并行程序设计实践作业报告

|  |  |
| --- | --- |
| 班级： | 智能科学与技术19-2 |
| 姓名： | 曹越 |
| 学号： | 201901061303 |

2021年6月24日

# 并行程序设计学习心得

对编程这一概念又有了深入的了解，明白了多线程，多进程的优势和基本概念。

学会了基础的cuda编程，mpi，openmp编程。

对机器学习、深度学习框架有关tensor计算的底层实现有了更深的理解，学到了用cuda c\c++写pytorch的自定义层。

任务分解不同的程序行为采用不同的线程实现常用于GUI应用程序

数据分解多个线程对不同的数据块执行相同的操作常用于音频、图像处理和科学计算应用程序。本次作业主要用的是数据分解型。

# GPU编程实践作业

# 2.1 题目描述

## 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=500，1000，1500，2000，3000，5000），画出规模和时间对比图。 矩阵A（n，n）矩阵B（n，n），C = A x B

## 2.2 设计思路

首先全部用一维数组模拟二维矩阵，gpu运算用了全局显存和共享显存两种方案，全局显存方案中每个线程计算结果矩阵的一个元素（即A矩阵的一行与B矩阵一列的点乘）；在共享显存方案中将A矩阵和B矩阵分块，每个块大小为blocksize\*blocksize，并行块内元素，主要优点是所有相关线程参与计算同一个相关结果点(即计算一个结果点的过程也是并行的)，但由于没解决函数内for循环的并行问题，所以效果并不好。

## 2.3 源码

### 2.3.1 总程序

### #include<stdio.h>

### #include<stdlib.h>

### #include <time.h>

### #include <math.h>

### #define BlockSize 16

### \_\_global\_\_ void multi(int \*M,int \*N,int \*P,int w)//全局显存核函数

### {

### int e1=0,e2=0,v=0;

### int l=blockIdx.x\*blockDim.x+threadIdx.x;//行

### int r=blockIdx.y \* blockDim.y + threadIdx.y;//列

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

### {

### e1 = M[r\*w+i];//行

### e2 = N[i\*w+l];//列

### v+=e1\*e2;

### }

### P[r\*w+l]=v;//r行l列

### }

### \_\_global\_\_ void multishare(int \*M,int \*N,int \*P,int w)//共享显存和函数

### {

### \_\_shared\_\_ int SM[BlockSize][BlockSize]; //共享显存 分块

### \_\_shared\_\_ int SN[BlockSize][BlockSize];

### int bx = blockIdx.x;

### int by = blockIdx.y;

### int tx = threadIdx.x;

### int ty = threadIdx.y;

### int row = by \* BlockSize + ty;

### int col = bx \* BlockSize + tx;

### int v =0;

### int GS=(int)(ceil((double)w / BlockSize));

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

### {

### if(i\*BlockSize+tx<w&&row<w)

### SM[ty][tx]=M[row\*w+i\*BlockSize+tx];//并行分配块内元素

### else

### SM[ty][tx]=0.0;

### if(i\*BlockSize+ty<w&&col<w)

### SN[ty][tx]=N[(i\*BlockSize+ty)\*w+col];//并行分配块内元素

### else

### SN[ty][tx]=0.0;

### \_\_syncthreads();//block内同步，保证分块矩阵分配完毕

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

### {

### v+=SM[ty][j]\*SN[j][tx];//分块矩阵相乘

### //主要优点，所有相关线程参与计算同一个相关结果点

### }

### \_\_syncthreads();//保证所有的v计算完

### }

### if(row<w&&col<w)

### P[row\*w+col]=v;

### }

### void multihost(int \*a,int \*b,int \*ccc,int w) //host乘积函数

### {

### int e1=0,e2=0;

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

### {

### 

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

### {

### int 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;

### }

### }

### }

### int a[10000\*10000],b[10000\*10000],c[100000000],cc[100000000],ccc[100000000];//乘，被乘，global积，share积,host积

### void MUL(int arg)

### {

### //........................................................................................

### // 全局显存方案

### const int maxx=arg;

### const int maxxx=maxx\*maxx;

### int w=maxx;

### int \*M,\*N,\*P,\*PP;

### cudaMalloc((void\*\*)&M,maxxx \* sizeof(int));//分配显存

### cudaMalloc((void\*\*)&N,maxxx \* sizeof(int));

### cudaMalloc((void\*\*)&P,maxxx \* sizeof(int));

### cudaMalloc((void\*\*)&PP,maxxx \* sizeof(int));

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

### {

### a[i] = (int)(rand() % 10 + 1);;

### b[i] = (int)(rand() % 10 + 1);;

### }

### cudaMemcpy(M,a,maxxx \* sizeof(int),cudaMemcpyHostToDevice);//拷贝数据

### cudaMemcpy(N,b,maxxx \* sizeof(int),cudaMemcpyHostToDevice);

### cudaEvent\_t start,stop;//准备计时

### float elapsedTime = 0;

### cudaEventCreate(&start);

### cudaEventCreate(&stop);

### dim3 blockSize(16,16);

### dim3 gridSize((maxx - 1) / blockSize.x + 1, (maxx - 1) / blockSize.y + 1);

### cudaEventRecord(start,0);//计时开始

### multi<<<gridSize,blockSize>>>(M,N,P,w);

### cudaDeviceSynchronize();

### cudaEventRecord(stop,0);

### cudaEventSynchronize(stop);//计时结束

### cudaEventElapsedTime(&elapsedTime,start,stop);

### cudaMemcpy(c,P,maxxx \* sizeof(int),cudaMemcpyDeviceToHost);//回拷贝

### //................................................................................................

### // 共享显存方案

### cudaMemcpy(M,a,maxxx \* sizeof(int),cudaMemcpyHostToDevice);

### cudaMemcpy(N,b,maxxx \* sizeof(int),cudaMemcpyHostToDevice);

### cudaEvent\_t start1,stop1;

### float elapsedTime1 = 0;

### cudaEventCreate(&start1);

### cudaEventCreate(&stop1);

### cudaEventRecord(start1,0);

### multishare<<<gridSize,blockSize>>>(M,N,PP,w);

### cudaDeviceSynchronize();

### cudaEventRecord(stop1,0);

### cudaEventSynchronize(stop1);

### cudaEventElapsedTime(&elapsedTime1,start1,stop1);

### cudaMemcpy(cc,PP,maxxx \* sizeof(int),cudaMemcpyDeviceToHost);

### //..................................................................................................

### // cpu方案

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

### start2=clock();//计时开始

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

### stop2=clock();//计时结束

### float Time = 1000\*(float)(stop2 - start2) / CLOCKS\_PER\_SEC;

### //...................................................................................................

### printf("timeglobal = %fms N = %d\n",elapsedTime,w);

### // for(int i=0;i<maxx;++i)

### // {

### // for(int j=0;j<maxx;++j)

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

### // printf("\n");

### // }

### 

### printf("timeshare = %fms N = %d\n",elapsedTime1,w);

### // for(int i=0;i<maxx;++i)

### // {

### // for(int j=0;j<maxx;++j)

### // printf("%d ",cc[i\*w+j]);

### // printf("\n");

### // }

### printf("timehost = %fms N = %d\n",Time,w);

### // for(int i=0;i<maxx;++i)

### // {

### // for(int j=0;j<maxx;++j)

### // printf("%d ",ccc[i\*w+j]);

### // printf("\n");

### // }

### printf("\n");

### cudaFree(M);//释放显存空间

### cudaFree(N);

### cudaFree(P);

### cudaFree(PP);

### return;

### }

### int main()

### {

### MUL(500);

### MUL(1000);

### MUL(1500);

### MUL(2000);

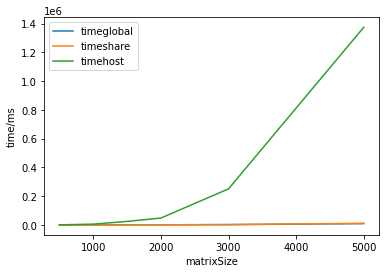
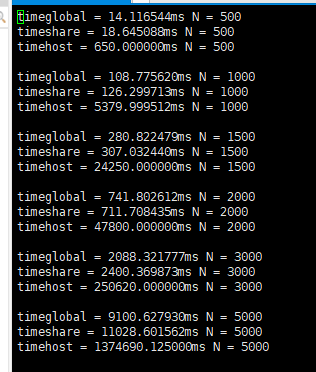
### MUL(3000);

### MUL(5000);

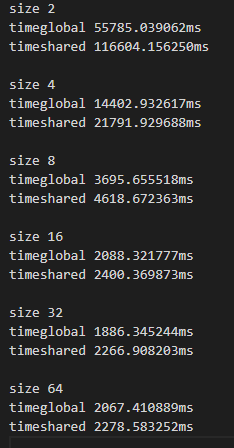
### return 0;

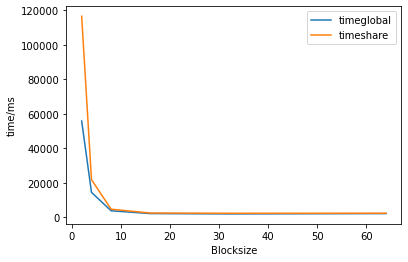
### }

### 2.3.3性能对比与分析



**矩阵大小与运行时间（blocksize为16\*16）**



**Blocksize大小与运行时间（矩阵为3000\*3000）**

可见gpu运算速度远高于cpu运算速度，且cpu运算时间随矩阵变大增长迅速，gpu则增长较慢。

并且在gpu运算时blocksize的大小32\*32最佳。

# 3. MPI+OpenMP编程实践作业

## 3.1 题目描述

分别用mpi，openmp，mpi+openm，串行实现归并排序

## 3.2 设计思路

串行：经典的非递归归并排序

Openmp：在串行的基础上加上编译制导

Mpi：每个进程内先各自排好序。

master进程分批接收数据并进行归并排序，边接收数据边排序

例如：

（1）1-10进程分别给master（0号进程）发10条消息，每条消息中包含100个数

（2）master先分别接收其他进程1条消息，然后10路归并排序，

当某个进程发来的数据全部排完序写到硬盘后，又去读取该进程发的下一条消息边接收边排序可使master只需要1000个数的内存存储数据。

Openmp+mpi：每个进程内排序时用上述openmp的排序，进程间传递同上。

## 3.3 源码

### 3.3.1 串行程序

### #include <stdio.h>

### #include<algorithm>

### #include<cstring>

### #include<iostream>

### #include <time.h>

### //合并

### void merge(int l1, int r1, int r2, int\* data, int\* temp) {

### int top = l1, p = l1, q = r1;

### while (p < r1 || q < r2) {

### if (q >= r2 || (p < r1 && data[p] <= data[q])) {

### temp[top++] = data[p++];

### }

### else {

### temp[top++] = data[q++];

### }

### }

### for (top = l1; top < r2; top++) {

### data[top] = temp[top];

### }

### }

### //归并排序

### void merge\_sort(int l, int r, int\* data, int N) {

### int i, j, t, \*temp;

### temp = (int\*)malloc(N \* sizeof(int));

### //这里做了一些优化，预处理合并了单个的区间，略微提高的速度

### for (i = 0; i < N/2; i++)

### if (data[i\*2] > data[i\*2+1]) {

### t = data[i\*2];

### data[i\*2] = data[i\*2+1];

### data[i\*2+1] = t;

### }

### for (i = 2; i < r; i \*= 2) {

### for (j = 0; j < r-i; j += i\*2) {

### merge(j, j+i, (j+i\*2 < r ? j+i\*2 : r), data, temp);

### } for (int k=0;k<100\*N;k++);

### }

### }

### const int size=5000;

### int arr[size];

### int main()

### {

### FILE\* fp;

### fp = fopen("chuanout.txt", "w");

### clock\_t start=clock();

### srand((unsigned)time(NULL));

### int i;

### for( i =0;i<size;i++)

### {

### arr[i]=rand()%2000;

### }

### merge\_sort(0,size,arr,size);

### for( i =0;i<5000;i++){

### fprintf(fp,"%10d",arr[i]);

### }

### clock\_t end=clock();

### printf("time = %lf s\n",(end\*1.0-start\*1.0)/ CLOCKS\_PER\_SEC);

### }

### 3.3.2 MPI程序

### #include <stdio.h>

### #include <mpi.h>

### #include<algorithm>

### #include<cstring>

### #include<iostream>

### #include <time.h>

### using namespace std;

### const int perins=1000;//每个进程数据的个数

### const int persr=100;//每个进程每次发送的个数

### //合并

### void merge(int l1, int r1, int r2, int\* data, int\* temp) {

### int top = l1, p = l1, q = r1;

### while (p < r1 || q < r2) {

### if (q >= r2 || (p < r1 && data[p] <= data[q])) {

### temp[top++] = data[p++];

### }

### else {

### temp[top++] = data[q++];

### }

### }

### for (top = l1; top < r2; top++) {

### data[top] = temp[top];

### }

### }

### //归并排序

### void merge\_sort(int l, int r, int\* data, int N) {

### int i, j, t, \*temp;

### temp = (int\*)malloc(N \* sizeof(int));

### //这里做了一些优化，预处理合并了单个的区间，略微提高的速度

### for (i = 0; i < N/2; i++)

### if (data[i\*2] > data[i\*2+1]) {

### t = data[i\*2];

### data[i\*2] = data[i\*2+1];

### data[i\*2+1] = t;

### }

### for (i = 2; i < r; i \*= 2) {

### for (j = 0; j < r-i; j += i\*2) {

### merge(j, j+i, (j+i\*2 < r ? j+i\*2 : r), data, temp);

### } for (int k=0;k<(1e6\*N);k++);

### }

### }

### int \* readData(int \*arr,int b) {

### srand((unsigned)time(NULL));

### int i;

### for( i =0;i<perins;i++)

### {

### arr[i]=rand()%2000+b\*rand()%20;

### }

### merge\_sort(0,perins,arr,perins);

### return arr;

### }

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

### FILE\* fp;

### fp = fopen("mpiout.txt", "w");

### int myid, numprocs, namelen;

### char processor\_name[MPI\_MAX\_PROCESSOR\_NAME];

### MPI\_Init(&argc, &argv); // starts MPI

### MPI\_Comm\_rank(MPI\_COMM\_WORLD, &myid); // get current process id

### MPI\_Comm\_size(MPI\_COMM\_WORLD, &numprocs); // get number of processes

### MPI\_Get\_processor\_name(processor\_name, &namelen);

### MPI\_Barrier(MPI\_COMM\_WORLD);//记录时间

### double start = MPI\_Wtime();

### if (myid != 0) {

### int \*a = (int\*) malloc(sizeof(int)\*perins);

### a = readData(a,myid);

### int i;

### for ( i = 1; i < (perins/persr+1); i++) {

### MPI\_Send(a + (i - 1) \* persr, persr, MPI\_INT, 0, i, MPI\_COMM\_WORLD);//如果不是0进程就向0进程发送10个数据（一共发两次）

### }

### }

### else {

### int \*\*pointArr = new int\*[numprocs];//pointArr 0进程的数组

### int i;

### for ( i = 0; i < numprocs; i++) {

### pointArr[i] = new int[persr+1];

### }

### int \*flag = new int[numprocs];//flag标志每个进程的10个数用到了第几个

### memset(flag, 0, sizeof(int)\*numprocs);

### int \*proFlag = new int[numprocs];//proFlag标志0进程接受了每个进程的几次数据（对每个进程 0进程共接受2次数据）

### for ( i = 0; i < numprocs; i++) {

### proFlag[i] = 1;

### }

### for ( i = 1; i < numprocs; i++) {

### MPI\_Recv(pointArr[i], persr+1, MPI\_INT, i, proFlag[i], MPI\_COMM\_WORLD, MPI\_STATUSES\_IGNORE);//接收全部进程的第一次数据

### }

### int min;

### int index = 1;

### int k = 0;

### while (k != perins \* (numprocs - 1)) {

### //初始化min

### int i;

### for ( i = 1; i < numprocs; i++) {

### if (flag[i] != persr) {//如果i进程的数据没用完

### min = pointArr[i][flag[i]];

### index = i;

### break;

### }

### else {

### if (proFlag[i] == (perins/persr)) {//如果i进程的数据用完了，并且已经是最后一轮了

### continue;

### }

### else {//如果i进程的数据用完了，但不是最后一轮，0进程就再接受一次

### proFlag[i]++;

### MPI\_Recv(pointArr[i], persr+1, MPI\_INT, i, proFlag[i], MPI\_COMM\_WORLD, MPI\_STATUSES\_IGNORE);

### flag[i] = 0;

### min = pointArr[i][flag[i]];

### index = i;

### }

### }

### }

### //寻找min

### for ( i = 1; i < numprocs; i++) {

### if (flag[i] == persr) {

### if (proFlag[i] == perins/persr) {

### continue;

### }

### else {

### proFlag[i]++;

### MPI\_Recv(pointArr[i], persr+1, MPI\_INT, i, proFlag[i], MPI\_COMM\_WORLD, MPI\_STATUSES\_IGNORE);

### flag[i] = 0;

### }

### }

### if (min > pointArr[i][flag[i]]) {

### min = pointArr[i][flag[i]];

### index = i;

### }

### }

### //找到当前min后打印

### printf( "%10d", min);

### flag[index]++;

### k++;

### }

### for ( i = 0; i < numprocs; i++) {

### delete pointArr[i];

### }

### delete pointArr;

### delete flag;

### delete proFlag;

### }

### MPI\_Barrier(MPI\_COMM\_WORLD); //记录时间

### double end = MPI\_Wtime();

### MPI\_Finalize();

### if (myid == 0) { /\* use time on root node \*/

### printf("Runtime = %f s\n", end-start);

### }

### return 0;

### }

### 3.3.3 OpenMP程序

### #include <stdio.h>

### #include<algorithm>

### #include<cstring>

### #include<iostream>

### #include<omp.h>

### //合并

### void merge(int l1, int r1, int r2, int\* data, int\* temp) {

### int top = l1, p = l1, q = r1;

### while (p < r1 || q < r2) {

### if (q >= r2 || (p < r1 && data[p] <= data[q])) {

### temp[top++] = data[p++];

### }

### else {

### temp[top++] = data[q++];

### }

### }

### #pragma omp parallel for

### for (top = l1; top < r2; top++) {

### data[top] = temp[top];

### }

### }

### //归并排序

### void merge\_sort(int l, int r, int\* data, int N) {

### int i, j, t, \*temp;

### temp = (int\*)malloc(N \* sizeof(int));

### //这里做了一些优化，预处理合并了单个的区间，略微提高的速度

### #pragma omp parallel for

### for (i = 0; i < N/2; i++)

### if (data[i\*2] > data[i\*2+1]) {

### t = data[i\*2];

### data[i\*2] = data[i\*2+1];

### data[i\*2+1] = t;

### }

### for (i = 2; i < r; i \*= 2) {

### #pragma omp parallel for

### for (j = 0; j < r-i; j += i\*2) {

### merge(j, j+i, (j+i\*2 < r ? j+i\*2 : r), data, temp);

### }

### }

### }

### const int size=5000;

### int arr[size];

### int main()

### {

### FILE\* fp;

### fp = fopen("opmpout.txt", "w");

### double start = omp\_get\_wtime( );

### srand((unsigned)time(NULL));

### int i;

### #pragma omp parallel for

### for( i =0;i<size;i++)

### {

### arr[i]=rand()%2000;

### }

### merge\_sort(0,size,arr,size);

### for( i =0;i<size;i++){

### fprintf(fp,"%10d",arr[i]);

### }

### double end = omp\_get\_wtime( );

### printf("time = %lf s\n",(end-start));

### }

### 3.3.4 MPI+OpenMP程序

### #include <stdio.h>

### #include <mpi.h>

### #include<algorithm>

### #include<cstring>

### #include<iostream>

### #include <time.h>

### using namespace std;

### const int perins=1000;//每个进程数据的个数

### const int persr=100;//每个进程每次发送的个数

### //合并

### void merge(int l1, int r1, int r2, int\* data, int\* temp) {

### int top = l1, p = l1, q = r1;

### while (p < r1 || q < r2) {

### if (q >= r2 || (p < r1 && data[p] <= data[q])) {

### temp[top++] = data[p++];

### }

### else {

### temp[top++] = data[q++];

### }

### }

### #pragma omp parallel for private(top)

### for (top = l1; top < r2; top++) {

### data[top] = temp[top];

### }

### }

### //归并排序

### void merge\_sort(int l, int r, int\* data, int N) {

### int i, j, t, \*temp;

### temp = (int\*)malloc(N \* sizeof(int));

### //这里做了一些优化，预处理合并了单个的区间，略微提高的速度

### #pragma omp parallel for private(i, t) shared(N, data)

### for (i = 0; i < N/2; i++)

### if (data[i\*2] > data[i\*2+1]) {

### t = data[i\*2];

### data[i\*2] = data[i\*2+1];

### data[i\*2+1] = t;

### }

### for (i = 2; i < r; i \*= 2) {

### #pragma omp parallel for private(j) shared(r, i)

### for (j = 0; j < r-i; j += i\*2) {

### merge(j, j+i, (j+i\*2 < r ? j+i\*2 : r), data, temp);

### }

### }

### }

### int \* readData(int \*arr,int b) {

### srand((unsigned)time(NULL));

### int i;

### #pragma omp parallel for private(i) shared(arr,b)

### for( i =0;i<perins;i++)

### {

### arr[i]=rand()%2000+b\*rand()%20;

### }

### merge\_sort(0,perins,arr,perins);

### return arr;

### }

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

### FILE\* fp;

### fp = fopen("mpi\_opmpout.txt", "w");

### int myid, numprocs, namelen;

### char processor\_name[MPI\_MAX\_PROCESSOR\_NAME];

### MPI\_Init(&argc, &argv); // starts MPI

### MPI\_Comm\_rank(MPI\_COMM\_WORLD, &myid); // get current process id

### MPI\_Comm\_size(MPI\_COMM\_WORLD, &numprocs); // get number of processes

### MPI\_Get\_processor\_name(processor\_name, &namelen);

### MPI\_Barrier(MPI\_COMM\_WORLD);//记录时间

### double start = MPI\_Wtime();

### if (myid != 0) {

### int \*a = (int\*) malloc(sizeof(int)\*perins);

### a = readData(a,myid);

### int i;

### for ( i = 1; i < (perins/persr+1); i++) {

### MPI\_Send(a + (i - 1) \* persr, persr, MPI\_INT, 0, i, MPI\_COMM\_WORLD);//如果不是0进程就向0进程发送10个数据（一共发两次）

### }

### }

### else {

### int \*\*pointArr = new int\*[numprocs];//pointArr 0进程的数组

### int i;

### #pragma omp parallel for private(i)

### for ( i = 0; i < numprocs; i++) {

### pointArr[i] = new int[persr+1];

### }

### int \*flag = new int[numprocs];//flag标志每个进程的10个数用到了第几个

### memset(flag, 0, sizeof(int)\*numprocs);

### int \*proFlag = new int[numprocs];//proFlag标志0进程接受了每个进程的几次数据（对每个进程 0进程共接受2次数据）

### #pragma omp parallel for private(i)

### for ( i = 0; i < numprocs; i++) {

### proFlag[i] = 1;

### }

### for ( i = 1; i < numprocs; i++) {

### MPI\_Recv(pointArr[i], persr+1, MPI\_INT, i, proFlag[i], MPI\_COMM\_WORLD, MPI\_STATUSES\_IGNORE);//接收全部进程的第一次数据

### }

### int min;

### int index = 1;

### int k = 0;

### while (k != perins \* (numprocs - 1)) {

### //初始化min

### int i;

### for ( i = 1; i < numprocs; i++) {

### if (flag[i] != persr) {//如果i进程的数据没用完

### min = pointArr[i][flag[i]];

### index = i;

### break;

### }

### else {

### if (proFlag[i] == (perins/persr)) {//如果i进程的数据用完了，并且已经是最后一轮了

### continue;

### }

### else {//如果i进程的数据用完了，但不是最后一轮，0进程就再接受一次

### proFlag[i]++;

### MPI\_Recv(pointArr[i], persr+1, MPI\_INT, i, proFlag[i], MPI\_COMM\_WORLD, MPI\_STATUSES\_IGNORE);

### flag[i] = 0;

### min = pointArr[i][flag[i]];

### index = i;

### }

### }

### }

### //寻找min

### for ( i = 1; i < numprocs; i++) {

### if (flag[i] == persr) {

### if (proFlag[i] == perins/persr) {

### continue;

### }

### else {

### proFlag[i]++;

### MPI\_Recv(pointArr[i], persr+1, MPI\_INT, i, proFlag[i], MPI\_COMM\_WORLD, MPI\_STATUSES\_IGNORE);

### flag[i] = 0;

### }

### }

### if (min > pointArr[i][flag[i]]) {

### min = pointArr[i][flag[i]];

### index = i;

### }

### }

### //找到当前min后打印

### fprintf( fp,"%10d", min);

### flag[index]++;

### k++;

### }

### #pragma omp parallel for private(i)

### for ( i = 0; i < numprocs; i++) {

### delete pointArr[i];

### }

### delete pointArr;

### delete flag;

### delete proFlag;

### }

### MPI\_Barrier(MPI\_COMM\_WORLD); //记录时间

### double end = MPI\_Wtime();

### MPI\_Finalize();

### if (myid == 0) { /\* use time on root node \*/

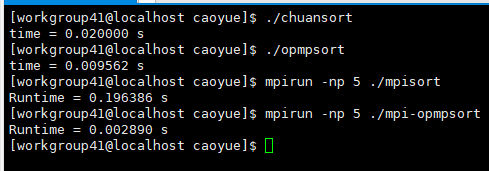
### printf("Runtime = %f s\n", end-start);

### }

### return 0;

### }

### 3.3.5性能对比与分析



数据量500000

（本次截图为数次测试最好结果）

各方法用时如上图，纯mpi性能不佳，opmp+mpi性能最好。（可能是因为最近服务器压力较大，并行程序的运行时间来回跳）