并行程序设计实践项目报告

（小组完成）

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **序号** | **学号** | **姓名** | **班级** | **所参加项目序号** |
| 1 |  |  |  | GPU项目1、GPU项目2  MPI项目1、MPI项目2 |
| 2 |  |  |  |  |
| 3 |  |  |  |  |
| 4 |  |  |  |  |
| 5 |  |  |  |  |

2021年 月 日

# 1. GPU项目1（用CUDA设计一个KNN分类算法（K近邻算法）程序，分类鸢尾花）

一级标题：黑体，小三，加粗。前后6磅行间距，居左。

二级标题：楷体，四号，加粗。前后6磅行间距，居左。

全文1.5倍行距。全文所有阿拉伯数字、英文（代码除外）用Times New Roman字体，中文字体按照批注框格式要求设置。提交时，将所有红色字体及批注框删除。

## 1.1 题目描述

## 在特征空间中查找K个最相似或者距离最近的样本，然后根据K个最相似的样本对未知样本进行分类。通过训练集和测试集给出算法的正确率。

## 1.2. 设计思路

正文：楷体，小四，两端对齐，首行缩进2字符。

## 1.Host和device分配内存，把Host数据初始化并复制到device中。

## 2.KNN算法实现：

## 3. kernel1:并行计算query点和reference点的距离；

## 4. kernel2:多线程并行排序每个query点到空间参考点的距离；

## 5. kernel3:并行计算每个线程中前k个距离的平方跟。

## 6.结果返回到Host的cpu空间。

采用一维数组模拟二维矩阵。

kernel2排序函数对前k个点插入排序，再在后面的点里选择比前面最数小的插入到前面。（此方法啊效率提升显著）

数据集为处理过后的iris数据集，每一列代表一个特征，最后一行为类别

## 1.3 源码

### 1.3.1 串行程序

三级标题：楷体，小四，加粗。前后6磅行间距，居左。

### #include <cstdio>

### #include<iostream>

### #include<cmath>

### #include<ctime>

### //计算矩阵 A（reference points）和 B（query points）中各点的距离平方

### void ComputeDistanceTexture(float \*A,int wA, float \* B, int wB, int dim, float\* AB){

### for(int xIndex=0;xIndex<wB;xIndex++){

### for(int yIndex=0;yIndex<wA;yIndex++){

### if ( xIndex<wB && yIndex<wA ){

### float ssd = 0;

### for (int i=0; i<dim; i++){

### float tmp = A[i\*wA+yIndex] - B[ i \* wB + xIndex ];

### ssd += tmp \* tmp;

### }

### AB[yIndex \* wB + xIndex] = ssd;

### }

### }

### }

### }

### //对距离矩阵排序

### void InsertionSort(float \*dist, int \*ind, int width, int height, int k){

### int l, i, j;

### float \*p\_dist;

### int \*p\_ind;

### float curr\_dist, max\_dist;

### int curr\_row, max\_row;

### int xIndex = 0;

### for(xIndex=0;xIndex<width;xIndex++){

### // 列偏移，最大距离初始化

### p\_dist = dist + xIndex;

### p\_ind = ind + xIndex;

### max\_dist = p\_dist[0];

### p\_ind[0] = 1;

### // Part 1 :对前k行排序（插入排序）

### for (l=1; l<k; l++){

### curr\_row = l \* width;//行偏移

### curr\_dist = p\_dist[curr\_row];

### if (curr\_dist<max\_dist){

### i=l-1;

### for (int a=0; a<l-1; a++){

### if (p\_dist[a\*width]>curr\_dist){

### i=a;

### break;

### }

### }

### for (j=l; j>i; j--){

### p\_dist[j\*width] = p\_dist[(j-1)\*width];

### p\_ind[j\*width] = p\_ind[(j-1)\*width];

### }

### p\_dist[i\*width] = curr\_dist;

### p\_ind[i\*width] = l;

### }

### else

### p\_ind[l\*width] = l;

### max\_dist = p\_dist[curr\_row];

### }

### // Part 2 :选择k行之后的比前面小的插入到前面（类似选择排序）

### max\_row = (k-1)\*width;

### for (l=k; l<height; l++){

### curr\_dist = p\_dist[l\*width];

### if (curr\_dist<max\_dist){

### i=k-1;

### for (int a=0; a<k-1; a++){

### if (p\_dist[a\*width]>curr\_dist){

### i=a;

### break;

### }

### }

### for (j=k-1; j>i; j--){

### p\_dist[j\*width] = p\_dist[(j-1)\*width];

### p\_ind[j\*width] = p\_ind[(j-1)\*width];

### }

### p\_dist[i\*width] = curr\_dist;

### p\_ind[i\*width] = l;

### max\_dist = p\_dist[max\_row];

### }

### }

### }

### }

### void knn(float\* ref\_host, int ref\_width, float\* query\_host,

### int query\_width, int height, int k, float\* dist\_host, int\* ind\_host)

### {

### // 1: Compute all the distances

### ComputeDistanceTexture(ref\_host,ref\_width,

### query\_host, query\_width, height, dist\_host);

### // 2: Sort each column

### InsertionSort(dist\_host, ind\_host,

### query\_width, ref\_width, k);

### }

### float ref[100000000]; // Pointer to reference point array

### float query[100000000]; // Pointer to query point array

### float dist[100000000]; // Pointer to distance array

### int ind[100000000]; // Pointer to index array

### int ref\_nb = 0; // Reference point number, max=65535

### int query\_nb = 3; // Query point number, max=65535

### int dim = 4; // Dimension of points

### int k = 4; // Nearest neighbors to consider

### int CLASS[1000000],classq[1000000],CLASSQ[100000];

### int main()

### {

### clock\_t startTime,endTime;

### startTime = clock();

### //读取训练文件

### freopen("iristrain.txt","r",stdin);

### float aa;

### int i=0;int j=0;

### int flagc=0;

### while(std::cin>>aa)

### {

### if(std::abs(aa-(-1))<0.001){

### flagc=1;

### continue;

### }

### if(flagc==0)

### ref[i++]=aa;

### else

### CLASS[j++]=(int)aa;

### }

### ref\_nb=i/4;

### std::cin.clear();//清除EOF状态

### //读取测试文件

### freopen("iristest.txt","r",stdin);

### i=0;j=0;

### flagc=0;

### while(std::cin>>aa)

### {

### if(std::abs(aa-(-1))<0.001){

### flagc=1;

### continue;

### }

### if(flagc==0)

### query[i++]=aa;

### else

### CLASSQ[j++]=(int)aa;

### }

### query\_nb=i/4;

### //初始化ind

### printf("%d \n",query\_nb);

### for(int i=0;i<ref\_nb;i++)

### {

### for(int j=0;j<query\_nb;j++)

### {

### ind[i\*query\_nb+j]=i;

### }

### }

### printf("Number of reference points : %6d\n", ref\_nb );

### printf("Number of query points : %6d\n", query\_nb);

### printf("Dimension of points : %4d\n", dim );

### printf("Number of neighbors to consider : %4d\n", k );

### printf("Processing kNN search :" );

### knn(ref, ref\_nb, query, query\_nb, dim, k, dist, ind);//进行训练

### int flag1=0,flag2=0,flag3=0;int max=0;

### //进行预测

### for(int j=0;j<query\_nb;j++)

### {

### flag1=0;flag2=0;flag3=0;max=0;

### for(int h=0;h<k;h++)

### {

### if(CLASS[ind[h\*query\_nb+j]]==1)

### flag1++;

### else if(CLASS[ind[h\*query\_nb+j]]==2)

### flag2++;

### else

### flag3++;

### }

### if(flag1>=flag2)

### {

### max=flag1;

### classq[j]=1;

### }

### else

### {

### max=flag2;

### classq[j]=2;

### }

### if(max<flag3)

### classq[j]=3;

### }

### endTime = clock();

### //打印结果

### printf(" done in %lf s \n",(double)(endTime - startTime) / CLOCKS\_PER\_SEC);

### int ac=0;

### for(int i=0;i<query\_nb;i++){

### printf("%d ",classq[i]);

### if(classq[i]==CLASSQ[i])

### ++ac;

### }

### printf("\nAccuracy is : %lf\n",ac\*1.0/query\_nb);

### return 0;

### }

### 1.3.2 GPU程序

### #include <cstdio>

### #include "cuda.h"

### #include<iostream>

### #include<cmath>

### //计算矩阵 A（reference points）和 B（query points）中各点的距离平方

### \_\_global\_\_ void cuComputeDistanceTexture(float \*A,int wA, float \* B, int wB, int dim, float\* AB){

### unsigned int xIndex = blockIdx.x \* blockDim.x + threadIdx.x;

### unsigned int yIndex = blockIdx.y \* blockDim.y + threadIdx.y;

### if ( xIndex<wB && yIndex<wA ){

### float ssd = 0;

### for (int i=0; i<dim; i++){

### float tmp = A[i\*wA+yIndex] - B[ i \* wB + xIndex ];

### ssd += tmp \* tmp;

### }

### AB[yIndex \* wB + xIndex] = ssd;

### }

### }

### //对距离矩阵排序

### \_\_global\_\_ void cuInsertionSort(float \*dist, int \*ind, int width, int height, int k){

### int l, i, j;

### float \*p\_dist;

### int \*p\_ind;

### float curr\_dist, max\_dist;

### int curr\_row, max\_row;

### unsigned int xIndex = blockIdx.x \* blockDim.x + threadIdx.x;

### if (xIndex<width){

### // 列偏移，最大距离初始化

### p\_dist = dist + xIndex;

### p\_ind = ind + xIndex;

### max\_dist = p\_dist[0];

### p\_ind[0] = 1;

### // Part 1 :对前k行排序（插入排序）

### for (l=1; l<k; l++){

### curr\_row = l \* width;//行偏移

### curr\_dist = p\_dist[curr\_row];

### if (curr\_dist<max\_dist){

### i=l-1;

### for (int a=0; a<l-1; a++){

### if (p\_dist[a\*width]>curr\_dist){

### i=a;

### break;

### }

### }

### for (j=l; j>i; j--){

### p\_dist[j\*width] = p\_dist[(j-1)\*width];

### p\_ind[j\*width] = p\_ind[(j-1)\*width];

### }

### p\_dist[i\*width] = curr\_dist;

### p\_ind[i\*width] = l;

### }

### else

### p\_ind[l\*width] = l;

### max\_dist = p\_dist[curr\_row];

### }

### // Part 2 :选择k行之后的比前面小的插入到前面（类似选择排序）

### max\_row = (k-1)\*width;

### for (l=k; l<height; l++){

### curr\_dist = p\_dist[l\*width];

### if (curr\_dist<max\_dist){

### i=k-1;

### for (int a=0; a<k-1; a++){

### if (p\_dist[a\*width]>curr\_dist){

### i=a;

### break;

### }

### }

### for (j=k-1; j>i; j--){

### p\_dist[j\*width] = p\_dist[(j-1)\*width];

### p\_ind[j\*width] = p\_ind[(j-1)\*width];

### }

### p\_dist[i\*width] = curr\_dist;

### p\_ind[i\*width] = l;

### max\_dist = p\_dist[max\_row];

### }

### }

### }

### }

### void knn\_cuda(float\* ref\_host, int ref\_width, float\* query\_host,

### int query\_width, int height, int k, float\* dist\_host, int\* ind\_host)

### {

### int size\_of\_float = sizeof(float);

### int size\_of\_int = sizeof(int);

### //device矩阵

### float \*query\_dev;

### float \*ref\_dev;

### float \*dist\_dev;

### int \*ind\_dev;

### //分配全局显存

### cudaMalloc((void \*\*) &query\_dev, query\_width \* height \* size\_of\_float);

### cudaMalloc((void \*\*) &dist\_dev, query\_width \* ref\_width \* size\_of\_float);

### cudaMalloc((void \*\*) &ind\_dev, query\_width \* k \* size\_of\_int);

### cudaMalloc((void \*\*) &ref\_dev, ref\_width \* height \* size\_of\_float);

### //内容拷贝 host to device

### cudaMemcpy(ref\_dev, &ref\_host[0], ref\_width \* height \* size\_of\_float,

### cudaMemcpyHostToDevice);

### cudaMemcpy(query\_dev, &query\_host[0],

### query\_width \* height \* size\_of\_float, cudaMemcpyHostToDevice);

### cudaMemcpy(ind\_dev, &ind\_host[0],

### ref\_width \* query\_width \* size\_of\_int, cudaMemcpyHostToDevice);

### dim3 g\_16x16((query\_width-1)/16+1, (ref\_width-1)/16+1, 1);

### dim3 t\_16x16(16, 16);

### dim3 g\_256x1((query\_width-1)/256+1,1);

### dim3 t\_256x1(256, 1);

### // Kernel 1: Compute all the distances

### cuComputeDistanceTexture<<<g\_16x16,t\_16x16>>>(ref\_dev,ref\_width,

### query\_dev, query\_width, height, dist\_dev);

### // Kernel 2: Sort each column

### cuInsertionSort<<<g\_256x1,t\_256x1>>>(dist\_dev, ind\_dev,

### query\_width, ref\_width, k);

### // Memory copy of output from device to host

### cudaMemcpy(&dist\_host[0], dist\_dev,

### query\_width \* k \*size\_of\_float, cudaMemcpyDeviceToHost);

### cudaMemcpy(&ind\_host[0], ind\_dev,

### query\_width \* k \* size\_of\_int, cudaMemcpyDeviceToHost);

### // Free memory

### cudaFree(ref\_dev);

### cudaFree(ind\_dev);

### cudaFree(query\_dev);

### cudaFree(dist\_dev);

### }

### float ref[100000000]; // Pointer to reference point array

### float query[100000000]; // Pointer to query point array

### float dist[100000000]; // Pointer to distance array

### int ind[100000000]; // Pointer to index array

### int ref\_nb = 0; // Reference point number, max=65535

### int query\_nb = 3; // Query point number, max=65535

### int dim = 4; // Dimension of points

### int k = 4; // Nearest neighbors to consider

### int CLASS[1000000],classq[1000000],CLASSQ[100000];

### int main()

### {

### //读取训练数据

### freopen("iristrain.txt","r",stdin);

### float aa;

### int i=0;int j=0;

### int flagc=0;

### while(std::cin>>aa)

### {

### if(abs(aa-(-1))<0.001){

### flagc=1;

### continue;

### }

### if(flagc==0)

### ref[i++]=aa;

### else

### CLASS[j++]=(int)aa;

### }

### ref\_nb=i/4;

### std::cin.clear();//清除EOF状态

### //读取测试数据

### freopen("iristest.txt","r",stdin);

### i=0;j=0;

### flagc=0;

### while(std::cin>>aa)

### {

### if(abs(aa-(-1))<0.001){

### flagc=1;

### continue;

### }

### if(flagc==0)

### query[i++]=aa;

### else

### CLASSQ[j++]=(int)aa;

### }

### query\_nb=i/4;

### printf("%d \n",query\_nb);

### //初始化ind

### for(int i=0;i<ref\_nb;i++)

### {

### for(int j=0;j<query\_nb;j++)

### {

### ind[i\*query\_nb+j]=i;

### }

### }

### //准备计时

### cudaEvent\_t start, stop;

### cudaEventCreate(&start);

### cudaEventCreate(&stop);

### float elapsed\_time;

### printf("Number of reference points : %6d\n", ref\_nb );

### printf("Number of query points : %6d\n", query\_nb);

### printf("Dimension of points : %4d\n", dim );

### printf("Number of neighbors to consider : %4d\n", k );

### printf("Processing kNN search :" );

### cudaEventRecord(start, 0);

### knn\_cuda(ref, ref\_nb, query, query\_nb, dim, k, dist, ind);//进行训练

### cudaEventRecord(stop, 0);

### cudaEventSynchronize(stop);

### cudaEventElapsedTime(&elapsed\_time, start, stop);

### //进行预测

### int flag1=0,flag2=0,flag3=0;int max=0;

### for(int j=0;j<query\_nb;j++)

### {

### flag1=0;flag2=0;flag3=0;max=0;

### for(int h=0;h<k;h++)

### {

### if(CLASS[ind[h\*query\_nb+j]]==1)

### flag1++;

### else if(CLASS[ind[h\*query\_nb+j]]==2)

### flag2++;

### else

### flag3++;

### }

### if(flag1>=flag2)

### {

### max=flag1;

### classq[j]=1;

### }

### else

### {

### max=flag2;

### classq[j]=2;

### }

### if(max<flag3)

### classq[j]=3;

### }

### //打印数据

### printf(" done in %f s \n",elapsed\_time/1000);

### int ac=0;

### for(int i=0;i<query\_nb;i++){

### printf("%d ",classq[i]);

### if(classq[i]==CLASSQ[i])

### ++ac;

### }

### printf("\nAccuracy is : %lf\n",ac\*1.0/query\_nb);

### // Destroy cuda event object and free memory

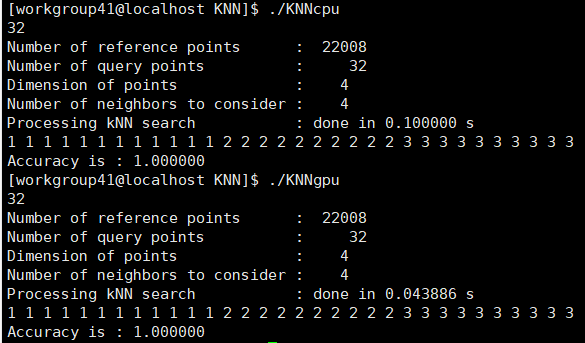
### cudaEventDestroy(start);

### cudaEventDestroy(stop);

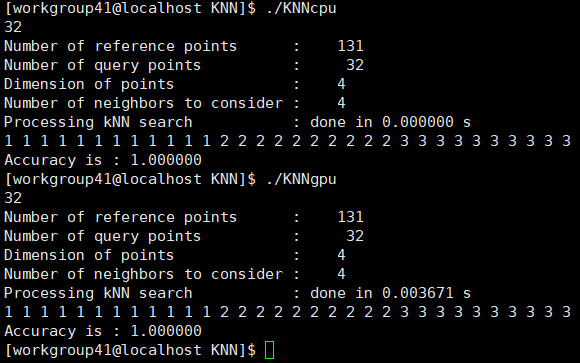
### return 0;

### }

### 1.3.3 性能对比与分析



处理22008条训练数据，cpu用时0.1s，gpu用时0.04s



处理22008条训练数据，cpu用时0.0000s，gpu用时0.00367s

当数据量较小时，cpu运算不用来回传数据，运行速度快。

当数据量较大时，gpu并行计算的速度优势就体现出来了。

图1 图题

图：居中，无首行缩进。必须有图题，按照图1、图2、……进行编号，在正文中必须有引用。

图中文字及图题：楷体，五号。图题居中，无首行缩进。

确保图和图题在同一页。

# 2. GPU项目2（名称）

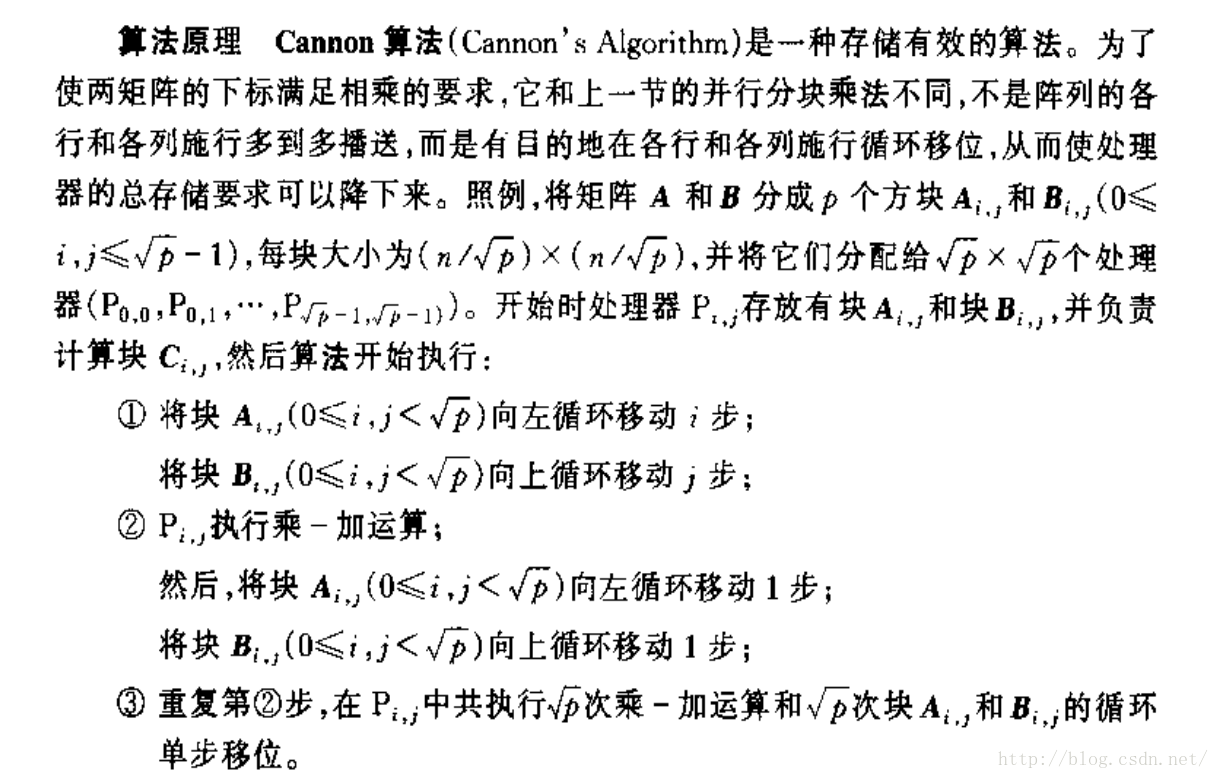
# 3. MPI+OpenMP项目1（名称）

## 2.1 题目描述

实现Cannon算法

## 2.2 设计思路

算法原理：



在本并行算法中需用户指定进程个数q和矩阵行（列）数dg，进程数必须是完全平方数，每个进程存储一个A分块矩阵，一个B分块矩阵，分块矩阵大小(dg/sqrt(q))^2。

A矩阵的左移和B矩阵的上移由进程通讯实现（每个进程都有自己的逻辑编号，表示块的位置）。0进程额外负责A,B矩阵的初始化，将分块矩阵内容发给其他进程，接受其他进程最后处理完的数据。

用mpirun -np ProcNum cannon MatrixDimension来运行程序。

## 2.3 源码

### 2.3.1 串行程序

### #include<stdio.h>

### #include<stdlib.h>

### #include <time.h>

### #include <math.h>

### void multihost(float \*a,float \*b,float \*ccc,int w)

### {

### float e1=0,e2=0;

### for(int i=0;i<w;++i)

### {

### for(int j=0;j<w;++j)

### {

### float v=0;

### for(int k=0;k<w;++k)

### {

### e1=a[i\*w+k];

### e2=b[k\*w+j];

### v+=e1\*e2;

### }

### ccc[i\*w+j]=v;

### }

### }

### }

### float a[10000\*10000],b[10000\*10000],c[100000000];

### int main(int argc, char \*argv[])

### {

### freopen( "outchuan.txt", "w", stdout);

### int w=atoi(argv[1]);

### printf("size=%d\*&d",w,w);

### int maxxx=w\*w;

### clock\_t start2 = 0, stop2 = 0;

### for(int i = 0;i < maxxx;i++)//初始化矩阵

### {

### a[i] = rand();

### b[i] = rand();

### c[i] = 0;

### }

### start2=clock();//计时开始

### multihost(a,b,c,w);

### stop2=clock();//计时结束

### printf("random matrix A : \n");

### for(int i=0;i<w;++i)

### {

### for(int j=0;j<w;++j)

### printf("%15.0f ",a[i\*w+j]);

### printf("\n");

### }

### printf("random matrix B : \n");

### for(int i=0;i<w;++i)

### {

### for(int j=0;j<w;++j)

### printf("%15.0f ",b[i\*w+j]);

### printf("\n");

### }

### printf("Matrix C = A \* B : \n");

### for(int i=0;i<w;++i)

### {

### for(int j=0;j<w;++j)

### printf("%15.0f ",c[i\*w+j]);

### printf("\n");

### }

### double Time = (double)(stop2\*1.0 - start2\*1.0) / CLOCKS\_PER\_SEC;

### printf("Time = %lf s",Time);

### return 0;

### }

### 2.3.2 MPI程序

#include <stdlib.h>

#include <string.h>

#include <mpi.h>

#include <time.h>

#include <stdio.h>

#include <math.h>

/\* 全局变量声明 \*/

float \*\*A, \*\*B, \*\*C; /\* 总矩阵,C = A \* B \*/

float \*a, \*b, \*c, \*tmp\_a, \*tmp\_b; /\* a、b、c表分块，tmp\_a、tmp\_b表缓冲区 \*/

int dg, dl, dl2,p, sp; /\* dg:总矩阵维数;dl:矩阵块维数;dl2=dl\*dl;p:处理器个数;sp＝sqrt(p) \*/

int my\_rank, my\_row, my\_col; /\* my\_rank:处理器ID;(my\_row,my\_col):处理器逻辑阵列坐标 \*/

MPI\_Status status;

/\*

\*函数名: get\_index

\*功能：处理器逻辑阵列坐标至rank号的转换

\*输入：坐标、逻辑阵列维数

\*输出：rank号

\*/

int get\_index(int row, int col, int sp)

{

return ((row+sp)%sp)\*sp + (col+sp)%sp;

}

/\*

\*函数名：random\_A\_B

\*功能：随机生成矩阵A和B

\*/

void random\_A\_B()

{

int i,j;

srand((unsigned int)time(NULL)); /\*设随机数种子\*/

/\*随机生成A,B,并初始化C\*/

for(i=0; i<dg ; i++)

for(j=0; j<dg ; j++)

{

A[i][j] = rand();

B[i][j] = rand();

C[i][j] = 0.0;

}

}

/\* 函数名：scatter\_A\_B

\* 功能：rank为0的处理器向其他处理器发送A、B矩阵的相关块

\*/

void scatter\_A\_B()

{

int i,j,k,l;

int p\_imin,p\_imax,p\_jmin,p\_jmax;

for(k=0; k<p; k++)

{

/\*计算相应处理器所分得的矩阵块在总矩阵中的坐标范围\*/

p\_jmin = (k % sp ) \* dl;

p\_jmax = (k % sp + 1) \* dl-1;

p\_imin = (k - (k % sp))/sp \* dl;

p\_imax = ((k - (k % sp))/sp +1) \*dl -1;

l = 0;

/\*rank=0的处理器将A,B中的相应块拷至tmp\_a,tmp\_b，准备向其他处理器发送\*/

for(i=p\_imin; i<=p\_imax; i++)

{

for(j=p\_jmin; j<=p\_jmax; j++)

{

tmp\_a[l] = A[i][j];

tmp\_b[l] = B[i][j];

l++;

}

}

/\*rank=0的处理器直接将自己对应的矩阵块从tmp\_a,tmp\_b拷至a,b\*/

if(k==0)

{

memcpy(a, tmp\_a, dl2 \* sizeof(float));

memcpy(b, tmp\_b, dl2 \* sizeof(float));

} else /\*rank=0的处理器向其他处理器发送tmp\_a,tmp\_b中相关的矩阵块\*/

{

MPI\_Send(tmp\_a, dl2, MPI\_FLOAT, k, 1, MPI\_COMM\_WORLD);

MPI\_Send(tmp\_b, dl2, MPI\_FLOAT, k, 2, MPI\_COMM\_WORLD);

}

}

}

/\*

\*函数名:init\_alignment

\*功能:矩阵A和B初始对准

\*/

void init\_alignment()

{

/\*将A中坐标为(i,j)的分块A(i,j)向左循环移动i步\*/

MPI\_Sendrecv(a, dl2, MPI\_FLOAT, get\_index(my\_row,my\_col-my\_row,sp), 1,

tmp\_a, dl2, MPI\_FLOAT, get\_index(my\_row,my\_col+my\_row,sp), 1, MPI\_COMM\_WORLD, &status);

memcpy(a, tmp\_a, dl2 \* sizeof(float) );

/\*将B中坐标为(i,j)的分块B(i,j)向上循环移动j步\*/

MPI\_Sendrecv(b, dl2, MPI\_FLOAT, get\_index(my\_row-my\_col,my\_col,sp), 1,

tmp\_b, dl2, MPI\_FLOAT, get\_index(my\_row+my\_col,my\_col,sp), 1, MPI\_COMM\_WORLD, &status);

memcpy(b, tmp\_b, dl2 \* sizeof(float) );

}

/\*

\*函数名：main\_shift

\*功能：分块矩阵左移和上移，并计算分块c

\*/

void main\_shift()

{

int i,j,k,l;

for(l=0; l<sp; l++)

{

/\*矩阵块相乘，c+=a\*b \*/

for(i=0; i<dl; i++)

for(j=0; j<dl; j++)

for(k=0; k<dl; k++)

c[i\*dl+j] += a[i\*dl+k]\*b[k\*dl+j];

/\* 将分块a左移1位 \*/

MPI\_Send(a , dl2, MPI\_FLOAT, get\_index(my\_row, my\_col-1, sp), 1, MPI\_COMM\_WORLD);

MPI\_Recv(a , dl2, MPI\_FLOAT, get\_index(my\_row, my\_col+1, sp), 1, MPI\_COMM\_WORLD, &status);

/\* 将分块b上移1位 \*/

MPI\_Send(b , dl2, MPI\_FLOAT, get\_index(my\_row-1, my\_col, sp), 1, MPI\_COMM\_WORLD);

MPI\_Recv(b , dl2, MPI\_FLOAT, get\_index(my\_row+1, my\_col, sp), 1, MPI\_COMM\_WORLD, &status);

}

}

/\*

\*函数名：collect\_c

\*功能：rank为0的处理器从其余处理器收集分块矩阵c

\*/

void collect\_C()

{

int i,j,i2,j2,k;

int p\_imin,p\_imax,p\_jmin,p\_jmax; /\* 分块矩阵在总矩阵中顶点边界值 \*/

/\* 将rank为0的处理器中分块矩阵c结果赋给总矩阵C对应位置 \*/

for (i=0;i<dl;i++)

for(j=0;j<dl;j++)

C[i][j]=c[i\*dl+j];

for (k=1;k<p;k++)

{

/\*将rank为0的处理器从其他处理器接收相应的分块c\*/

MPI\_Recv(c, dl2, MPI\_FLOAT, k, 1, MPI\_COMM\_WORLD, &status);

p\_jmin = (k % sp ) \*dl;

p\_jmax = (k % sp + 1) \*dl-1;

p\_imin = (k - (k % sp))/sp \*dl;

p\_imax = ((k - (k % sp))/sp +1) \*dl -1;

i2=0;

/\*将接收到的c拷至C中的相应位置,从而构造出C\*/

for(i=p\_imin; i<=p\_imax; i++)

{

j2=0;

for(j=p\_jmin; j<=p\_jmax; j++)

{

C[i][j]=c[i2\*dl+j2];

j2++;

}

i2++;

}

}

}

/\*函数名：print

\*功能：打印矩阵

\*输入：指向矩阵指针的指针，字符串

\*/

void print(float \*\*m,char \*str)

{

int i,j;

printf("%s",str);

/\*打印矩阵m\*/

for(i=0;i<dg;i++)

{

for(j=0;j<dg;j++)

printf("%15.0f ",m[i][j]);

printf("\n");

}

printf("\n");

}

/\*

\*函数名：main

\*功能：主过程，Cannon算法，矩阵相乘

\*输入：argc为命令行参数个数，argv为每个命令行参数组成的字符串数组

\*/

int main(int argc, char \*argv[])

{

freopen( "out.txt", "w", stdout);

int i;

MPI\_Init(&argc, &argv); /\* 启动MPI计算 \*/

MPI\_Comm\_size(MPI\_COMM\_WORLD, &p); /\* 确定处理器个数 \*/

MPI\_Comm\_rank(MPI\_COMM\_WORLD, &my\_rank); /\* 确定各自的处理器标识符 \*/

sp = sqrt(p);

/\* 确保处理器个数是完全平方数，否则打印错误信息，程序退出 \*/

if (sp\*sp != p)

{

if (my\_rank == 0)

printf("Number of processors is not a quadratic number!\n");

MPI\_Finalize();

exit(1);

}

if (argc != 2)

{

if (my\_rank == 0)

printf("usage: mpirun -np ProcNum cannon MatrixDimension\n");

MPI\_Finalize();

exit(1);

}

dg = atoi(argv[1]); /\* 总矩阵维数 \*/

printf("size=%d\*&d",dg,dg);

dl = dg / sp; /\* 计算分块矩阵维数 \*/

dl2 = dl \* dl;

/\* 计算处理器在逻辑阵列中的坐标 \*/

my\_col = my\_rank % sp ;

my\_row = (my\_rank-my\_col) / sp ;

/\* 为a、b、c分配空间 \*/

a = (float \*)malloc( dl2 \* sizeof(float) );

b = (float \*)malloc( dl2 \* sizeof(float) );

c = (float \*)malloc( dl2 \* sizeof(float) );

/\* 初始化c \*/

for(i=0; i<dl2 ; i++)

c[i] = 0.0;

/\* 为tmp\_a、tmp\_b分配空间 \*/

tmp\_a = (float \*)malloc( dl2 \* sizeof(float) );

tmp\_b = (float \*)malloc( dl2 \* sizeof(float) );

if (my\_rank == 0)

{

/\* rank为0的处理器为A、B、C分配空间 \*/

A = (float \*\*)malloc( dg \* sizeof(float\*) );

B = (float \*\*)malloc( dg \* sizeof(float\*) );

C = (float \*\*)malloc( dg \* sizeof(float\*) );

for(i=0; i<dg; i++)

{

A[i] = (float \*)malloc( dg \* sizeof(float) );

B[i] = (float \*)malloc( dg \* sizeof(float) );

C[i] = (float \*)malloc( dg \* sizeof(float) );

}

random\_A\_B(); /\* rank为0的处理器随机化生成A、B矩阵 \*/

scatter\_A\_B(); /\* rank为0的处理器向其他处理器发送A、B矩阵的相关块 \*/

} else /\* rank不为0的处理器接收来自rank为0的处理器的相应矩阵分块 \*/

{

MPI\_Recv(a, dl2, MPI\_FLOAT, 0 , 1, MPI\_COMM\_WORLD, &status);

MPI\_Recv(b, dl2, MPI\_FLOAT, 0 , 2, MPI\_COMM\_WORLD, &status);

}

double start = MPI\_Wtime();//记录时间

init\_alignment(); /\* A、B矩阵的初始对准 \*/

main\_shift(); /\* 分块矩阵左移、上移, cannon算法的主过程 \*/

double end = MPI\_Wtime();

if(my\_rank == 0)

{

collect\_C(); /\* rank为0的处理器从其余处理器收集分块矩阵c \*/

print(A,"random matrix A : \n"); /\* 打印矩阵A \*/

print(B,"random matrix B : \n"); /\* 打印矩阵B \*/

print(C,"Matrix C = A \* B : \n"); /\* 打印矩阵C \*/

} else

{

MPI\_Send(c,dl2,MPI\_FLOAT,0,1,MPI\_COMM\_WORLD); /\* rank不为0的处理器向rank为0的处理器发送矩阵块c \*/

}

MPI\_Barrier(MPI\_COMM\_WORLD); /\* 同步所有处理器 \*/

//freopen("/dev/tty", "w", stdout);//重定向会终端

if(my\_rank == 0)

{

printf("Time = %lf s",end-start);

}

MPI\_Finalize(); /\* 结束MPI计算 \*/

return 0;

}

### 2.3.5性能对比与分析

矩阵大小为625\*625时

并行最佳运行时间：



串行运行时间：



矩阵大小为160\*160时

并行最佳运行时间：



串行运行时间：



由此可见，数据量较大时并行优势，数据量较小时串行程序无需传递数据，串行优势。

查看cpu信息：

28 Intel(R) Xeon(R) CPU E5-2690 v4 @ 2.60GHz

此cpu为28核。

理论上25进程最快，测试如下（size取250\*250）（取多次测试的最短时间）

9进程：



16进程：



25进程：



确实25进程时最快

经实现还发现并行程序的运行时间变化极大，串行程序的运行时间十分稳定。