并行程序设计实践大作业

专业和班级：计算机 18-5

姓名：刘澳

学号：201801011312

时间：2020 年 10 月 25 日

目录

一、并行程序的发展和当前应用概述

随着计算机和计算方法的飞速发展，几乎所有的学科都走向定量化和精确化，从而产生了一系列诸如计算物理、计算化学、计算生物学、计算地质学、计算气象学和 计算材料科学等的计算科学，在世界上逐渐形成了一门计算性的学科分支，即计算科学与工程，简称为CSE(Computational Science & Engineering)。当今，计算科学已经和传统的两种科学即理论科学和实验科学，并列成为第三门科学，它们彼此相辅相成地推动科学发展与社会进步。 在很多情况下，或者是理论模型复杂甚至理论尚未建立，或者实验费用昂贵甚至实验无法进行，此时计算就成为求解问题的唯一或主要手段。计算极大地增强了人们 从事科学研究的能力，大大地加速了把科技转化为生产力的过程，深刻地改变着人类认识世界和改造世界的方法和途径。计算科学的理论和方法，作为新的研究手段 和新的设计与制造技术的理论基础，正推动着当代科学与技术向纵深发展。

计算科学所涉及的计算方法，即我们平常所说的算法，是求解问题的方法和步骤，而并行算法则是指用多台计算机联合求解问题的方法和步骤。现在，并行算法之所 以受到极大的重视，是为了提高计算速度（单机受物理速度限制无法满足）、提高计算精度（加密、计算网格等）以及满足实时计算需要（数值天气预报等）。而并 行计算，简单地说，就是在并行计算机上所作的计算，它和常说的高性能计算，超级计算是同义词，因为在任何高性能计算和超级计算总离不开使用并行技术。

所谓并行计算，是指同时使用多种计算资源解决计算问题的过程。为执行并行计算，计算资源应包括一台配有多处理机（并行处理）的计算机、一个与网络相连的计 算机专有编号，或者两者结合使用。并行计算的主要目的是快速解决大型且复杂的计算问题。此外还包括：利用非本地资源，节约成本 — 使用多个“廉价”计算资源取代大型计算机，同时克服单个计算机上存在的存储器限制。

传统地，串行计算是指在单个计算机（具有单个中央处理单元）上执行软件写操作。CPU 逐个使用一系列指令解决问题，但其中只有一种指令可提供随时并及时的使用。并行计算是在串行计算的基础上演变而来，它努力仿真自然世界中的事务状态：一个序列中众多同时发生的、复杂且相关的事件。

为利用并行计算，通常计算问题表现为以下特征：

（1）将工作分离成离散部分，有助于同时解决；
（2）随时并及时地执行多个程序指令；
（3）多计算资源下解决问题的耗时要少于单个计算资源下的耗时。

并行计算是相对于串行计算来说的，所谓并行计算分为时间上的并行和空间上的并行。时间上的并行就是指流水线技术，而空间上的并行则是指用多个处理器并发的执行计算。

并行计算科学中主要研究的是空间上的并行问题。空间上的并行导致了两类并行机的产生，按照 Flynn 的说法分为：单指令流多数据流（SIMD）和多指令流多数据流（MIMD）。我们常用的串行机也叫做单指令流单数据流（SISD）。 MIMD 类的机器又可分为以下常见的五类： 并行向量处理机(PVP) 、对称多处理机(SMP) 、大规模并行处理机(MPP) 、工作站机群(COW) 、分布式共享存储处理机(DSM)。

一、分布式计算

所谓分布式计算是一门计算机科学，它研究如何把一个需要非常巨大的计算能力才能解决的问题分成许多小的部分，然后把这些部分分配给许多计算机进行处理，最 后把这些计算结果综合起来得到最终的结果。最近的分布式计算项目已经被用于使用世界各地成千上万位志愿者的计算机的闲置计算能力，通过因特网，您可以分析 来自外太空的电讯号，寻找隐蔽的黑洞，并探索可能存在的外星智慧生命；您可以寻找超过 1000 万位数字的梅森质数；您也可以寻找并发现对抗艾滋病病毒的更 为有效的药物。这些项目都很庞大，需要惊人的计算量，仅仅由单个的电脑或是个人在一个能让人接受的时间内计算完成是决不可能的。

分布式计算是利用互联网上的计算机的中央处理器的闲置处理能力来解决大型计算问题的一种计算科学。下面，我们看看它是怎么工作的：

首先, 要发现一个需要非常巨大的计算能力才能解决的问题。这类问题一般是跨学科的、极富挑战性的、人类急待解决的科研课题。其中较为著名的是:

1. 解决较为复杂的数学问题，例如：GIMPS（寻找最大的梅森素数）。
2. 研究寻找最为安全的密码系统，例如：RC-72（密码破解）。
3. 生物病理研究，例如：Folding@home（研究蛋白质折叠，误解，聚合及由此引起的相关疾病）。

4. 各种各样疾病的药物研究，例如：United Devices（寻找对抗癌症的有效的药物）。
5. 信号处理，例如：SETI@Home（在家寻找地外文明）。

从这些实际的例子可以看出，这些项目都很庞大，需要惊人的计算量，仅仅由单个的电脑或是个人在一个能让人接受的时间内计算完成是决不可能的。在以前，这些问题都应该由超级计算机来解决。但是，超级计算机的造价和维护非常的昂贵，这不是一个普通的科研组织所能承受的。随着科学的发展，一种廉价的、高效的、维护方便的计算方法应运而生——分布式计算！

随着计算机的普及，个人电脑开始进入千家万户。与之伴随产生的是电脑的利用问题。越来越多的电脑处于闲置状态，即使在开机状态下中央处理器的潜力也远远不 能被完全利用。我们可以想象，一台家用的计算机将大多数的时间花费在“等待”上面。即便是使用者实际使用他们的计算机时,处理器依然是寂静的消费，依然是 不计其数的等待（等待输入，但实际上并没有做什么）。互联网的出现，使得连接调用所有这些拥有限制计算资源的计算机系统成为了现实。

当然，这看起来也似乎很原始、很困难，但是随着参与者和参与计算的计算机的数量的不断增加，计算计划变得非常迅速，而且被实践证明是的确可行的。目前一些较大的分布式计算项目的处理能力已经可以达到甚而超过目前世界上速度最快的巨型计算机。

分布式计算是近年提出的一种新的计算方式。所谓分布式计算就是在两个或多个软件互相共享信息，这些软件既可以在同一台计算机上运行，也可以在通过网络连接起来的多台计算机上运行。分布式计算比起其它算法具有以下几个优点：

1、稀有资源可以共享，
2、通过分布式计算可以在多台计算机上平衡计算负载，
3、可以把程序放在最适合运行它的计算机上，

其中，共享稀有资源和平衡负载是计算机分布式计算的核心思想之一。

实际上，网格计算就是分布式计算的一种。如果我们说某项工作是分布式的，那么，参与这项工作的一定不只是一台计算机，而是一个计算机网络，显然这种“蚂蚁搬山”的方式将具有很强的数据处理能力。网格计算的实质就是组合与共享资源并确保系统安全。

二、基于 GPU 的计算

GPU 英文全称 Graphic Processing Unit，中文翻译为“图形处理器”。GPU 是相对于 CPU 的一个概念，由于在现代的计算机中（特别是家用系统，游戏的发烧友）图形的处理变得越来越重要，需要一个专门的图形的核心处理器。

GPU 所有计算均使用浮点算法，而且目前还没有位或整数运算指令。此外，由于 GPU 专为图像处理设计，因此存储系统实际上是一个二维的分段存储空间，包括 一个区段号（从中读取图像）和二维地址（图像中的 X、Y 坐标）。此外，没有任何间接写指令。输出写地址由光栅处理器确定，而且不能由程序改变。这对于自然 分布在存储器之中的算法而言是极大的挑战。最后一点，不同碎片的处理过程间不允许通信。实际上，碎片处理器是一个 SIMD 数据并行执行单元，在所有碎片中 独立执行代码。

简单说 GPU 就是能够从硬件上支持 T&L(Transform and Lighting，多边形转换与光源处理）的显示芯片，因为 T&L 是 3D 渲染中的一个重要部分，其作用是计算多边形的 3D 位置和处理动态光线效 果，也可以称为“几何处理”。一个好的 T&L 单元，可以提供细致的 3D 物体和高级的光线特效；只大多数 PC 中，T&L 的大部分运算是交由 CPU 处理的(这就也就是所谓的软件 T&L)，由于 CPU 的任务繁多，除了 T&L 之外，还要做内存管理、输入响应等非 3D 图形处理工作， 因此在实际运算的时候性能会大打折扣，常常出现显卡等待 CPU 数据的情况，其运算速度远跟不上今天复杂三维游戏的要求。即使 CPU 的工作频率超过 1GHz 或更高，对它的帮助也不大，由于这是 PC 本身设计造成的问题，与 CPU 的速度无太大关系。

GPU 擅长执行矢量和矩阵运算，其三维变换计算都是矩阵运算。而且其片元处理中涉及大量的数学运算。GPU 天生具有并行性，往往是对屏幕上所有点进行相同的运算。这也是采用 GPU 作图形处理比 CPU 效率高得多的一个主要原因。

三、多核时代

多内核是指在一枚处理器中集成两个或多个完整的计算引擎(内核)。多核技术的开发源于工程师们认识到，仅仅提高单核芯片的速度会产生过多热量且无法带来相应 的性能改善，先前的处理器产品就是如此。他们认识到，在先前产品中以那种速率，处理器产生的热量很快会超过太阳表面。即便是没有热量问题，其性价比也令人 难以接受，速度稍快的处理器价格要高很多。英特尔工程师们开发了多核芯片，使之满足“横向扩展”（而非“纵向扩充”）方法，从而提高性能。该架构实现了 “分治法”战略。

通过划分任务，线程应用能够充分利用多个执行内核，并可在特定的时间内执行更多任务。多核处理器是单枚芯片（也称为“硅核”），能够直接插入单一的处理器 插槽中，但操作系统会利用所有相关的资源，将它的每个执行内核作为分立的逻辑处理器。通过在两个执行内核之间划分任务，多核处理器可在特定的时钟周期内执行更多任务。

多核架构能够使目前的软件更出色地运行，并创建一个促进未来的软件编写更趋完善的架构。尽管认真的软件厂商还在探索全新的软件并发处理模式，但是，随着向多核处理器的移植，现有软件无需被修改就可支持多核平台。

操作系统专为充分利用多个处理器而设计，且无需修改就可运行。为了充分利用多核技术，应用开发人员需要在程序设计中融入更多思路，但设计流程与目前对称多处理 (SMP) 系统的设计流程相同，并且现有的单线程应用也将继续运行。

现在，得益于线程技术的应用在多核处理器上运行时将显示出卓越的性能可扩充性。此类软件包括多媒体应用（内容创建、编辑，以及本地和数据流回放）、工程和其他技术计算应用以及诸如应用服务器和数据库等中间层与后层服务器应用。

多核技术能够使服务器并行处理任务，而在以前，这可能需要使用多个处理器，多核系统更易于扩充，并且能够在更纤巧的外形中融入更强大的处理性能，这种外形所用的功耗更低、计算功耗产生的热量更少。

现在面临的多核带给软件业的挑战：（1）对于传统软件而言，传统的软件开发多为串行程序开发，随着 CPU 频率的提升自然能得到性能提升，而多核 CPU 现行的频率并不高，在多核 CPU 上，单独一个核心的性能并不会比高频 CPU 高。软件如何充分利用多核 CPU 的资源成为需要解决问题。（2）现行的程序设计语言，包括它们的 Library 多为串行设计，如 C++及它的 STL，Boost；Java Class Library，.NET Framework。

四、量子计算

量子计算的概念最早由 IBM 的科学家 R. Landauer 及 C. Bennett 于 70 年代提出。他们主要探讨的是计算过程中诸如自由能、信息与可逆性之间的关系。80 年代初期，阿岗国家实验室的 P. Benioff 首先提出二能阶的量子系统可以用来仿真数字计算；稍后费因曼也对这个问题产生兴趣而着手研究，并在 1981 年于麻省理工学院举行的 First Conference on Physics of Computation 中给了一场演讲，勾勒出以量子现象实现计算的愿景。1985 年，牛津大学的 D. Deutsch 提出量子图林机的概念，量子计算才开始具备了数学的基本型式。然而上述的量子计算研究多半局限于探讨计算的物理本质，还停留在相当抽象的层次，尚未进一步跨入发展算法的阶段。

1994 年，贝尔实验室的应用数学家 P. Shor 指出，相对于传统电子计算器，利用量子计算可以在更短的时间内将一个很大的整数分解成质因子的乘积。这个结论开启量子计算的一个新阶段：有别于传 统计算法则的量子算法(quantum algorithm)确实有其实用性，绝非科学家口袋中的戏法。自此之后，新的量子算法陆续的被提出来，而物理学家接下来所面临的重要的课题之一，就是如 何去建造一部真正的量子计算器，来执行这些量子算法。许多量子系统都曾被点名做为量子计算器的基础架构，例如光子的偏振、空腔量子电动力学、离子阱以及核 磁共振等等。以目前的技术来看，这其中以离子阱与核磁共振最具可行性。事实上，核磁共振已经在这场竞赛中先驰得点：以 I. Chuang 为首的 IBM 研究团队在 2002 年的春天，成功地在一个人工合成的分子中(内含 7 个量子位)利用 NMR 完成 N =15 的因子分解。

矩阵大小	10	100	200	500	1000	1500
并行计算	0.00	0.65	0.84	5.88	48.80	163.47
串行计算	0.00	10.00	80.00	740.00	5080.00	19780.00

二、GPU 编程实践的作业

题目 1：编写一个矩阵乘法的 GPU 并行程序，并且与对应规模的串行程序进行运行时间的比对（n=10，100，200，500，1000，1500，2000……），

矩阵 A（n，n）

矩阵 B（n，n）

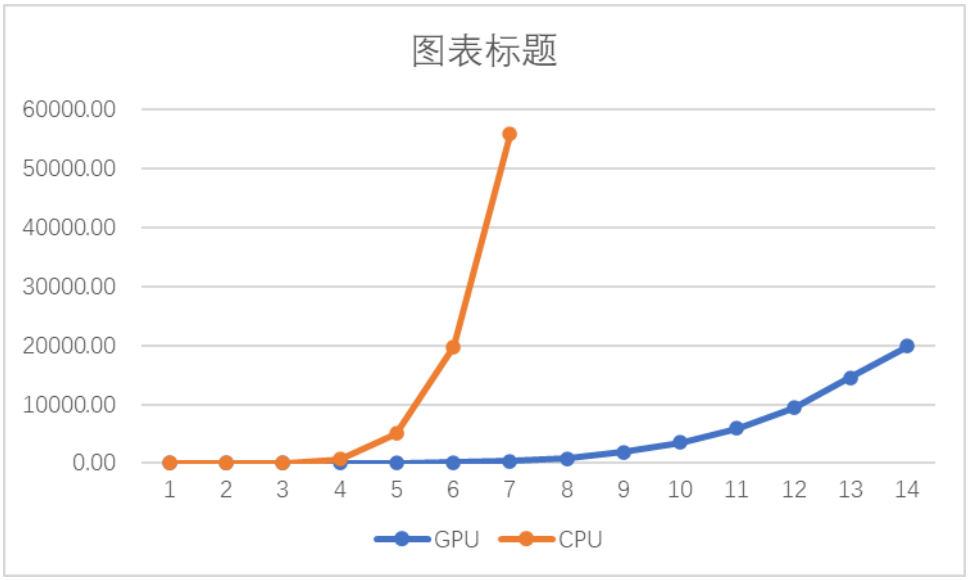
C = A x B

要求：

1、完成程序的开发并验证其正确性，完成一个实验报告（程序源代码、变量和语句的详细说明；

2、在实验报告中通过图表说明 CPU 串行和 GPU 并行在各种规模的运行时间；

3、在实验报告中通过图表说明 GPU 并行不同的数据分配在各种规模的运行时间）。



```
#include "cuda_runtime.h"
#include <iostream>
#include <random>
#include <math.h>
#include <algorithm>
#include "device_launch_parameters.h"
#include <stdio.h>
#include <cuda.h>
#include <stdlib.h>
#include <malloc.h>
#include <time.h>


#define ROWLEN 1024*5
#define COLLEN 1024*5
#define MAX 10

//N = arraySize
//#define n 512
//#define  multiple 3
//#define N 1024

//S = threadNum
#define S 32

#define RL ROWLEN / S
#define CL COLLEN / S


clock_t start, stop; //clock_t 为clock()函数返回的变量类型
double duration;

using namespace std;

//Use the GPU to calculate the KNN's answer
__global__ void getDistanceGPU(double trainSet[COLLEN][ROWLEN], double*
 testData, double* dis)
{

    int xid = threadIdx.x + blockDim.x * blockIdx.x;
    int yid = threadIdx.y + blockDim.y * blockIdx.y;

    int row = yid;
    int col = xid;
```

```
    if (col < ROWLEN && row < COLLEN)
    {
        double temp = 0;
        if (col == 0)
        {
            for (int i = 0; i < ROWLEN; i++)
            {
                temp += (trainSet[row][i] - testData[i]) * (trainSet[ro
w][i] - testData[i]);
            }
            dis[row] = sqrt(temp);
            //printf("%f \n", dis[row]);
        }
    }
}

void gpuCal(double a[ROWLEN][COLLEN], double b[ROWLEN], double c[COLLEN
])
{
    double (*dev_a)[ROWLEN];
    double* dev_b;
    double* dev_c;

    //在GPU中开辟空间
    cudaMalloc((void**)&dev_a, ROWLEN * COLLEN * sizeof(double));
    cudaMalloc((void**)&dev_b, ROWLEN * sizeof(double));
    cudaMalloc((void**)&dev_c, COLLEN * sizeof(double));

    //将CPU内容复制到GPU
    cudaMemcpy(dev_a, a, ROWLEN * COLLEN * sizeof(double), cudaMemcpyHo
stToDevice);
    cudaMemcpy(dev_b, b, ROWLEN * sizeof(double), cudaMemcpyHostToDevic
e);

    //声明时间Event
    float time = 0;
    cudaEvent_t start, stop;
    cudaEventCreate(&start);
    cudaEventCreate(&stop);
    cudaEventRecord(start, 0);

    //GPU开始计算
    dim3 threadsPerBlock(S, S);
    dim3 blocksPerGrid(RL, CL);
```

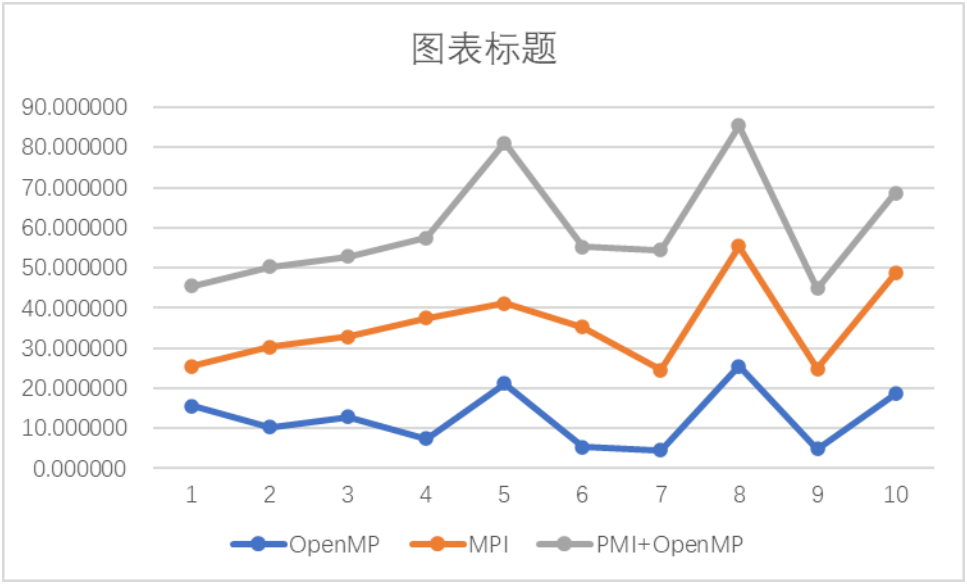
```
    getDistanceGPU << <blocksPerGrid, threadsPerBlock >> > (dev_a, dev_
b, dev_c);
    //结束计时
    cudaEventRecord(stop, 0);

    cudaEventSynchronize(start);
    cudaEventSynchronize(stop);
    //计算时间差
    cudaEventElapsedTime(&time, start, stop);
    //将内容拷贝回CPU
    cudaMemcpy(c, dev_c, COLLEN * sizeof(double), cudaMemcpyDeviceToHos
t);

   /* for (int j = 0; j < COLLEN; j++)
    {
        printf("%f ", c[j]);
    }
    printf("\n");*/
    printf("GPU: spendTime: %fms\n\n\n", time);
    cudaFree(dev_a);
    cudaFree(dev_b);
    cudaFree(dev_c);
}

//Calculate the distance between testData and dataSet[i]
double getDistance(double* d1, int* d2);

//calculate all the distance between testData and each training data
void getAllDistance(double trainSet[ROWLEN][COLLEN], double* testData,
double* discard_block);

// Randomly generated training set
void randNum(double trainSet[ROWLEN][COLLEN], int rlen, int clen);

//Randomly generated testDate
void randNum(double* testData, int clen);

//Print the trainSet
void print(double trainSet[ROWLEN][COLLEN], int rlen, int clen);

//Print the testSet
void print(double* testData, int clen);

int main(int argc, char const* argv[])
```

```
{
    double (*trainSet)[ROWLEN];
    double* testData;
    double* dis;
    trainSet = new double[ROWLEN][COLLEN];
    testData = new double[COLLEN];
    dis = new double[ROWLEN];
    randNum(trainSet, ROWLEN, COLLEN);
    randNum(testData, COLLEN);

    gpuCal(trainSet, testData, dis);
    getAllDistance(trainSet, testData, dis);

    cout << "-----------------trainSet----------------------------" <<
endl;
    //print(trainSet, ROWLEN, COLLEN);
    cout << "-----------------testSet----------------------------" << e
ndl;
    //print(testData, COLLEN);
    cout << "-----------------dis--------------------------------" << en
dl;
    //print(dis, COLLEN);
    sort(dis, dis + COLLEN);
    //print(dis, COLLEN);
    return 0;
}


//Calculate the distance between trainSet and testData
double getDistance(double* d1, double* d2)
{
    double dis = 0;
    for (int i = 0; i < COLLEN; i++)
    {
        dis += pow((d1[i] - d2[i]), 2);
    }
    return sqrt(dis);
}

//calculate all the distance between testData and each training data
void getAllDistance(double trainSet[ROWLEN][COLLEN], double* testData,
double* dis)
{
    start = clock();
    //*****************************
```

```
    //*这里写你所要测试运行时间的程序 *
    for (int i = 0; i < ROWLEN; i++)
    {
        dis[i] = getDistance(trainSet[i], testData);
    }
    //******************************
    stop = clock();
    duration = (double)(stop - start) / CLOCKS_PER_SEC; //CLK_TCK 为
clock()函数的时间单位，即时钟打点
    printf("CPU: spendTime: %fms\n\n\n", duration * 1000);

}

// Randomly generated training set
void randNum(double trainSet[ROWLEN][COLLEN], int rlen, int clen)
{
    for (int i = 0; i < rlen; i++)
    {
        for (int j = 0; j < clen; j++)
        {
            trainSet[i][j] = rand() % MAX;
        }
    }
}

//Randomly generated testDatd
void randNum(double* testData, int clen)
{
    for (int i = 0; i < clen; i++)
    {
        testData[i] = rand() % MAX;
    }
}
//Print the trainSet
void print(double trainSet[ROWLEN][COLLEN], int rlen, int clen)
{
    for (int i = 0; i < rlen; i++)
    {
        for (int j = 0; j < clen; j++)
        {
            cout << trainSet[i][j] << " ";
        }
        cout << endl;
    }
```

```
}

//Print the testSet
void print(double* testData, int clen)
{
    for (int i = 0; i < clen; i++)
    {
        cout << testData[i] << " ";
    }
    cout << endl;
}
```

三、MPICH 编程实践的作业



一、蒙特卡罗 MPI 求 PI

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include "mpi.h"
void main(int argc, char* argv[])
{
    //用来记录CPU核心数
    int numprocs;
    //当前的线程编号
```

```
    int myid;

    //消息状态。接收函数返回时，将在这个参数指示的变量中存放实际接收消息的状态信息，包括消息的源进程标识，消息标签，包含的数据项个数等。
    MPI_Status status;
    /*MPI 函数*/
    //MPI 初始化
    MPI_Init(&argc, &argv);

    //获得当前改 communicator 正在使用的进程号
    MPI_Comm_rank(MPI_COMM_WORLD, &myid);
    //获得改 communicator 下的总进程数
    MPI_Comm_size(MPI_COMM_WORLD, &numprocs);

    long total = 10;
    long local_total = 0;
    long count;
    long local_count;
    double x, y;

    // 非0 号进程发送消息
    if (myid == 0)
    {
        total = 999999999;
        local_total = total / numprocs;
    }
    // 把 local_total 广播给所有进程
    MPI_Bcast(&local_total, 1, MPI_LONG, 0, MPI_COMM_WORLD);

    for (long i = 0; i < local_total; i++)
    {
        x = (double)rand() / RAND_MAX;
        y = (double)rand() / RAND_MAX;
        if (x * x + y * y <= 1)
        {
            local_count++;
        }
    }

    MPI_Reduce(&local_count, &count, 1, MPI_LONG, MPI_SUM, 0, MPI_COMM_WORLD);

    double PI = 4 * (double)count / total;
```

```
    if (myid == 0)
    {
        printf("%f \n", PI);
    }

    MPI_Finalize();
} /* end main */

//mpicc MonteCarlo_MPI.c -o MonteCarlo_MPI
//mpirun -np 28 ./MonteCarlo_MPI
```

二、蒙特卡罗 OpenMP 求 PI

```
// OpenMP.cpp : 使用蒙特卡洛方法求PI
//可以优化的点：
//1. 随机生成数的时候的for
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
#include <stdlib.h>
#include <math.h>

//SIDE = 正方形边长
#define SIDE 10
//POINTS 生成的总点数
#define POINTS 999999999


//随机生成一个数组，该数组中的每一对数代表一个点
double randPoint(long points);

//

int main()
{
    double PI = randPoint(POINTS);

    return 0;

}

double randPoint(long points)
{

    long cycleNum = 0;
```

```
    long squareNum = points;
    double x = 0;
    double y = 0;

    #pragma omp parallel  for private(x, y) reduction(+:cycleNum)
    for (int i = 0; i < points; i++)
    {
        //生成随机的坐标值
        x = (double)rand() / (double)RAND_MAX;
        y = (double)rand() / (double)RAND_MAX;

        //计算是否落到了圆中
        if (sqrt(x*x + y*y) <= 1)
        {
            cycleNum++;
        }
    }

    //计算PI
    double PI =  (double)cycleNum / (double)squareNum;

    printf("PI = %f\n",4*PI);

    return PI;

}
//gcc ./MonteCarlo_OpenMP.c -o MonteCarlo_OpenMP -fopenmp -std=c11 -lm
// ./MonteCarlo_OpenMP
```

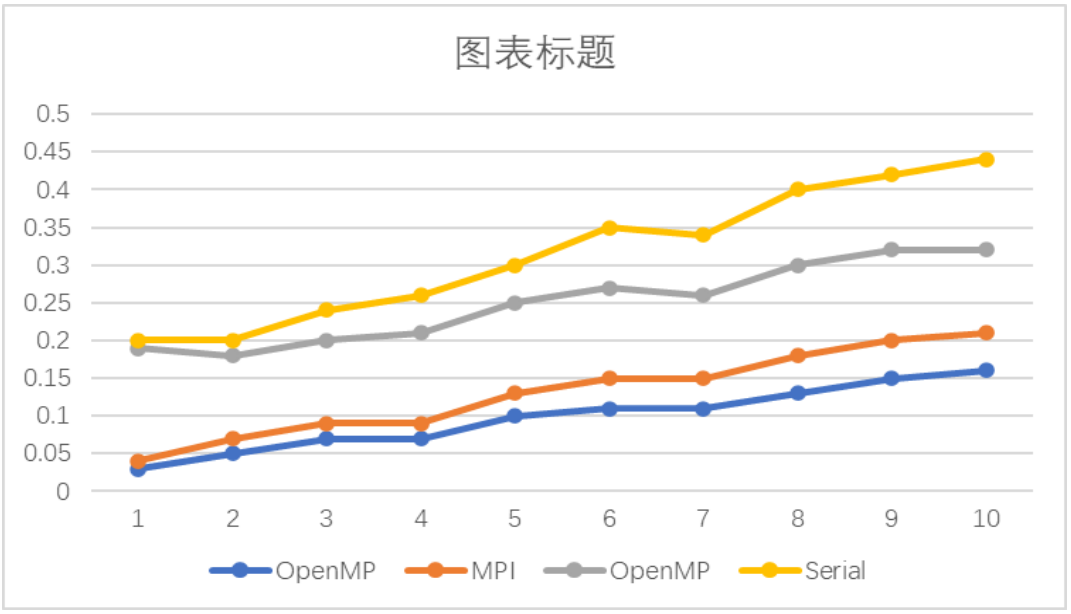
三、蒙特卡罗 OpenMP+MPI 求 PI

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include "mpi.h"
void main(int argc, char* argv[])
{
    //用来记录CPU 核心数
    int numprocs;
    //当前的线程编号
    int myid;

    //消息状态。接收函数返回时，将在这个参数指示的变量中存放实际接收消息的状态信息，包括消息的源进程标识，消息标签，包含的数据项个数等。
```

```
    MPI_Status status;
    /*MPI 函数*/
    //MPI 初始化
    MPI_Init(&argc, &argv);

    //获得当前改 communicator 正在使用的进程号
    MPI_Comm_rank(MPI_COMM_WORLD, &myid);
    //获得改 communicator 下的总进程数
    MPI_Comm_size(MPI_COMM_WORLD, &numprocs);

    long total = 10;
    long local_total = 0;
    long count;
    long local_count;
    double x, y;

    // 非 0 号进程发送消息
    if (myid == 0)
    {
        total = 999999999;
        local_total = total / numprocs;
    }
    // 把 local_total 广播给所有进程
    MPI_Bcast(&local_total, 1, MPI_LONG, 0, MPI_COMM_WORLD);
    #pragma omp parallel for private(x, y) reduction(+:local_count)
    for (long i = 0; i < local_total; i++)
    {
        x = (double)rand() / RAND_MAX;
        y = (double)rand() / RAND_MAX;
        if (x * x + y * y <= 1)
        {
            local_count++;
        }
    }

    MPI_Reduce(&local_count, &count, 1, MPI_LONG, MPI_SUM, 0, MPI_COMM_
WORLD);

    double PI = 4 * (double)count / total;

    if (myid == 0)
    {
        printf("%f \n", PI);
    }
```

```
    MPI_Finalize();
} /* end main */

//mpicc MonteCarlo_MPI+OpenMP.c -o MonteCarlo_MPI+OpenMP
//mpirun -np 28 ./MonteCarlo_MPI+OpenMP
```

四、积分 MPI 求 PI




```
#include<stdio.h>
#include<math.h>
#include<time.h>
#include "mpi.h"

clock_t start, stop; //clock_t 为clock()函数返回的变量类型
double duration;

// 利用积分函数，求解 PI 的值
double func(double xi){
    return (4.0 / (1.0 + xi * xi));
}
int main(int argc,char* argv[]){
    int rankId;  // 当前进程的 id
    int procsNum;  // 进程的数量
    double pi;  // PI 的结果
    double h;  // 积分区域中两个 x 点的间距
    double xi;  // x 变量
    double res;  // 各个进程获得的结果
    double startTime;  // 程序开始的时间
```

```
    double endTime;  // 程序结束的时间
    int count = 0;
    MPI_Init(&argc, &argv);  // MPI 初始化
    MPI_Comm_rank(MPI_COMM_WORLD, &rankId);  // 返回MPI 线程的ID
    MPI_Comm_size(MPI_COMM_WORLD, &procsNum);  // 返回MPI 线程的数
量
    long long start = 1000000;
    long long end = 10000000;
    int steps = 1000000;
    for(long long n = start; n <= end; n += steps){
        // 广播，将root 进程数据广播至所有进程
        MPI_Bcast(&n, // 通信消息中缓冲区的起始地址
                1, // 消息缓冲区中的数据个数
                MPI_LONG_LONG, // 数据类型为MPI_LONG_LONG
                0, // 发送广播的root 序列号
                MPI_COMM_WORLD);  // 通信子句柄
        //记录开始计算的时间
        if(rankId==0){
            //startTime = MPI_Wtime();
             startTime = clock();
        }
        h = 1.0 / (double)n;//计算间距
        res = 0.0;//当前进程计算的结果
        //每个进程负责计算自己那一部分
        //比如有一万个点分配给十个进程每个进程需要完成一千个点
        //每个进程从自己的进程id 开始进行累加
        for(int i = rankId; i < n; i += procsNum){
            xi = h * ((double)i + 0.5);
            res += func(xi);
        }
        res = h * res;
        //归约计算将每个进程获得的结果接收并相加到根结点
        MPI_Reduce(&res, // 发送消息的内存块的指针
                &pi, // 接受消息的内存块的指针
                1, // 数据量
                MPI_DOUBLE, // 数据类型为MPI_DOUBLE
                MPI_SUM, // 规约操作为求和
                0, // 接受消息的进程号
                MPI_COMM_WORLD);  // 通信子句柄
        // 记录结束运行的时间并输出
        if(rankId == 0){
            //endTime = MPI_Wtime();
            endTime = clock();
```

```
            duration = (double)(start - endTime) / CLOCKS_PER_SEC
; //CLK_TCK 为clock()函数的时间单位，即时钟打点
            printf("%ld Points calculate the PI spend %fms\n\n",
n, duration * 1000);
        }
    }
    MPI_Finalize();  // 进程销毁
    return 0;
}
//mpicc mpi_pi_qr.c -o
//mpirun -np 28 ./mpi_pi_qr >> MPI.txt
```

五、积分 OpenMP 求 PI

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <math.h>
#include <omp.h>

clock_t start, stop; //clock_t 为clock()函数返回的变量类型
double duration;


int main(){
    int nt = 10;//线程数
    long long start = 1000000;
    long long end = 10000000;
    int steps = 1000000;
    int times = 0;
    clock_t startTime;  // 程序开始的时间
    clock_t endTime;  // 程序结束的时间
    for(long long numSteps = start; numSteps <= end; numSteps +=
steps){
        double sum = 0.0;  // 记录积分运算的结果
        double temp = 0.0; //积分运算中间变量
        double pi = 0.0;  // 记录PI 值
        double step = 1.0 / (double) numSteps;  // 求和运算的步长
        startTime = clock();
        #pragma omp parallel num_threads(nt)
        {
            #pragma omp for reduction(+:sum)

                for (int i = 0; i < numSteps; i++)
                {
```

```
                    temp = (i + 0.5) * step;
                    sum += 4.0 / (1.0 + temp * temp);
                }

        }

        pi = step * sum;
        endTime = clock();
        duration = (double)(start - endTime) / CLOCKS_PER_SEC; //
CLK_TCK 为clock()函数的时间单位，即时钟打点
        printf("%ld Points calculate the PI spend %fms\n\n", numS
teps, duration * 1000);
    }
    return 0;
}
//gcc openmp_pi_qr.c -o openmp_pi_qr -fopenmp -std=c11 -lm
```

六、积分 MPI+OpenMP 求 PI

```
#include "mpi.h"
#include "omp.h"
#include <math.h>
#include <stdlib.h>
#include<stdio.h>
#include <time.h>
int main( int argc, char* argv[] ){
    int rankId; // 线程的ID 号 -> mpi
    int nproc;  // 总的线程数量 -> mpi
    double startTime, endTime;  // 记录运算时间
    double local = 0.0;  // 记录求和数据 -> openmp
    double pi = 0.0;  // PI 的计算值 -> mpi
    double step = 0.0;  // 计算的步长 -> mpi
    double xi = 0.0;  // x 坐标的值 -> openmp
    MPI_Init( &argc, &argv );  // MPI 线程初始化
    MPI_Comm_size( MPI_COMM_WORLD, &nproc );  // 返回线程的总数
    MPI_Comm_rank( MPI_COMM_WORLD, &rankId );  // 返回线程的ID
    long long start = 100000;
    long long end = 1000000;
    int steps = 100000;
    int times = 0;
    for(long long n = start; n <= end; n += steps){
        int low; // 线程的最低ID -> openmp
        int up;  // 线程的最高ID -> openmp
        local = 0.0;
```

```
        pi = 0.0;
        step = 0.0;
        xi = 0.0;
        step = 1.0 / (double)n;  // 累加计算的步长
        low = rankId * (n / nproc);  // 最低的线程ID
        up = low + n / nproc - 1;  // 最高的线程ID
        if(rankId == 0){
            //startTime = MPI_Wtime();  // 记录并行计算开始时间
            startTime = clock();
        }
        #pragma omp parallel
        {
            #pragma omp for reduction(+:local)
            for (int i = low; i < up; i++){
                // xi 累加，local 进行记录
                xi = ((double)i + 0.5) * step;
                local += 4.0 / (1.0 + xi * xi);
            }
        }
        MPI_Reduce(&local, // 发送消息的内存块的指针
                &pi, // 接受消息的内存块的指针
                1, // 数据量
                MPI_DOUBLE, // 数据类型为MPI_DOUBLE
                MPI_SUM, // 规约操作为求和
                0,  // 接受消息的进程号
                MPI_COMM_WORLD);  // 通信子句柄
        pi *= step;
        // root 进行PI 的输出
        if(rankId == 0){
            //endTime = MPI_Wtime();
            endTime = clock();
            printf("PI= %f n= %d Time= %f threads= %d times= %d\n
",
             pi, n, (double)(endTime - startTime)/CLOCKS_PER_SEC,
 nproc, ++times);
        }
    }
    MPI_Finalize();  // 进程的销毁
}
//mpicc MonteCarlo_MPI+OpenMP.c -o MonteCarlo_MPI+OpenMP
//mpirun -np 28 ./MonteCarlo_MPI+OpenMP
```

四、心得体会

通过学习并行程序，了解到编程的另一个世界，原来之前学习过的编程都是比较基础的编程，而在之后的工作或者学习或者研究中，我们还应掌握并行程序的设计！

而且这个学期正好在学操作系统，很多并行的知识在两门课中相互联系，这门实践能更好的帮助我理解进程和线程，而操作系统又在底层给我讲述了什么是进程和线程，以及为什么要使用他们。这使我受益匪浅！

我认为之后的学习工作中，应该建立一个使用并行程序的设计思想的习惯，这样可以提高计算机的运行效率并且节省大量的宝贵时间。

而且在超级计算机领域，并行计算是我们非常需要的，只有利用并行，我们才能够更好的发挥硬件的作用！才能够设计出更好的程序！

五、附录（学习笔记）

CUDA

CUDA 简介

CUDA(Compute Unifide Device Architecture):是英伟达公司(nVIDIA)2006 年 11 月提吃的一种通用并行的架构(应用于计算密集型,SIMD)，是一种通用并行计算平台和编程模型。是随着多核 CPU 和众核 GPU 的出现，而出现的新的并行程序架构。

CUDA 设计为支持多种编程语言和应用程序接口！CUDA 是在底层 API 的基础上，封装了一层，使得程序员可以使用 C 语言来方便的编程。

CUDA 还支持 C++/Python 等更高级的语言编程；

此外，nVIDIA 还提供了 CuDNN，TensorRT，NPP 等更高级的库函数。

CUDA(Compute Unified Device Architecture，计算机统一设备架构)，竞争对手 OpenCL(from 2008，苹果公司)。

CUDA 是 nVIDIA 专有的，即只能用 nVIDIA 的 GPU。

OpenCL 是所有 nVIDIA 主流媒介采用的一种标准，OpenCL 可以在所有平台(nVIDIA,AMD 等)执行，但是能否具有好的运行效果会有差异，同一时刻 CUDA 更快，CUDA 未来会比 OpenCL 发展更快。

计算能力(Compute Capability)

所谓计算能力(Compute Capability)，说白了就是 GPU 的版本号。有时也被称作 SM Version。不同版本的 GPU 具有不同的特性，因此程序编写也会有所差异。计算能力为 X，Y，其中 X 代表架构，Y 代表次版本号。

英伟达 GPU 系列

可扩展的编程模型

CUDA 的编程模型，使得同一个 CUDA 程序，可以在不同的显卡上运行。如图所示，CUDA 程序会创建一些线程块(Block)，线程块会被调度到空闲的流处理簇(SM)上去。当线程执行完毕后，线程块会退出 SM，释放出 SM 的资源，以供其他待执行线程块调度进去。

因此，无论是只有两个 SM 的 GPU，还是有 4 个 SM 的 GPU，这些线程块都会被调度执行，只不过执行的时间有长有短。

GPU 在编程时被称为设备(device)端，CPU 被称为主机(host)端

GPU 段代码需要以核函数的形式加载，执行在 GPU 设备端

线程层级

在讲内核函数前，先讲解一下线程层级。CUDA 编程时一个多线程编程，数个线程(Thread)组成一个线程块(Block)，所有线程块组成一个线程网格(Grid)，图中的线程块，以及线程块的线程，是按照二维的方式排布的。实际上，CUDA 编程模型允许使用一维、二维、三维三种方式来排布。

另外，即使线程块使用的是一维排布，线程块中的线程也不一定要按照一维排，而是可以任意排布。

目前的 GPU 限制一个线程块中，最多可以安排 1024 个线程。一个线程块用多少线程，以及一个线程网格用多少线程块，是程序员可以自由安排的。由于 32 个相邻的线程会组成一个线程束(Tread Wrap)，而一个线程束中的线程会运行同样的指令

因此一般线程块中线程的数量被安排为 32 的倍数，选用 256 是比较合适的。在线程数定下来之后，一般根据数据的排布情况来确定线程块的个数(一维排列 256，二维排列(16,16))

例如：一个数组的长度为 4096，安排每个线程处理一个元素。如果安排一个线程块为 256 个线程，则需要 4096/256=16 个线程块

内核函数(Kernels)

内核函数是 CUDA **每个线程**执行的函数，它运行在 GPU 设备上。CUDA 使用扩展的 C 语言编写内核函数，关键字为*__global__*。内核函数返回值只能是 void。定义格式：

```
__global__ void 函数名 (参数......)
{
    指令集和
}
```

主函数的调用格式：

```
函数名 <<<blocksPerGrid, threadsPerBlock>>>(参数...)
```

blocksPerGrid：每个网格中进程块的排布方式(可以采用一维或二维)

threadsPerBlock：每个进程块中进程的排布方式(可以采用一维或二维)

- 多个 thread 集结成一个 block，多个 bolck 集结成一个 grid。
- 一个 block 里面的线程数，以及一个 grid 里面的 block 数完全是由编程人员去决定的，但是数量的多少和排布方式会根据不同问题产生不同的效果，同时会影响到运行的效率，所以这是 GPU 优化速度的一个关键因素(计算线程和数据的对应关系也不同)
- block 里面的 thread 可以以 1d，2d，3d 的方式进行排布；grid 里面的 block 也可以以 1d，2d，3d 的方式进行排布。
- 每个核函数都可以通过预定义的变量名称去访问各个线程，如 threadidx，blockidx，blockDim，gridDim 的(.x,.y)分量
- 线程块，网格的设定需要用<<<gridset, blockset>>>括起来。
- 内建类型 dim3 可以设定 grid 或 block 里面二维，三维的结构模型，如 dim3 BLOCKs(2，2)

- 核函数前面需要加__global__进行限定，并且必须是没有返回值的 void 函数
- 主机端函数可以是 C++的，但是 kernel 中必须是 C 的，而且很多 C 中字符串相关的函数在目前来说都不能使用

SIMT：相同指令，不同线程

疑问，ppt 和 word 上写的 blockDim.x 的方向不一样。。。

内存层级

同 CPU 一样，GPU 也有不同层级的内存。越靠近核心的内存速度越快，但容量越小；反之，越远离核心的内存速度越慢，但容量较大。

- 主机端内存(host memory)

  指的就是我们常说的内存，一般主机端内存通过 PCI-E 总线与设备端测 i 村交换数据。数据交换的速度等于 PCI-E 总线的速度。

- GPU 的全局内存(global memory)、常量内存(constantly memory)、纹理内存(text memory)、本地内存(local memory)

  都位于 GPU 板上，但不在片内。因此速度相对于片内内存较慢。常量内存和纹理内存对于 GPU 来说是只读的。

  –GPU 上有 L2 cache 和 L1 cache

  其中 L2 cache 为所有流处理簇(SM)共享，而 L1 cache 为每个 SM 内部共享。这里的 cache 和 CPU 的 cache 一样，程序员无法对 cache 显式操纵。

  –纹理缓存和常量缓存再 SM 内部共享

  在早期 1.x 计算能力的时代，这两种缓存是片上唯一的缓存，十分宝贵。而当 Fermi 架构出现后，普通的全局内存也具有了缓存，因此就不那么突出了。

  –共享内存(shared memory,SMEM)

  具有和 L1 缓存同样的速度，且可以被程序员显式操纵，因此经常被用作存放一些需要反复使用的数据。共享内存只能在 SM 内共享，且对于 CUDA 编程模型来说，即使线程块被调度到了同一个 SM 内页无法相互访问。

  –GPU 的寄存器(registers)

  和 CPU 不一样，其空间非常巨大，以至于可以为每一个线程分配一块 dulling 的寄存器空间。因此，不像 CPU 那样切换进程时需要保存上下文，GPU 只需要修改一下寄存器空间的指针即可继续运行。所有巨大的寄存器空间，使得 GPU 上线程切换成为了一个几乎无损的操作。

不过有一点需要注意，寄存器的空间不是无限大的。

寄存器(register)	每个线程独有，可读可写
共享内存(shared memory)	每个 block 独有，可以可写
全局显存(global memry)	任何线程都可以访问读写
常量显存	任何线程都可访问，只读
纹理显存	任何线程都可访问，可读可写

注意事项：

- 寄存器和本地内存绑定到了每个线程，其他线程无法访问。
- 同一个线程块内的线程，可以访问同一块共享内存。即使两个线程块调度到了同一个 SM 上，他们的共享内存也是隔离开的，不能互相访问。
- 网格中的所有线程都可以自由读写全局内存。
- 常量内存和纹理内存只能被 CPU 端修改，GPU 内的线程只能读取数据。

CPU/GPU 混合编程

一种最简单的 CPU/GPU 混合编程方式。程序运行额环境：

主机端 (Host，即 CPU 执行的串行代码)

设备端 (Device，即 GPU 执行的并行代码)

CPU 和 GPU 的内存是独立的。因此在运行内核函数前，主机端需要调用内存拷贝函数，将数据通过 PCI-E 总线拷贝至设备端。内核运行结束后，需要 CPU 再次调用内存拷贝函数，将数据拷贝回主机端内存。

CUDA 并行编程

cuda c 是对 c/c++进行拓展后形成的编程，对 c/c++兼容，文件类型为'.cu'，编译器为 nvcc。cuda c 允许用内核函数来扩展 c，调用时由 N 个不同的线程共执行 N 次。块内的线程通过共享存储器共享数据并通过他们的执行力来协调存储器访问，aka 通过调用_syncthreads()内部函数来指定内核中的同步点。

相比传统的 cpp，添加了以下几个方面：

1)函数类型限定符

2)执行配置运算符

3)五个内置变量

4)变量类型限定符

5)其他的还有数学函数，原子函数，纹理函数，绑定函数等

函数类型限定符

用来确定是在 CPU 还是在 GPU 运行，以及这个函数是从 CPU 还是从 GPU 调用

__device__	表示从 GPU 调用，在 GPU 运行
__globa__	表示从 CPU 调用，在 GPU 执行，也称 kernel 核函数
__host__	表示在 CPU 上调用，在 CPU 上执行

执行配置运算符

用来传递核函数的执行参数。使用__global__声明说明符定义内核用<<<...>>>来为内核指定 cuda 线程数。每一个线程都有一个唯一的 ID，可以通过内置 threadIdx(x,y)、blockIdx(x,y)、blockDim(x,y)来访问。<<<...>>>中可以是 int(通过 dim 定义)或者 dim3 类型。

kNN(k-nearest neighbors)

第一种：分类

针对测试样本 Xu，想要知道它属于那个分类，就先 for 循环所有**训练样本**找出 Xu 最近的 K 个邻居，然后判断这 K 个邻居中，大多数属于哪个类别，就将该类别作为测试样本的预测结果。

第二种：回归

根据样本点，描绘出一条曲线，使得样本点的误差最小，然后给定任意坐标，返回该曲线上的值，叫做回归

你有一系列样本坐标，然后给定一个测试点坐标 x，求回归曲线对应的 y 值。用 kNN 的话，最简单的做法就是取 k 个离 x 最近的样本坐标，然后对他们的 y 值求平均。

算法思路

kNN 算法的基础是对给定的 query 点集，对应查找在参考空间中距离最近的 K 个紧邻点。

kNN 算法存在的一个重要问题是，繁重的计算量。如果参考空间由 M 个点，query 点集有 N 个点，空间维度 d 维，那么计算 query 点集与参考空间各个点集的距离就有 O(mnd)的时间复杂度，仅仅排序就要有 O(m*n*logm)的时间复杂度。因此，整个 kNN 问题的时间复杂度为 O(mnd)+O(mnlogm)。

算法原理

- 通用步骤
  - –计算距离（常用欧几里得距离或马氏距离）
  - –升序排序
  - –取前 k 个
  - –加权平均
- K 的选取
  - –K 太大：导致分类模糊
  - –K 太小：受个例影响
- 然后选取 K
  - –经验
  - –均方根误差