基于MPI的并行KNN算法实现

作者	邮箱
吴佳卫	jason_wooo@qq.com

引言

MPI是实现并行算法的一种常用的通信协议，作者基于MPI使用c++语言实现了KNN算法的并行化框架。这篇文档首先会对KNN算法以及MPI通信协议进行简单的说明，然后重点介绍本文中实现的基于MPI的并行KNN算法，最后介绍实验数据集并展示实验结果。

目录

一、KNN算法

k近邻法（k-nearest neighbor, kNN）是一种基本分类与回归方法，其基本做法是：给定测试实例，基于某种距离度量找出训练集中与其最靠近的k个实例点，然后基于这k个最近邻的信息来进行预测。KNN算法一共有三个核心要素：距离度量、k值的选择以及分类决策规则。

1.1 距离度量

特征空间中的两个实例点的距离是两个实例点相似程度的反映。KNN算法的第一个关键步骤就是衡量每一个测试集样本与训练集样本之间的距离。常用的距离度量方式有两种，分别是曼哈顿距离与欧式距离.

曼哈顿距离又称为街区距离，计算表达式如下：

$$L_1(x_i, x_j) = \sum_{l=1}^{n} |x_i^{(l)} - x_j^{(l)}|$$

欧式距离的计算公式如下：

$$L_2(x_i, x_j) = (\sum_{l=1}^{n} |x_i^{(l)} - x_j^{(l)}|^2)^{\frac{1}{2}}$$

1.2 k值的选择

k值的选择会对k近邻法的结果产生重大影响。在应用中，k值一般取一个比较小的数值，通常采用交叉验证法来选取最优的k值。

1.3 分类决策规则

k近邻法中的分类决策规则往往是多数表决，即由输入实例的k个邻近的训练实例中的多数类，决定输入实例的类。

二、MPI

2.1 MPI简介

MPI是一种基于消息传递的并行编程技术，为我们标准定义了一组具有可移植性的编程接口，解决了多线程程序运行过程中各个线程互相通信的问题。

MPI程序是基于消息传递的并行程序。消息传递指的是并行执行的各个进程具有自己独立的堆栈和代码段，作为互不相关的多个程序独立执行，进程之间的信息交互完全通过显示地调用通信函数来完成。在接下来的部分会介绍我们实现算法过程中使用到的一些MPI函数。

```
MPI_Init(&argc, &argv);
```

MPI_INIT是MPI程序的第一个调用。它完成MPI程序所有的初始化工作，所有MPI程序 的第一条可执行语句都是这条语句。其中，argc为变量数目，argv为变量数组，两个参数均来自main函数的参数。

```
MPI_Finalize();
```

MPI_FINALIZE是MPI程序的最后一个调用，它结束MPI程序的运行。它是MPI程序的最 后一条可执行语句，否则程序的运行结果是不可预知的。

```
MPI_Comm_rank(MPI_COMM_WORLD, &myid);
```

这一调用返回调用进程在给定的通信域中的进程标识号，有了这一标识号，不同的进程就可以将自身和其它的进程区别开来，实现各进程的并行和协作。因此我们在编程时只要通过判断myid的值就可以将不同的进程区分开来。

```
MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
```

这一调用返回给定的通信域中所包括的进程的个数，不同的进程通过这一调用得知在给 定的通信域中一共有多少个进程在并行执行。

```
MPI_Bcast(
    void * data_p, //通信消息缓冲区的起始地址
    int count, //通信消息缓冲区中的数据个数
    MPI_Datatype datatype, //通信消息缓冲区中的数据类型
    int root, //发送广播的根的序列号
    MPI_Comm comm, //通信子
);
```

MPI_BCAST是从一个序列号为root的进程将一条消息广播发送到组内的所有进程,包括它本身在内.调用时组内所有成员都使用同一个comm和root,其结果是将根的通信消息缓冲区中的消息拷贝到其他所有进程中去。

```
MPI_Scatter(
    void* send_data, //通信消息缓冲区的起始地址，储存在根进程中
    int send_count, //具体需要给每个进程发送的数据的个数
    MPI_Datatype send_datatype, //发送数据的类型
    void* recv_data, //接收缓存，缓存recv_count个数据
    int recv_count, //与send_count的值相同
    MPI_Datatype recv_datatype, //与send_datatype的值相同
    int root, //root进程的编号
    MPI_Comm comm //通信子
);
```

MPI_Scatter是从一个序列号为root的进程将一个array的数据分配到每一个进程中，包括它本身在内。array中的第一个元素发送给0号进程，第二个元素则发送给1号进程，以此类推。通常send_count等于array的元素个数除以进程个数。

```
MPI_Gather(
    void* send_data, //通信消息缓冲区的起始地址，储存在每一个进程中
    int send_count, //具体每个进程要发送的发送的数据的个数
    MPI_Datatype send_datatype, //发送数据的类型
    void* recv_data, //接收来自所有进程的消息的缓存，储存在根进程中
    int recv_count, //与send_count的值相同
    MPI_Datatype recv_datatype, //与send_datatype的值相同
    int root, //root进程的编号
    MPI_Comm comm //通信子
);
```

MPI_Gather和MPI_Scatter刚好相反，他的作用是从所有的进程中将每个进程的数据集中到根进程中，同样根据进程的编号对array元素排序。

```
MPI_Allreduce(
    void* send_data, //待发送数据地址，储存在每一个进程中
    void* recv_data, //待接收数据地址，储存在每一个进程中
    int count, //要发送的发送的数据的个数
    MPI_Datatype datatype, //发送数据的类型
    MPI_Op op, //具体的约归操作
    MPI_Comm comm //通信子
);
```

将通信子内各进程的同一个变量参与规约计算，并输出结果拷贝进每一个进程中。其实相当于先做MPI_Reduce，然后再做MPI_Bcast。

```
MPI_Barrier(MPI_Comm);
```

阻止调用直到communicator中所有进程已经完成调用。任意进程的调用只能在所有communicator中的成员已经开始调用之后进行。通常用于同步所有的进程。

```
MPI_Wtime();
```

用于执行MPI运行时间的计量，通常需要和MPI_Barrier函数一起使用，返回值为当前的时间节点，以秒为单位。

除了这些函数以外MPI还为我们定义了很多的函数，上面只展示了我们实现算法过程中使用到的一些函数。其余部分的函数可以根据需要通过参考文献部分中《高性能计算并行编程技术：MPI并行程序设计》手册来学习。

2.2 MPI安装及配置（以VS2019为例）

2.2.1 MPI的下载及安装

下载地址：

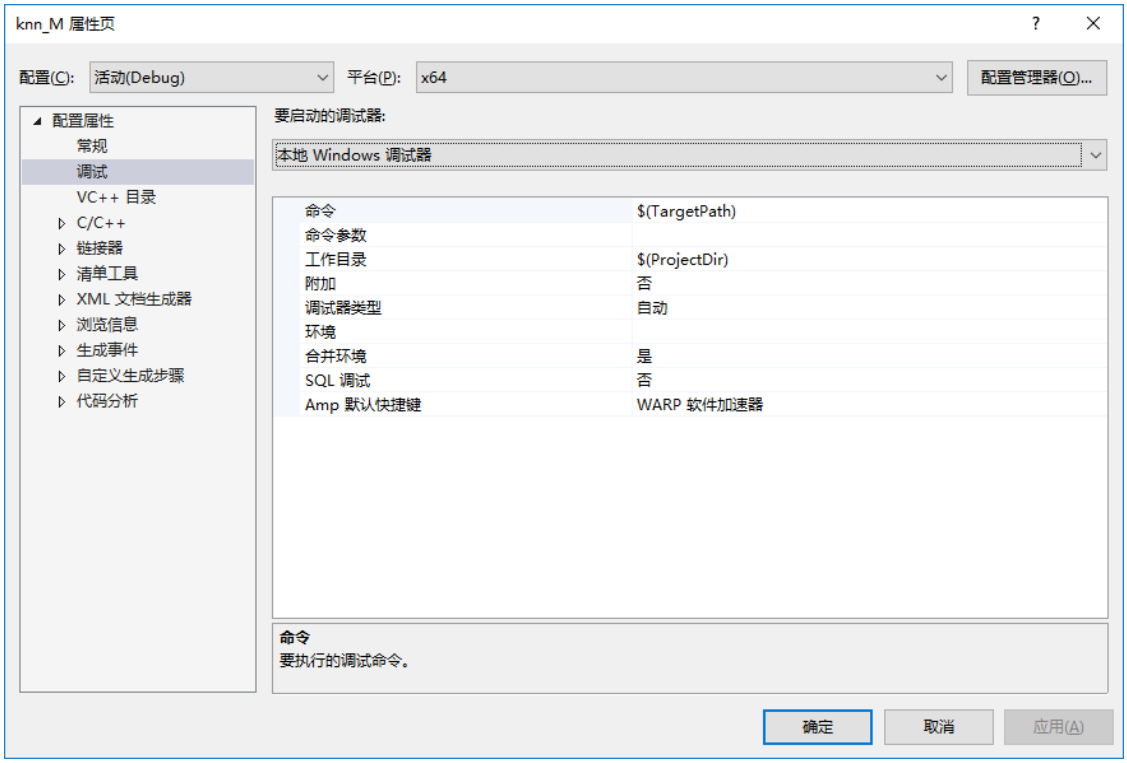
https://docs.microsoft.com/en-us/message-passing-interface/microsoft-mpi

下载完成后直接选定文件夹安装即可。

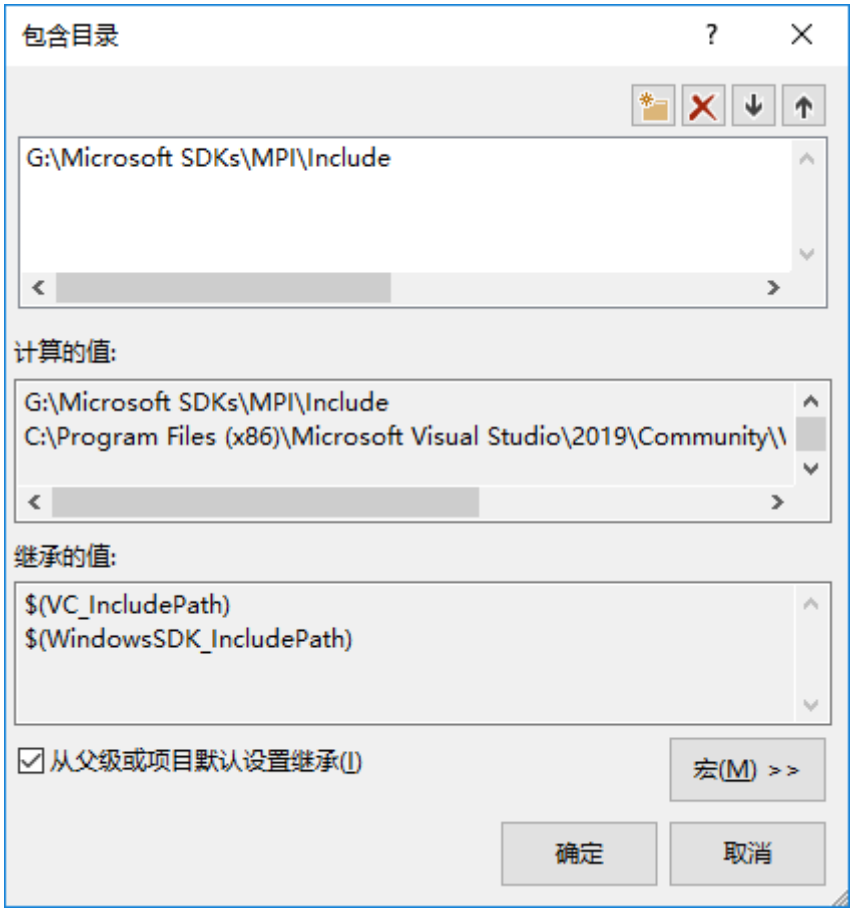
2.2.2 MPI的配置

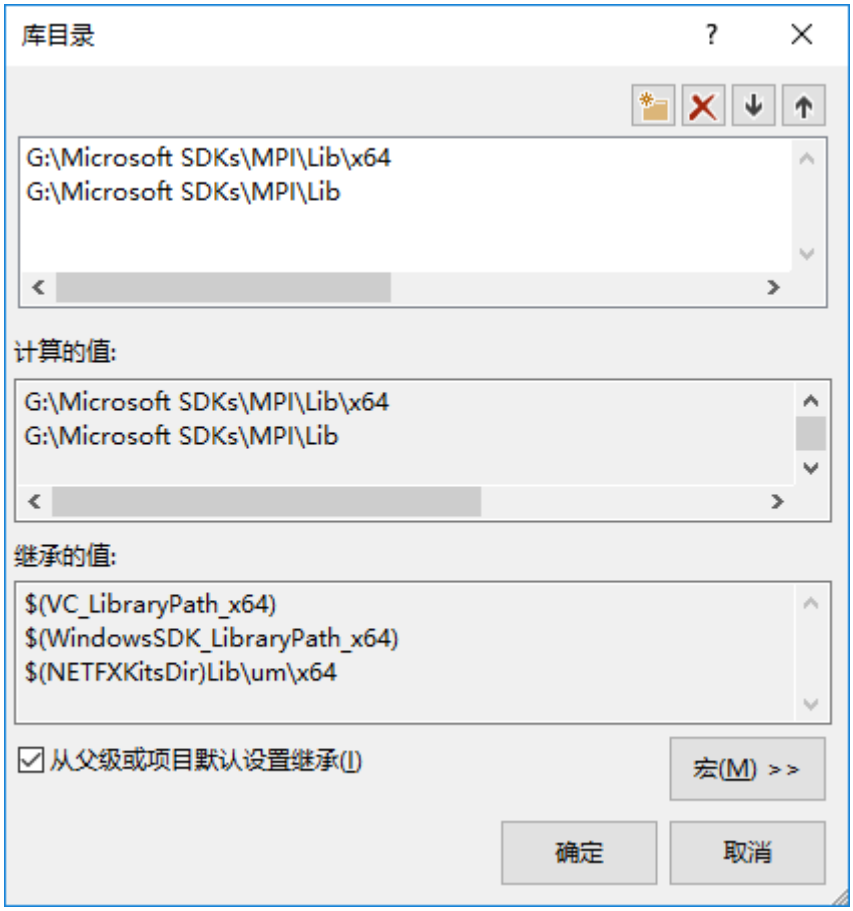
我的MPI安装地址为 G:\Microsoft SDKs\MPI，安装成功的话这个地址下应该包括Include，Lib以及License三个文件夹，接下来的配置都会以这个地址为例。

1. 新建一个Visual Studio项目，右键打开项目属性页。

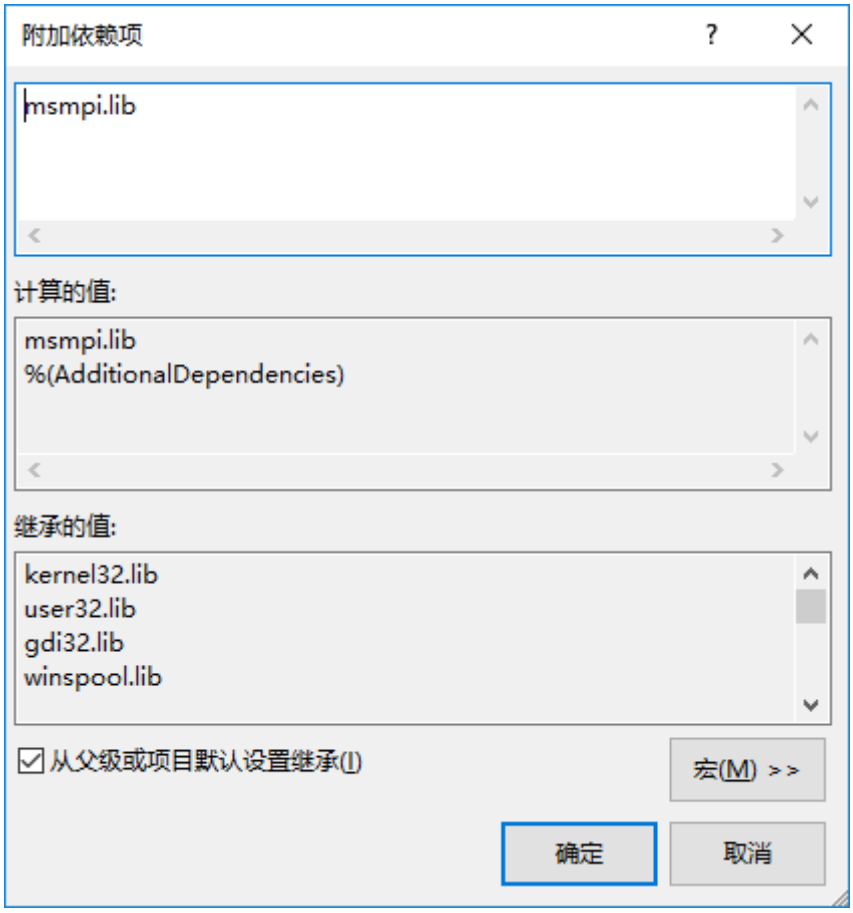


2. 进入”VC++目录“标签页下，配置“包含目录”及“库目录”两项。





3. 进入”连接器-输入“标签页下，配置“附加依赖项”。



4. 点击应用即可完成配置。

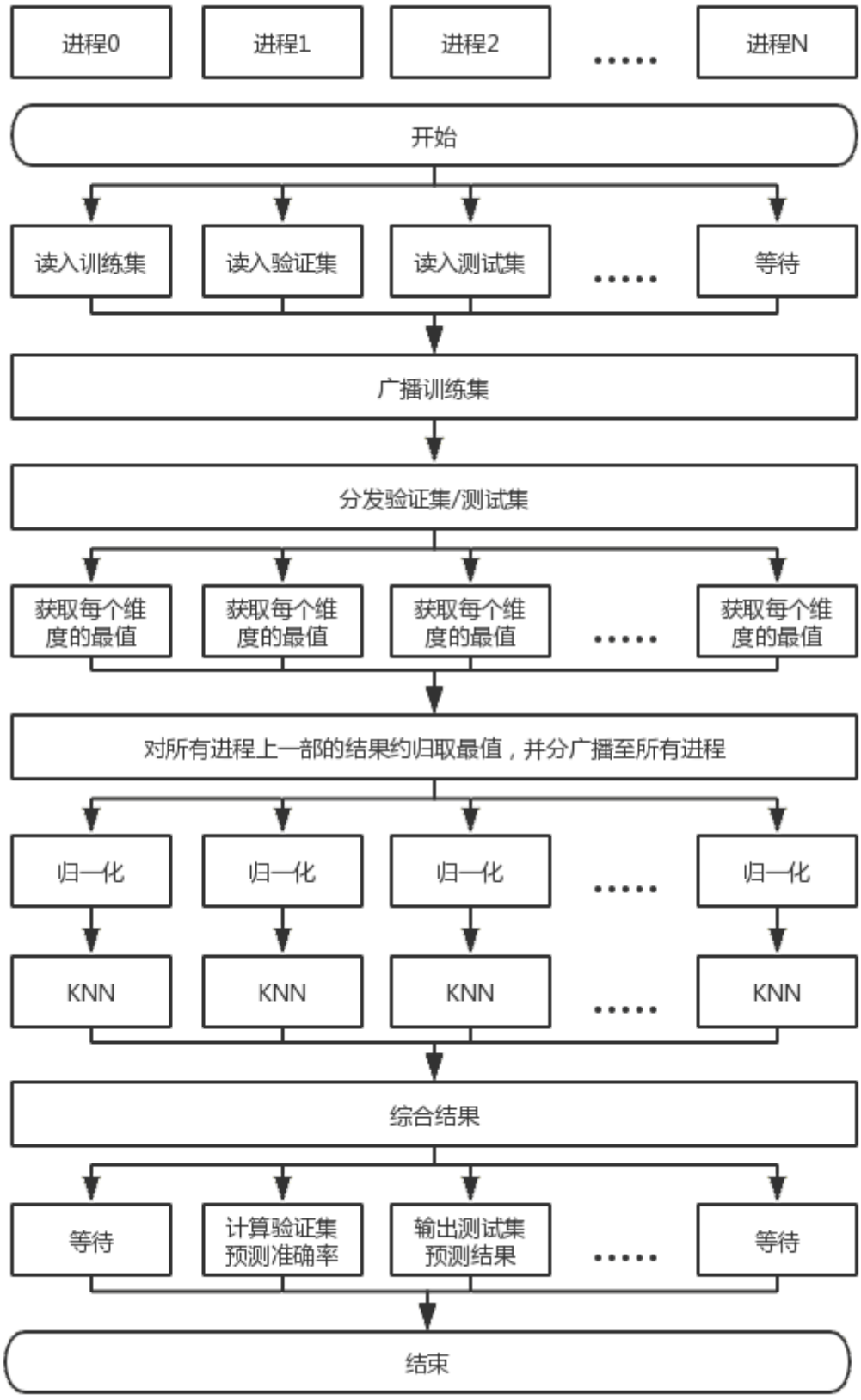
三、基于MPI的并行KNN算法

这部分会介绍本文实现的算法的架构，解释一些关键的函数及变量的意义，最后简单介绍算法的运行方式。

3.1 算法流程

下图展示的是算法的流程图，为了保证并行算法的高校，算法的设计遵循了以下几个原则：

1. 不使用主从式算法结构，最大限度保证每一个进程在算法流程中不会空闲。
2. 减小并行的粒度，尽可能开发更多的并行性，提高并行度。
3. 加大计算时间相对于通信时间的比重，减少通信次数甚至以计算换通信。



接下来的部分，我们把算法分为几个关键步骤来讲解。

3.1.1 数据输入

算法默认有三个输入文件，分别为训练集，测试集以及验证集，其中验证集是否适用是可选的。三个文件分别由三个独立的进程读取，读取完毕后训练集通过MPI_Bcast函数广播至所有的进程，测试集以及验证集则通过MPI_Scatter函数平均分配至所有的进程。

3.1.2 归一化

首先每个进程需要分别获取自己缓冲区内训练集，测试集以及验证集的每一维度数据的最大最小值。在这里为了减少对训练集的重复计算，每个进程只负责一个部分的训练集数据。

接下来，MPI_Allreduce函数会接收上一步中每一个进程计算得到的最值并进行约归操作，计算出全局的最大最小值，并发送回每一个进程。每一个进程根据接收到的值对自己缓冲区内的训练集、测试集以及验证集数据进行归一化操作。

是否需要归一化在算法中也是可选操作，归一化虽然需要额外的时间，但是可以平衡不同维度信息对结果的影响，对最终结果的准确率有着比较大的影响。

3.1.3 KNN

这一步骤中每一个进程读取自己缓冲区内的测试集以及验证集，并与所有的训练集进行比较。计算的方式遵循常规的KNN流程。

3.1.4 合并输出

上一步每一个进程会给出自己缓冲区内测试集与验证集的预测标签，由MPI_Gather函数负责整合至根进程。对于验证集，根进程需要结合真实类标来计算准确率；对于测试集，根进程将预测的标签输出至目标文件。

3.2 函数及变量

这个部分会对代码中的一些重要的函数以及变量进行解释。

3.2.1 全局函数及变量

```
double Euclidean_D(
    int a, //测试集或验证集目标样本点编号
    int b, //训练集目标样本点编号
    int dim, //样本点的维度
    double* Data_train, //完整训练集
    double* Data_test_buffer //储存在该进程缓冲区内的测试集或验证集
);
```

Euclidean_D函数返回的是两个样本点之间的距离，计算使用的的方式是欧式距离。

```
double Manhattan_D(
    int a, //测试集或验证集目标样本点编号
    int b, //训练集目标样本点编号
    int dim, //样本点的维度
    double* Data_train, //完整训练集
    double* Data_test_buffer //储存在该进程缓冲区内的测试集或验证集
);
```

Manhattan_D函数返回的是两个样本点之间的距离，计算使用的的方式是曼哈顿距离。

```
double acc_calc(
    int* real_label, //验证集的真实类标
    int* label, //算法预测出的验证集类标
    int size //验证集大小
);
```

acc_calc是用于计算算法验证集预测结果的准确率的，输入验证集的真实类标以及KNN预测类标，返回值准确率。

```
struct train_data_dis
{
    int label; //该训练集样本的类标
    double dis; //测试集或验证集与该训练集样本的距离
};
```

train_data_dis结构体储存了测试集或验证集的某一个点与训练集中某一个点的距离以及这个点的类标。通过引入这个结构体我们可以更方便地在KNN的过程中选出距离较小的K个样本点以及对应类标。

```
bool Comp(
    train_data_dis a,
    train_data_dis b
);
```

Comp函数自定义了快速排序算法sort的比较函数。通过这种方式我们可以对train_data_dis类的对象根据储存的距离大小进行排序。

3.2.2 一些关键变量

下表展示了一些代码中的关键变量，通过设置这些变量可以按照我们的意愿来运行并行KNN程序。

变量名	数据类型	变量作用
dim	int	样本点的维度
K	int	KNN中的K值
N_train, N_test, N_val	int	训练集/测试集/验证集大小
class_cnt	int	总类标数
Euclidean_distance	boolean	是否适用欧式距离，为false则使用曼哈顿距离
Normalize	boolean	是否归一化，为false则不对训练、验证、测试集数据进行归一化
Validation	boolean	是否使用验证集，为false则不引入验证集
train_file_name	string	训练集的存储路径及文件名
validation_file_name	string	验证集的存储路径及文件名
test_file_name	string	测试集的存储路径及文件名

3.3 算法运行

在安装完成MPI后就可以尝试运行我们提供的并行KNN算法，为了保证成功地运行，请先阅读这部分的内容。

3.3.1 参数设置

在3.2.2小节中我们解释了一些对于算法比较关键的变量，这些变量的默认值是根据下一章中的MNIST数据集设置的，因此读者只需要在运行前根据自己的实际要求对上述参数进行重新设置即可。

3.3.2 注意事项

为了程序可以顺利运行，本文实现的算法对于变量设置以及输入有着以下的要求：

- 输入的数据集需要储存在csv格式的文件中，一个样本点的信息占一行。对于训练集以及验证集，真实类标在每一行的第一位，测试集不需要包括类标信息。
- 数据集的类标需要为整数。
- 如果需要输入验证集，需要保证进程数至少在3以上，否则需要在2以上。
- 测试集、验证集以及测试集的大小都要能整除进程数。
- 测试集的预测类标会存贮在csv格式的文件中，自动生成到算法的运行的目录下。

3.3.3 运行方法（Windows系统为例）

1. 设置好参数，编译算法并准备好输入文件。
2. 打开控制台找到编译后文件的所在目录。
3. 使用指令`mpiexec -n N knn_mpi.exe`运行算法，其中N的值就是具体使用的进程数，需要根据自己的需要设置。

四、实验

4.1 数据集

本次实验所选用的是MNIST手写数字数据集，是机器学习分类任务经常使用的数据集。

4.1.1 下载链接

算法所使用到的MNIST数据集是csv格式的，下载链接为：

https://pjreddie.com/projects/mnist-in-csv/

MNIST数据集的官方下载地址则为：

http://yann.lecun.com/exdb/mnist/index.html

4.1.2 数据集参数

关于使用到的MNIST数据集的一些详细参数见下表。

数据集名称	数据数量	数据维度	数据范围	是否有标签
MNIST_Train	60000	784	0~255	是
MNIST_Test	10000	784	0~255	是

4.2 实验结果

我们设置不同的进程数，在超算上运行实验并统计结果。为了避免偶然情况的发生，每一组实验都重复了五次并对结果取平均值。

4.2.1 算法准确率

我们使用MNIST数据集来验证我们实现的算法的可靠性，关于参数的设置以及算法的准确率统计如下：

K值	距离计算方式	是否归一化	测试集错误率（%）
50	Euclidean (L2)	是	4.61

我们针对同一任务选取其他模型的算法准确率进行比较，整理的结果见下表：

分类器模型	测试集错误率（%）
inear classifier (1-layer NN)	12.0
K-nearest-neighbors, Euclidean (L2)	5.0
SVM, Gaussian Kernel	1.4
2-layer NN, 300 hidden units, mean square error	4.7
Convolutional net LeNet-1	1.7
KNN_MPI（我们的模型）	4.61

以上的数据摘录自MNIST数据集的官方网站：http://yann.lecun.com/exdb/mnist/index.html

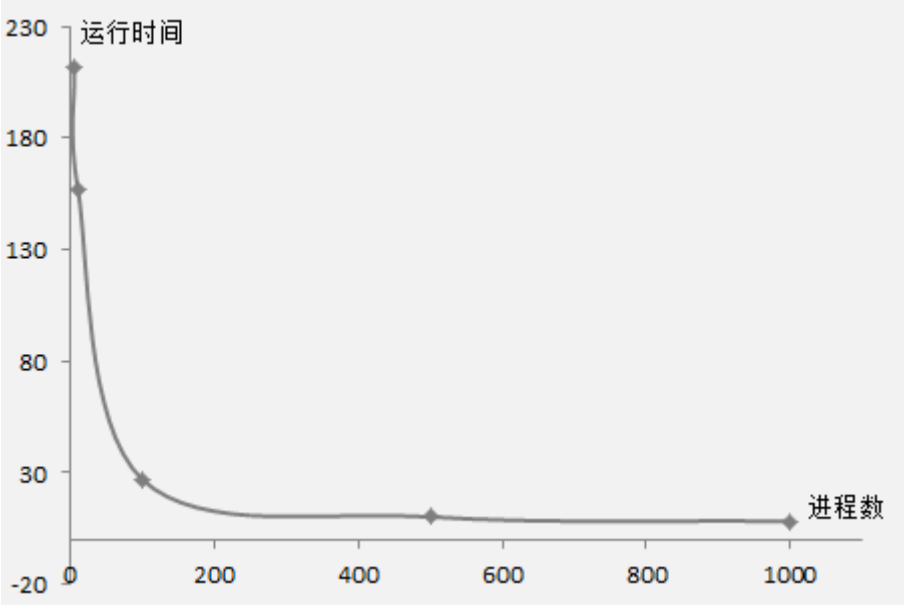
在上表中我们选取了机器学习领域一些经典的分类算法，我们可以观察到我们实现的KNN_MPI算法的测试集错误率达到了作者给出的传统KNN分类器标准。由此可以证明我们的算法是可靠有效的。

4.2.2 算法运行时间

我们设置了六种不同级别的线程数运行我们的模型，并分别对运行时间与准确率进行了统计，结果见下表。

	单线程	5线程	10线程	100线程	500线程	1000线程
运行时间1	3019.42	181.614	178.228	26.752	10.254	8.28103
运行时间2	2901.73	219.197	160.792	25.4775	10.1461	9.23044
运行时间3	2887.52	219.786	143.19	27.49	10.2662	8.0642
运行时间4	2895.35	219.592	143.872	28.7387	10.3346	7.90485
运行时间5	2901.53	219.166	160.284	26.3715	10.4135	7.86987
运行时间（平均）	2921.11	211.871	157.2732	26.96594	10.28288	8.270078

根据5线程至1000线程的结果绘图，得到的曲线如下图：



分析得到的图像，我们不难观察到以下两个特征：

1. 当进程数较小时，继续扩增运行进程并行对于算法的加速是超线性的。
2. 当进程数较大时，继续扩增运行进程对并行算法的加速效果变得不明显。

针对以上的现象我们可以分别给出以下的解释：

1. 由于在常规的KNN中我们需要将目标样本与训练集所有样本之间的距离进行排序，而这里的使用到的排序算法是时间复杂度为$O(n\log_2 n)$。当进程数增加时，每一个进程需要处理的数据样本减小，最终导致排序算法的时间复杂度以线性对数阶减小。
2. 而当进程数足够大时，并行算法进行KNN运算所消耗的时间已经降到足够小，相对地算法内进程通信，文件读写操作等运算的时间就显得比较长。并且这一类操作本身的特性使他们占用的时间并不会与进程数有着直接的关系，因此最后继续扩增运行进程对并行算法的加速效果变得不明显。

五、参考资料

- 机器学习（二）：k近邻法（kNN）
  https://blog.csdn.net/eeeee123456/article/details/79927128
- 都志辉. 高性能计算并行编程技术：MPI并行程序设计[M]. 清华大学出版社, 2001.
- MPI编程简介
  https://blog.csdn.net/qinggebuyao/article/details/8059300
- MPI_Bcast函数的用法
  https://blog.csdn.net/wyg1065395142/article/details/53647698
- MPI之聚合通信-Scatter，Gather，Allgather
  https://blog.csdn.net/sinat_22336563/article/details/70229243
- THE MNIST DATABASEof handwritten digits
  http://yann.lecun.com/exdb/mnist/index.html