使用 MPI_OpenMP 的 LLC 为 0，这可能是因为数据分布上存在

不均匀的情况，导致某些数据访问频率较高，而其他数据则很少访问。这种不均匀的数据分布可能导致 LLC 缓存的 miss，因为缓存无法有效地预取或保持所有数据。或者还可能因为线程间竞争：AVX_MPI_OpenMP 算法中的并行优化可能引入了线程之间的竞争条件，例如互斥锁或其他同步机制，这可能导致数据访问时的延迟和缓存冲突，从而增加了 LLC 的 miss。

接下来我们考虑对池化操作分析，池化操作存在一些异常，即平凡算法的运行时间反而比优化操作的运行时间快，我们推测可能是因为朴素算法具有较低的计算复杂度和较少的计算量。朴素算法在每个池化窗口内直接计算最大值，并将其添加到池化结果中，而不需要进行额外的计算或数据重排。这种简单的计算方式可以在较短的时间内完成。

相比之下，优化算法可能引入了更多的计算步骤或数据重排操作，以提高计算效率或利用硬件加速。虽然这些优化技术可以在某些情况下显著提高性能，但在某些特定的数据集或场景下，可能会因为额外的计算或数据操作导致更长的执行时间。

因此，对于池化操作，选择适当的算法取决于具体的应用场景和需求。如果对计算速度要求较高且不需要额外的优化，朴素算法可能是一个合理的选择。但如果需要更高的计算效率和优化性能，则可以考虑使用更复杂的优化算法。

6 实验总结与反思

6.1 关于 MPI 编程

我们在本次实验中，更深刻的理解了 MPI 编程的一些问题，我们整理出以下几点：

- 并行计算模型：MPI 适用于并行计算模型中的分布式内存系统，其中多个计算节点相互连接，并且每个节点具有自己的本地内存。
- 进程间通信：MPI 的主要目的是在多个进程之间进行通信。进程可以通过发送和接收消息来交换数据，并可以进行同步操作以保证数据的一致性。
- 点对点通信：MPI 提供了点对点通信操作，包括发送和接收。发送者可以将消息发送给接收者，并通过标识符指定消息的类型。
- 并行性和负载平衡：MPI 编程允许在多个进程之间分配计算任务，实现并行性。同时，需要考虑负载平衡，确保每个进程获得合理的计算负载，避免性能瓶颈。
- 有关于算法设计和数据划分：我们需要将计算任务划分为适当的子任务，并在不同的进程上并行执行。这需要考虑数据划分、通信开销、同步机制等因素，针对不同的问题，我们应该灵活的根据任务需求和任务特点进行灵活的选择。

6.2 实验分工

本次实验较为困难，包括需要在 ARM 平台上进行实验，在此前的几次并行实验中，我们都是在 x86 平台上进行实验，在前期摸索过程中，感谢助教们耐心的讲解答疑!! 没有助教的指导，实验环境配置的过程会艰难无数倍。

本次实验对于几种优化算法的论文资料，有张惠程搜集，张铭徐负责阅读文献，弄清原理；张铭徐负责 PCA，池化部分的编程，张惠程负责 embedding 层，池化操作的编程；并且一同在鲲鹏服务器上完成程序的运行，在本地的 VTune 性能剖析工具中完成对程序的 profiling 分析。有关于本次实验的所有源文件和代码，都可以在并行项目仓库处找到。

参考文献

[1] Yoshua Bengio, Réjean Ducharme, Pascal Vincent, and Christian Jauvin. A neural probabilistic language model. *Journal of machine learning research*, 3(Feb):1137–1155, 2003.

[2] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. *Advances in Neural Information Processing Systems*, 25:1097–1105, 2012.

[3] MIT. Singular Value Decomposition. http://web.mit.edu/be.400/www/SVD/Singular_Value_Decomposition.htm, Accessed 2023.

[4] Kaipeng Zhang, Zhanpeng Zhang, Zhifeng Li, and Yu Qiao. Joint face detection and alignment using multitask cascaded convolutional networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 680–688, 2016.

[5] Shuchang Zhou, Yibing Wu, Zhiwei Ni, Xinyu Zhou, and He Wen. Compression with max and average pooling. *arXiv preprint arXiv:1806.10723*, 2018.