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 [2] 运算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：

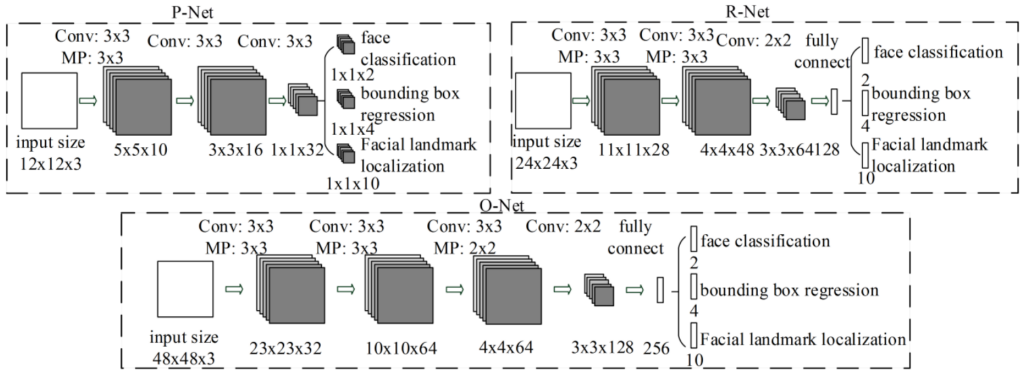

图 1.1: MTCNN's Framework

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：

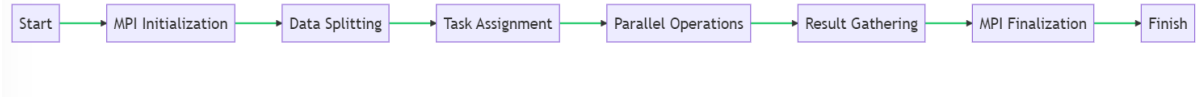

图 1.2: MPI_process_table

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 卷积操作 MPI 优化

2.1 朴素算法设计思想

朴素版的卷积运算算法使用四重循环进行计算，对于输入矩阵 $I \in \mathbb{R}^{m\times n}$ 和卷积核 $K \in \mathbb{R}^{k\times k}$，可以得到其输出矩阵 $S \in \mathbb{R}^{(m-k+1)\times(n-k+1)}$：

$$S_{i,j} = \sum_{u=1}^{k}\sum_{v=1}^{k} I_{i+u-1,j+v-1}K_{u,v} \quad (i = 1,\ldots,m-k+1;\ j = 1,\ldots,n-k+1)$$

算法的时间复杂度为 $O(n^4)$，在卷积核较大时，计算代价会比较高。对于朴素算法，其算法流程图如算法 3 所示：

Algorithm 1 naive-Convolution

Input: 输入矩阵 I 和卷积核 K
Output: 输出矩阵 S
$m \leftarrow$ 输入矩阵 I 的行数 $n \leftarrow$ 输入矩阵 I 的列数 $k \leftarrow$ 卷积核 K 的大小 初始化输出矩阵 S 为 $(m-k+1)\times(n-k+1)$ 的零矩阵 **for** $i \leftarrow 1$ **to** $m-k+1$ **do**
  for $j \leftarrow 1$ **to** $n-k+1$ **do**
    $S_{i,j} \leftarrow 0$ **for** $u \leftarrow 1$ **to** k **do**
      for $v \leftarrow 1$ **to** k **do**
        $S_{i,j} \leftarrow S_{i,j} + I_{i+u-1,j+v-1} \times K_{u,v}$
      end
    end
  end
end

2.2 MPI 优化设计思想

卷积操作是 CNN 中的重要组成部分，但由于大量的矩阵运算往往需要极高的算力要求，我们发现：卷积操作的本质是矩阵的操作，而矩阵的操作是可以并行化的，于是我们考察卷积操作的并行化，通过 MPI 的集合通信操作，可以将局部计算结果进行有效的汇总，确保最终的卷积结果的正确性。他的正确性和设计思想是显然的，对于算法的正确性如下所示：

- 卷积操作是可并行的，因为每个输出元素可以独立计算，不需要依赖其他元素。
- 使用 MPI 进行优化时，可以将输入数据和卷积核分割成多个块，并分配给不同的 MPI 进程进行计算。
- 每个 MPI 进程只处理自己分配的块，计算出局部的卷积结果。
- 最后，使用 MPI 的集合通信操作将所有进程的局部卷积结果汇总到主进程，得到最终的卷积结果。

不过我们需要注意的是：卷积操作的 MPI 优化在实现时需要考虑数据的划分和通信的开销，确保任务的划分合理，并且在通信过程中进行数据的同步和汇总，以保证最终结果的正确性。同时，还需要针对具体的硬件和并行计算环境进行优化，以提高计算性能和效率。针对上述的讨论结果，我们可以给出算法流程图2, 如下所示：

Algorithm 2 卷积操作的 MPI 优化算法

Input: 输入数据和卷积核
Output: 卷积结果
将输入数据和卷积核广播给所有 MPI 进程
将输入数据和卷积核分割为多个块，并分配给各个 MPI 进程
for 每个 *MPI 进程* **do**
    在本地计算块的局部卷积结果
end
使用 MPI 的集合通信操作，将局部卷积结果汇总到主进程
主进程输出最终的卷积结果

3 Embedding(嵌入操作)MPI 优化

3.1 朴素算法设计思想

Algorithm 3 朴素版嵌入

输入: 数据, 嵌入向量
输出: 结果

```
num_data ← 数据大小
embedding_size ← 嵌入向量维度
result_size ← 结果维度

for i ← 0 to num_data do
    for j ← 0 to embedding_size do
        for k ← 0 to result_size do
            result[i][k] ← result[i][k] + embeddings[j][k] × data[i]
        end
    end
end
```

朴素版的 embedding 层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

在我们第一节中，介绍了嵌入操作的主要原理，我们发现嵌入操作的实质便是通过矩阵乘法将一系列输入数据嵌入为若干维度的向量，对于这种计算密集型的任务，我们显而易见的想要对其使用 MPI 优化，我们下面来考虑使用 MPI 优化的算法正确性，首先我们考虑嵌入结果的正确性：

- 每个进程根据其负责的工作范围执行嵌入操作，根据给定的输入数据和嵌入矩阵，更新局部结果矩阵的对应元素。
- 每个线程在自己的工作范围内并行执行嵌入操作，可以通过 SIMD 加速计算。
- 在进行结果的全局归约之后，每个进程都会通过 MPI_Allreduce 操作将局部结果与其他进程的结果进行求和，得到全局一致的结果矩阵。

接下来是结果的一致性：

- 每个进程在进行嵌入操作时，只更新自己负责工作范围内的结果矩阵，不会影响其他进程。
- 在进行结果的全局归约时，使用 MPI_Allreduce 操作将每个进程的局部结果与其他进程的结果进行求和，确保所有进程最终得到的结果矩阵是一致的。

综合以上两个方面的考虑，可以得出该嵌入操作算法使用 MPI 进行优化后的正确性。那么接下来我们可以通过算法的原理，给出算法的流程图4：

Algorithm 4 使用 MPI 的并行嵌入

输入: 数据, 嵌入向量
输出: 结果

```
for i ← start to end do
    for j ← 0 to embedding_size do
        for k ← 0 to result_size do
            result[i][k] ← result[i][k] + embeddings[j][k] × data[i]
        end
    end
end
for i ← 0 to num_data do
    for k ← 0 to result_size do
        // 全局归约
        MPI_Allreduce(result[i][k], MPI_SUM)
    end
end
```

4 实验结果分析

4.1 Conv 卷积操作结果分析

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：

4.1.1 结果分析

我们观察上面的结果，可以发现以下几点：

- 平凡算法（Origin）的运行时间随着信号长度和卷积核长度的增加而逐渐增加，同时比其余优化算法的时间高很多，这是由于平凡算法的计算复杂度较高，在每个输出位置上，需要执行大量的乘法和累加操作。随着输入规模的增加，计算量也增加，导致了运行时间大量增加。

表 1: 卷积操作程序运行时间图

signal_length	kernel_length	Origin	MPI	MPI_OpenMP	SIMD_OpenMP_MPI
100000	8000	0.772419	0.194989	0.115987	0.105485
200000	16000	3.1691	0.806252	0.574984	0.474485
300000	24000	7.12401	1.8042	1.04989	1.06939
400000	32000	12.6769	3.18735	1.98786	1.89599
500000	40000	19.8055	4.96612	3.00127	2.96107
600000	48000	28.7963	7.12824	4.26987	4.24801
700000	56000	38.8059	9.70485	6.24531	5.86649
800000	64000	50.7671	12.6991	7.67589	7.76565
900000	72000	66.2713	17.5021	9.97546	9.99211
1000000	80000	91.2357	23.1332	11.99654	12.3811

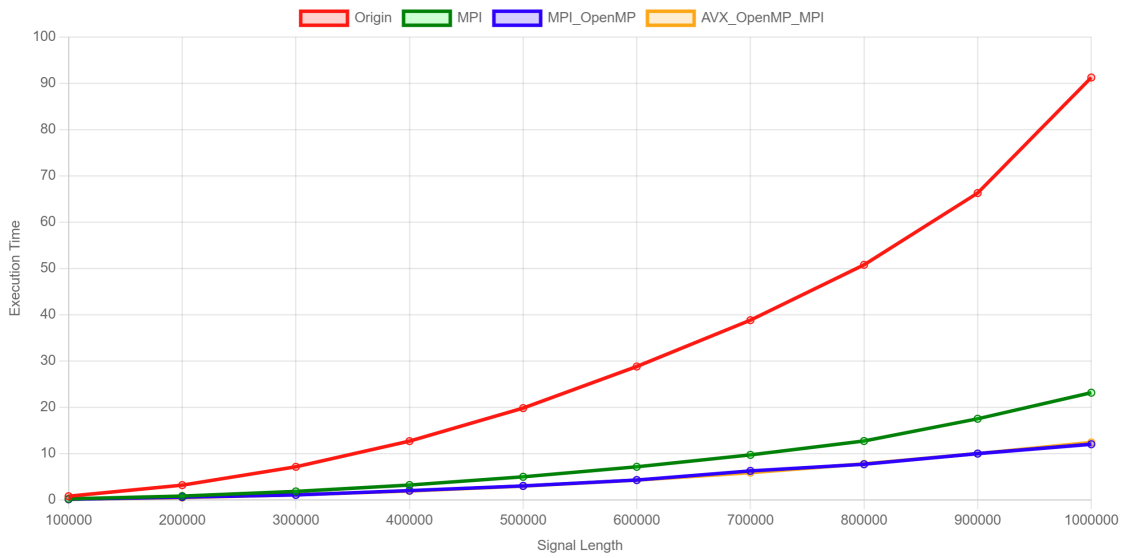

图 4.3: Conv 各种算法运行时间图

- MPI 优化算法在性能上相对于平凡算法有所提升。在 MPI 优化中，多个计算节点之间进行通信和协调，每个节点独立地执行一部分卷积计算。这样可以将工作负载分布到多个节点上，减少计算时间。随着输入规模的增加，MPI 优化的性能提升更加明显，我们推测其原因是通过 MPI 编程，计算机可以利用更多的计算资源进行并行计算。

- 对于 MPI 和 OpenMP 优化算法，我们采用了 MPI 和 OpenMP 并行编程模型，将 OpenMP 和 MPI 共同作用在该问题上。MPI 用于跨多个计算节点的并行化，而 OpenMP 用于在每个计算节点内部的多线程并行化。这种结合可以同时利用多个计算节点和每个节点内部的多个线程进行并行计算，以进一步提高性能。从数据中可以看出，MPI 和 OpenMP 优化的运行时间相对于 MPI 优化有所减少，原因为在每个节点内部的并行化计算减少了计算时间。

- SIMD、OpenMP 和 MPI 共同优化的算法进一步提高了性能。SIMD 可以在一个时钟周期内同时处理多个数据。该方法结合了 SIMD、OpenMP 和 MPI，并发地利用向量化指令和多线程并行，以提高卷积运算的速度。从数据中可以观察到，SIMD、OpenMP 和 MPI 共同优化的算法相对于其他优化方法，其运行时间更短，这是由于并行化计算和向量化指令的结合所带来的计算效率提升。

- 但是我们注意到，事实上，对于后两种方法而言，其相差的并不是很多，但是从理论上分析，使用 SIMD 进行优化后运行时间应该更短，效率应该更高才对，我们推测出现上述情况的原因可能是数据规模的限制。当数据规模较小时，向量化带来的优势可能被限制，并行化和向量化的开销可能超过了它们所带来的性能提升。所以，在较小的数据规模下，OpenMP_MPI 优化和 OpenMP_MPI_SIMD 优化的差异可能不太明显。

4.2 embedding 嵌入操作结果分析

同样的，我们应用同样的方法，对嵌入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

表 2: Embedding 运行时间比较

input_size	embedding_dim	Origin	MPI	MPI_OpenMP	SIMD_OpenMP_MPI
600000	1800	729.235	179.6515	140.89841	135.8489
500000	1500	179.798	64.2789	43.2365	45.14654
400000	1200	45.656	19.7265	13.1215	12.79865
300000	900	13.689	5.05952	3.5478	3.72877
200000	600	5.789	1.61169	1.4988	1.49291
100000	300	0.999	0.317183	0.35987	0.329065
50000	150	0.16584	0.0734297	0.078964	0.0693496
25000	75	0.019545	0.020658	0.0149876	0.0139778

4.2.1 结果分析

同样的，我们对数据进行分析，与卷积操作类似的，四种算法的程序运行时间基本上成下降趋势，优化效果越来越好，我们在这里分析一些上面没有关注的地方：

- 在较小的数据规模下，我们发现甚至有可能朴素算法的程序运行要远远低于其他三种，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

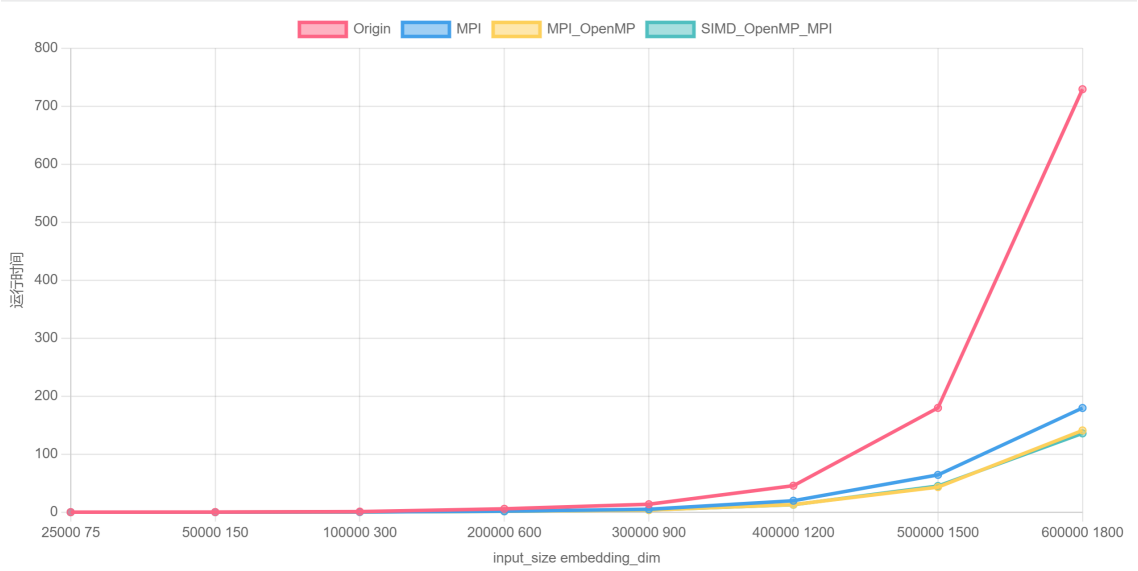

图 4.4: 嵌入操作程序运行时间折线图

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，我们推测这就导致了在表中 $input\_size = 25000$ 时，朴素算法的运行时间最低的原因。

- 我们考虑算法非线性变化的问题：理论上我们的数据都是成比例变化的，对应时间复杂度应该是线性变化的，但是事实上，我们看数据变化的趋势，每一段折线的斜率都不一致，并且随着数据规模增大，斜率一直在增加，我们推测原因可能是因为并行化技术的影响：在使用并行化技术时，如 MPI 和 OpenMP，存在一定的并行化开销。这包括线程间的通信、数据同步等开销。随着数据规模的增加，这些开销可能会更加显著，导致运行时间的非线性增长；此外，当数据规模增加时，也可能会出现更多的内存访问或处理器资源竞争，从而导致运行时间的非线性增长。

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，针对卷积算法，我们设置输入数据的规模为：$signal\_length = 100000$,$kernel\_length = 8000$，对于 embedding 算法，我们控制输入数据为 $input\_size = 500000$,$embedding\_din = 1500$，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3,4,5,6所示：

表 3: 卷积操作性能指标分析 1

Method	CPU Time	Parallelism	CPU Utilization	LLC MISS COUNT
origin_MPI	8.329	6.10%	4.50%	0
AVX_MPI	1.443	6.10%	3.90%	0
origin	28.735	5.70%	4.40%	169895
AVX_openMP_MPI	1.356	6.70%	4.30%	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: 卷积操作性能指标分析 2

Method	CPI Rate	Instruction	L1 bound	L2 bound	L3 bound	Memory Bound
origin_MPI	0.282	133887600000	0.5	0	0.1%	0
AVX_MPI	0.28	23460000000	0.1	0	0	0
origin	0.713	1.68535E+11	0.70%	0.20%	0.1%	0
AVX_openMP_MPI	0.27	159813600000	0	0	0	0

表 5: 嵌入操作性能指标对比 1

Method	CPU Time	Parallelism	CPU Utilization	LLC MISS COUNT	CPI Rate
origin	62.23	6.20%	3.60%	6052541	0.506
MPI	68.708	6.10%	3.10%	5502310	0.462
MPI_AVX_Openmp	20.453	5.90%	3.60%	2200924	0.364

表 6: 嵌入操作性能指标对比 2

Method	Instruction	L1 bound	L2 bound	L3 bound	Memory Bound
origin	559688900000	8.9%	0.1%	0.2%	20.50%
MPI	644195500000	7.8%	0.1%	0.2%	11.70%
MPI_AVX_Openmp	16727900000	10.50%	1.1	1.6	21.40%

- 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卷积操作有极高的加速效果。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，甚至于朴素算法在这里面还不算很低，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。但是在 embedding 和卷积操作中，由于算法的时间复杂度不是很高，增大数据规模首先出现的问题是内存占用率过高，导致程序有可能无法正常运行。
- 从 cache 的角度来看，我们的并行算法毫无疑问特别成功，无论是嵌入操作还是卷积操作，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：我们发现对于嵌入操作而言，使用多种并行化策略进行优化，反而对应的值要大，我们考虑原因可能是因为 AVX_MPI-OpenMP 共同优化的算法通过并行化和向量化操作，可以更好地利用数据的重用性。这意味着相同的数据可以在不同的计算单元中共享，这就可能导致我们的程序运行时间要更加依赖于对应的缓存是否命中。

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.