多核平台下的并行计算课程实验报告

姓名：吴文林

学号：202000300086

班级：2020级软件工程6班

# 实验总体概览

## 实验参数

**本次小组实验所采用的软硬件环境参数如下表所示：**

|  |  |
| --- | --- |
| 操作系统版本 | Linux version  4.15.0-76-generic (buildd@lcy01-amd64-029) |
| 并行编程模型 | Pthread |
| 编译器版本 | gcc version 7.5.0 (Ubuntu 7.5.0-3ubuntu1~18.04) |
| CPU型号、主频及核数 | Intel(R) Xeon(R) Gold 6130 CPU @ 2.10GHz \*2  CPU个数：2  内核数：16  线程数：32  缓存：22 MB L3 Cache |
| 内存型号，大小及主频 | 内存型号：DDR4-2666  最大内存大小（取决于内存类型）: 768 GB  最大内存速度：2666 MHz |

表1

## 不同线程数目时的运行时间和加速比

### 2.1 测试数据为：uniformvector-2dim-5h.txt

|  |  |  |
| --- | --- | --- |
| 线程数目 | 运行时间（秒） | 加速比 |
| 1 | 69.38 | 1.0 |
| 2 | 37.82 | 1.87 |
| 4 | 20.30 | 3.42 |
| 8 | 11.11 | 6.25 |
| 16 | 6.211 | 11.17 |
| 32 | 4.710 | 14.73 |
| 64 | 4.286 | 16.19 |

表2

### 测试数据为：uniformvector-4dim-1h.txt

|  |  |  |
| --- | --- | --- |
| 线程数目 | 运行时间（秒） | 加速比 |
| 1 |  | 1.0 |
| 2 |  |  |
| 4 |  |  |
| 8 |  |  |
| 16 |  |  |
| 32 |  |  |
| 64 |  |  |

表3

上述两张表显示的是随着线程数量的增加，运行时间和加速比随着线程数量的变化情况。由上述两表可知，随着线程数的增加，加速比不呈现一个线性变化的趋势。在两个不同数据集上的测试表明，本实验优化后的代码均在64线程的条件下取得了最优秀的性能表现。

# 程序优化分析

## 编译器优化

1. 编译器选项优化

本实验一开始就利用串行程序的编译过程进行优化，gcc 7.5.0的编译优化选项如下所示：

1. **-O0：**不做任何优化，这是默认的编译选项
2. **-O1：**优化会消耗较多的编译时间，它主要对代码的分支，常量以及表达式等进行优化

其中：**-O 和 -O1** : 对程序做部分编译优化，对于大函数,优化编译占用稍微多的时间和相当大的内存。使用本项优化，编译器会尝试减小生成代码的尺寸，以及缩短执行时间，但并不执行需要占用大量编译时间的优化。

1. **-O2：**会尝试更多的寄存器级的优化以及指令级的优化，它会在编译期间占用更多的内存和编译时间
2. **-O3**：在O2的基础上进行更多的优化

在**O2** 编译优化的基础之上，使用普通函数的内联，针对循环做更多的优化操作。

1. **-Ofast：**为了提高程序的执行速度，GCC可以无视严格的语言标准。**-Ofast**会开启所有**-O3**的编译开关，且会对不符合标准的程序进行优化。
2. **-Og：**优化调试信息。相对于-O0生成的调试信息，-Og是为了能够生成更好的调试信息。和-O0一样，-Og选项关闭了很多优化开关。

本实验在串行程序条件下采用**O3** 编译优化选项。

1. SIMD自动向量化

高级向量扩展指令集（Advanced Vector Extensions）是x86架构微处理器中的指令集。这里查看了本机器支持avx512指令集，但是小组成员加入

**-mavx512f**进行测试时发现，测试性能均弱于不加入 **-mavx512f** 时的测试性能，但是又知道 **-O3** 编译选项在编译过程中已经实现了对于SIMD自动向量化的操作，所以本次加入 **-mavx512f** 之后性能无明显提升的原因应该就在此处，故本实验的编译指令中不含有 **-mavx512f** 选项。

## 串行程序分析及其优化

通过分析文件 pivot.c 的代码部分可得，程序首先读取数据文件txt，然后采用递归实现的思路运行坐标重建和切比雪夫距离求和算法，再进行切比雪夫距离和的大小比较以及排序算法，最后返回切比雪夫距离和最大和最小的M=1000个数值和对应的支撑点的集合。

1. 递归算法Combination的分析和优化

程序通过不断的递归调用，依次创建1，2，3，…,k个支撑点，通过数组存储支撑点集合的索引值，根据索引值和数据点的坐标进行重建点坐标的计算，重建点的坐标是其到各支撑点的距离形成的向量。

想要使用Combination并行算法的设计，首先需要对Combination进行改造，这里的改造思路化递归程序为循环，可以更加便利的使用并行优化技术。因此，本小组针对两个数据集，假设线程数量为Thread\_number，主线程计算计算组合数的值，将计算任务平均分配给Thread\_number个线程，那么每个线程分配到的计算量则为/Thread\_number，每个节点计算完毕之后，报告计算结束，等所有子节点计算结束之后，主程序进行排序操作，获得M=1000个最大值和最小值以及支撑点集合。

为了做到上述每个线程能够计算部分递归情况的要求，我们首先需要对Combination方法进行修改，使Combination不在负责递归计算每种结果，而是首先在main函数中利用递归操作将所有情况所对应的索引值存储到combined数组中，在Combination方法中是不断地读取combined数组的情况，然后计算相应值，并存储到自己维护的数组中。

这部分优化的核心代码如下所示：

1. **计算组合数和对应线程数的计算量**

*//compute how much*   
int flag = 0;   
unsigned int tempk = 1;   
**for** (i = 1; i <= k; i++)   
{   
 tempk\*=i;   
}   
howMuchDk = 1;   
*// 计算组合数的值*   
**for** (i = 0; i < k; i+=1)   
{   
 **if**(flag==0)   
 **if**(howMuchDk%tempk==0){   
 howMuchDk/=tempk;   
 flag=1;   
 }   
 howMuchDk\*=(n-i);   
}   
*// printf("%d\n",howMuchDk);*   
howMuch = howMuchDk\*k;

1. **计算所有递归情况并存储到combined数组中：**

int len = 0;

void combine(int pos,int num,int n,int k,int\* temp,int \*a)

{

if (pos==k) // 已有k个数

{

for (int i=0;i<k;i++){

a[len++] = temp[i];

}

return;

}

if(k-pos>n-num+1) return ;

for(int i=num;i<=n;i++)

{

temp[pos]=i;

combine(pos+1,i+1,n,k,temp,a);

}

}

*//combined（main）*

int \* combined = (int\*)malloc(sizeof(int) \* howMuch);

int \* tempCom = (int\*)malloc(sizeof(int) \* k);

combine(0,0,n-1,k,tempCom,combined);

1. **对Combination函数进行串行化操作**

// Combination 函数的串行化   
void\* Combination(void\* in){   
 struct fun\_para \*arg = (struct fun\_para \*)in; // 全局数据结构体指针   
 int \*pivots = (int\*)aligned\_alloc(8,sizeof(int) \* arg->k); // 支撑点坐标索引   
 int i,indexNum;   
 // printf("begin%d:from:%d,to:%d\n",arg->whichP,arg->begin,arg->end);   
 for (i = arg->begin; i < arg->end; i+=arg->k)   
 {   
 int \* pp = pivots;   
 int j;   
 for (j = 0; j < arg->k; j++)   
 {   
 // \*(pp++) = arg->combineData[i+j];   
 memcpy(pp++,arg->combineData+i+j,sizeof(int)); // 复制对应的坐标索引值   
 // printf("data:%d\n",arg->combineData[i+j]);   
   
 }   
 indexNum = i/arg->k;   
 // 计算切比雪夫距离和   
 double distanceSum = SumDistance(arg->k, arg->n, arg->dim, arg->coord, pivots);   
 // printf("%lf\n",arg->minDistanceSum[0]);   
 int mBar = arg->M+1;   
 // 将 distanceSum 存入距离数组   
 memcpy(arg->maxDistanceSum+(arg->M+1)\*(arg->whichP+1)-1,&distanceSum,sizeof(double));   
 memcpy(arg->minDistanceSum+(arg->M+1)\*(arg->whichP+1)-1,&distanceSum,sizeof(double));   
 // 将 distanceSum 对应的支撑点坐标存入对应数组   
 for (j = 0; j < arg->k; j++)   
 {   
 memcpy(arg->maxDisSumPivots+(arg->M+1)\*arg->k\*(arg->whichP) + arg->M\*arg->k+j,arg->combineData+i+j,sizeof(int));   
 memcpy(arg->minDisSumPivots+(arg->M+1)\*arg->k\*(arg->whichP) + arg->M\*arg->k+j,arg->combineData+i+j,sizeof(int));   
 }   
   
 for (j = (arg->M+1)\*(arg->whichP+1)-1; j > (arg->M+1)\*(arg->whichP) +1; j-=1)   
 {   
 // 对新计算出来的切比雪夫距离和进行排序的代码模块   
 }   
 }   
 // printf("end%d:from:%d,to:%d\n",arg->whichP,arg->begin,arg->end);   
 if(!arg->end==howMuch)pthread\_exit(0);   
}

1. 切比雪夫距离和的加速计算的优化
2. 一个成功的尝试

根据分析可得，这里切比雪夫的计算方法的复杂度为O(n^2)，小组成员仔细分析这段代码，想到可以从计算n\*n 个点减少为计算 n\*n/2 个点，于是将这部分的代码改为如下的状态：

// Calculate the sum of Chebyshev distance with rebuilt coordinates between every points   
double chebyshevSum = 0;   
for(i=0; i<n-1; i+=1){ // 修改为 i的上限为n-2   
 int j;   
 for(j=i+1; j<n; j+=1){ // 修改为j = i+1开始   
 double chebyshev = 0;   
 int ki;   
 for(ki=0; ki<k; ki+=1){   
 double dis = fabs(rebuiltCoord[i\*k + ki] - rebuiltCoord[j\*k + ki]);   
 chebyshev = dis>chebyshev ? dis : chebyshev;   
 }   
 chebyshevSum += chebyshev;   
 }   
}

经过这样的修改之后，测试程序可以发现，时间极大地得到了减少，由原来的74s减少到37s左右，经过多轮测试，上述事件均取得了减半的优良性能。

1. 一个没产生效果的尝试-向量化技术

在学长课上讲授的实验指导下，小组成员尝试了向量化的优化思路。这里说明以下向量化技术，SIMD（Single Instruction Multiple Data）是单指令多数据技术，目前Intel处理器支持的SIMD技术包括MMX，SSE，AVX。SSE（Stream SIMD Extentions，数据流单指令多数据扩展）是英特尔继MMX（Multi Media eXtension，多媒体扩展指令集）之后推出的新一代CPU指令集。MMX提供了8个64bit的寄存器进行SIMD操作，SSE系列提供了8个128bit的寄存器进行SIMD操作。而最新的AVX指令则支持256bit的SIMD操作。这里采用SSE指令对于计算切比雪夫距离和进行优化，优化思路如下：

*// 引入的头文件和定义的宏函数*   
#include <mmintrin.h> *//mmx*   
#include <xmmintrin.h> *//sse*   
#include <emmintrin.h> *//sse2*   
#include <pmmintrin.h> *//sse3*   
   
#define likely(x) \_\_builtin\_expect(!!(x), 1)   
#define unlikely(x) \_\_builtin\_expect(!!(x), 0)   
   
*// 优化切比雪夫距离和计算的代码模块*   
 double chebyshevSum = 0;   
 double dis1 = 0;   
 double temp1 = 0;   
 double dis2 = 0;   
 double temp2 = 0;   
 double dis3 = 0;   
 double temp3 = 0;   
 double dis4 = 0;   
 double temp4 = 0;   
 double dis5 = 0;   
 double temp5 = 0;   
 double dis6 = 0;   
 double temp6 = 0;   
 double dis7 = 0;   
 double temp7 = 0;   
 double dis8 = 0;   
 double temp8 = 0;   
 double dis9 = 0;   
 double temp9 = 0;   
 double dis10 = 0;   
 double temp10 = 0;   
 double dis11 = 0;   
 double temp11 = 0;   
 double dis12 = 0;   
 double temp12 = 0;   
 **for** (i = 0; likely((i+11)<n); i += 12) {   
 int j;   
 **for** (j = 0; likely(j < n); j++) {   
 int ki;   
 temp1 = 0;   
 temp2 = 0;   
 temp3 = 0;   
 temp4 = 0;   
 temp5 = 0;   
 temp6 = 0;   
 temp7 = 0;   
 temp8 = 0;   
 temp9 = 0;   
 temp10 = 0;   
 temp11 = 0;   
 temp12 = 0;   
 **for** (ki = 0; ki < k; ki += 1) {   
 dis1 = fabs(rebuiltCoord[i \* k + ki] - rebuiltCoord[j \* k + ki]);   
 temp1 = dis1 > temp1 ? dis1 : temp1;   
 dis2 = fabs(rebuiltCoord[(i + 1) \* k + ki] - rebuiltCoord[j \* k + ki]);   
 temp2 = dis2 > temp2 ? dis2 : temp2;   
 dis3 = fabs(rebuiltCoord[(i + 2) \* k + ki] - rebuiltCoord[j \* k + ki]);   
 temp3 = dis3 > temp3 ? dis3 : temp3;   
 dis4 = fabs(rebuiltCoord[(i + 3) \* k + ki] - rebuiltCoord[j \* k + ki]);   
 temp4 = dis4 > temp4 ? dis4 : temp4;   
 dis5 = fabs(rebuiltCoord[(i + 4) \* k + ki] - rebuiltCoord[j \* k + ki]);   
 temp5 = dis5 > temp5 ? dis5 : temp5;   
 dis6 = fabs(rebuiltCoord[(i + 5) \* k + ki] - rebuiltCoord[j \* k + ki]);   
 temp6 = dis6 > temp6 ? dis6 : temp6;   
 dis7 = fabs(rebuiltCoord[(i + 6) \* k + ki] - rebuiltCoord[j \* k + ki]);   
 temp7 = dis7 > temp7 ? dis7 : temp7;   
 dis8 = fabs(rebuiltCoord[(i + 7) \* k + ki] - rebuiltCoord[j \* k + ki]);   
 temp8 = dis8 > temp8 ? dis8 : temp8;   
 dis9 = fabs(rebuiltCoord[(i + 8) \* k + ki] - rebuiltCoord[j \* k + ki]);   
 temp9 = dis9 > temp9 ? dis9 : temp9;   
 dis10 = fabs(rebuiltCoord[(i + 9) \* k + ki] - rebuiltCoord[j \* k + ki]);   
 temp10 = dis10 > temp10 ? dis10 : temp10;   
 dis11 = fabs(rebuiltCoord[(i + 10) \* k + ki] - rebuiltCoord[j \* k + ki]);   
 temp11 = dis11 > temp11 ? dis11 : temp11;   
 dis12 = fabs(rebuiltCoord[(i + 11) \* k + ki] - rebuiltCoord[j \* k + ki]);   
 temp12 = dis12 > temp12 ? dis12 : temp12;   
 }   
   
 chebyshevSum += temp1;   
 chebyshevSum += temp2;   
 chebyshevSum += temp3;   
 chebyshevSum += temp4;   
 chebyshevSum += temp5;   
 chebyshevSum += temp6;   
 chebyshevSum += temp7;   
 chebyshevSum += temp8;   
 chebyshevSum += temp9;   
 chebyshevSum += temp10;   
 chebyshevSum += temp11;   
 chebyshevSum += temp12;   
 }   
 }

经过上述尝试之后，测试发现，多轮测试结果下该向量化技术并未带来性能上的提升，在部分测试条件下，会拖慢程序的运行速度。据此结果分析原因可得，猜测可能是因为这里的向量化技术优化的并未有 O3 编译优化选项对于代码的向量化优化效果好或者是编译优化选项中，O3编译优化对于程序代码块能够优化的地方均进行了优化处理。综合分析可得，采取前面一种优化方式，舍弃该向量化方式的优化。最后经查询可得，O3编译优化选项包含自动SIMD，表明支持第二种可能的原因。

## Pthread程序优化思路及分析

### Pthread 并行算法设计

前文提到，程序将计算任务平均分配给各个子线程进行计算，每个子线程计算完毕之后通知主线程子线程已经计算完毕。由于Pthread是共享内存编程模型，故首先创建结构体存放共享类型变量。

// 共享变量的结构体   
struct fun\_para{   
int k; int n; int dim; int M;int whichP;unsigned int begin;unsigned int end;int\* maxDisSumPivots;int\* combineData;int\* minDisSumPivots;   
double\* coord;double\* maxDistanceSum; double\* minDistanceSum;   
};

随后定义number(线程数)个这样的结构体，每个结构体包含计算任务的开始和结束。在创建线程之前告诉系统我将要创建子线程(#pragma omp parallel for)，然后使用for循环创建子线程，并为每个子线程创建计算起止点，调用Combination函数完成计算任务，完成计算任务之后，返回计算完成的状态，那么主线程等待计算完成，等所有主线程都计算完毕，主线程则开始下一步的操作。这里每个子线程都维护一个长度为M+1(M=1000)的int数组和一个长度k\*(M+1) (M=1000)的int类型数组用于存放切比雪夫距离和的最大值和最小值和其对应的重建点的坐标。具体实现如下所示：

#pragma omp parallel for   
for (i = 0; i < number; i++)   
{   
 // 每个子线程创建和属性值的分配   
 paras[i].n = n;   
 paras[i].k = k;   
 paras[i].dim = dim;   
 paras[i].M = M;   
 paras[i].combineData = combined;   
 paras[i].coord = coord;   
 paras[i].maxDistanceSum = maxDistanceSum;   
 paras[i].maxDisSumPivots = maxDisSumPivots;   
 paras[i].minDistanceSum = minDistanceSum;   
 paras[i].minDisSumPivots = minDisSumPivots;   
 paras[i].begin = pre\_num;   
 paras[i].end = pre\_num+every\_count;   
 paras[i].whichP = i;   
 pre\_num += every\_count;   
 // 创建子线程   
 status = pthread\_create(&threads[i],NULL,Combination,(void\*)&paras[i]);   
 if(status!=0){//线程创建不成功，打印错误信息   
 printf("pthread\_create returned error code %d\n",status);   
 exit(-1);   
 }   
}

当所有子线程报告计算完毕之后，主线程对number\*(M+1)的数组进行最大值和最小值排序(这里的排序只需要选出M(M=1000)个最大和最小的值即可)，并交换对应的支撑点集合数组的对应索引即可得到最大M(M=1000) 的支撑点集合坐标以及对应的切比雪夫距离和的值。

下面给出了找出M(M=1000)个最大值/最小值的算法代码：

int indexNum;   
int howMuchDistance = (number+1)\*(M+1);   
for (i = 0; i < M; i++)   
{   
 int maxIndex;   
 int minIndex;   
 int j;   
 // 寻找最大/最小切比雪夫距离和   
 for (maxIndex=minIndex=j = i; j < howMuchDistance; j++)   
 {   
 if (maxDistanceSum[j]>maxDistanceSum[maxIndex])   
 {   
 maxIndex = j;   
 }   
 if (minDistanceSum[j]<minDistanceSum[minIndex])   
 {   
 minIndex = j;   
 }   
 }   
 // 交换对应的支撑点集合   
 double temp = maxDistanceSum[maxIndex];   
 maxDistanceSum[maxIndex] = maxDistanceSum[i];   
 maxDistanceSum[i] = temp;   
 int kj;   
 for(kj=0; kj<k; kj+=1){   
 int temp = maxDisSumPivots[maxIndex\*k + kj];   
 maxDisSumPivots[maxIndex\*k + kj] = maxDisSumPivots[i\*k + kj];   
 maxDisSumPivots[i\*k + kj] = temp;   
 }   
 sort(maxDisSumPivots+i\*k,k); // 保证支撑点集合的顺序   
   
 temp = minDistanceSum[minIndex];   
 minDistanceSum[minIndex] = minDistanceSum[i];   
 minDistanceSum[i] = temp;   
 for(kj=0; kj<k; kj+=1){   
 int temp = minDisSumPivots[minIndex\*k + kj];   
 minDisSumPivots[minIndex\*k + kj] = minDisSumPivots[i\*k + kj];   
 minDisSumPivots[i\*k + kj] = temp;   
   
 }   
 sort(minDisSumPivots+i\*k,k);   
}

在加入Pthread编程模型之后，也就是上述方法，在 64 线程的机器上测试出来的用时平均为 2s 。也就是说，加入Pthread之后，程序性能得到了极大地的提升，线程数与加速比的对应关系请参阅表2和表3。在加入Pthread编程模型之后，小组成员还对其他的程序代码进行了细微的优化，这部分将在3.2节详细说明。

### 其他的优化策略

1. 数组初始化的优化

在数组初始化部分，小组成员考虑到大数组的数组初始化比较费时间，于是考虑调用已有的更加快速的API完成对于数组的初始化，以提高程序的运行速度。memset：作用是在一段内存块中填充某个给定的值，它是对较大的**结构体**或**数组**进行清零操作的一种最快方法。实现的代码如下所示：

// maxDistanceSum : the largest M distance sum   
double\* maxDistanceSum = (double\*)aligned\_alloc(8,sizeof(double) \* (number+1)\*(M+1));   
memset(maxDistanceSum,0,(number+1)\*(M+1)); // memset 数组初始化   
// maxDisSumPivots : the top M pivots combinations   
int\* maxDisSumPivots = (int\*)malloc(sizeof(int)\* (number+1)\*(M+1)\*k);   
memset(maxDisSumPivots,0,(number+1)\*(M+1)\*k); // memset 数组初始化   
// minDistanceSum : the smallest M distance sum   
double\* minDistanceSum = (double\*)aligned\_alloc(8,sizeof(double) \* (number+1)\*(M+1));   
for(i=0; i<(number+1)\*(M+1); i++){   
minDistanceSum[i] = \_\_DBL\_MAX\_\_;   
}   
// memset(minDistanceSum,\_\_DBL\_MAX\_\_,(M+1));   
   
// minDisSumPivots : the bottom M pivots combinations   
int\* minDisSumPivots = (int\*)malloc(sizeof(int) \* (number+1)\*(M+1)\*k);   
memset(minDisSumPivots,0,(number+1)\*(M+1)\*k); // memset 数组初始化

1. 简化点与点之间距离的计算

小组成员发现，在重建点的坐标的阶段，因为含有计算距离的操作，涉及大量的浮点数运算操作，且大部分浮点数运算操作是重复操作。考虑到计浮点数乘法运算的运行时间一般要小于直接访问内存取得该值的时间，于是考虑将n个点两两之间的距离存储下来，以便在重建坐标时能够大大减少计算量，在重建点的坐标的时候直接从数组中读取相应的距离值。这是典型的空间换取时间的思想，但是需要考虑到n的数目大小，如果n的数目过大，这样的计算在时间复杂度和空间复杂度上表现都很差，所以小组成员对小数据集做了这样的测试，发现每个子线程模块的计算时间并未有明显的提升，与原来的时间相差无几。

// 计算n个点两两之间的距离的算法   
distance = (double\*)aligned\_alloc(8,sizeof(double) \* n\*n);   
double \*dp = distance;   
for (i = 0; i < n; i+=1)   
{   
 int j;   
 for(j=0; j<n; j+=1){   
 double dis = 0;   
 int k;   
 for (k = 0; k < dim; k+=1)   
 {   
 dis += (coord[i\*dim+k] - coord[j\*dim + k])\*   
 (coord[i\*dim+k] - coord[j\*dim + k]);   
 }   
 dis = sqrt(dis);   
 memcpy(dp++,&dis,sizeof(double));   
 }   
}

小组成员考虑到，这种方法适合n(点个数)和k(选择的基准点个数)中n比较小，k比较大的场景，当n=500，k=2是与直接计算的结果表现无明显区别。这种方法不具有普适性，由于在程序进行测试时需要考虑到普适性的特征，于是小组成员讨论之后放弃这样的思路及其实现。

1. 数组赋值的优化。类似于memset，我们采用C库函数memcpy对数组赋值，当需要将一个地址所指向的值赋于另一个地址时，采用此方法可以有效对复制过程进行内存的优化：

memcpy(arg->maxDistanceSum+(arg->M+1)\*(arg->whichP+1)-1,&distanceSum,sizeof(double));

# 实验总结

本次实验小组成员选择Pthread编程模型进行程序的优化，POSIX线程（英语：POSIX Threads，常被缩写为pthreads）是POSIX的线程标准，定义了创建和操纵线程的一套API。实现POSIX线程标准的库常被称作pthreads，一般用于Unix-like POSIX系统，如Linux、 Solaris。选择Pthread主要是因为本次实验可以采用共享变量的形式进行优化，MPI由于其存在大量的线程之间的通信，会影响到程序的运行速度，随着线程数的增加，通信效率也会收到很大的影响。而本次使用的Pthread编程模型在每个子线程进行独立运算的过程中，只需要报告主线程子线程已经计算完毕程序已经退出即可，不涉及大量的线程间的通信对程序性能的影响的优化。

由表2和表3显示，本次实验的加速比不是线性的。加速比是指顺序执行算法的执行时间与有p个处理器时，并行算法的执行时间的比值。若该比值与处理器的数量成线性正比关系，就可以称之为线性加速比。当某一并行算法的加速比为理想加速比时，若将处理器数量加倍，执行速度也会加倍，即如“理想”之意，有“优秀的可扩展性”。本次实验并未做到线性加速比，考虑到当线程数量增多时，每个线程的创建时间，计算运行时间都会存在时间消耗的情况。而本次采用的Pthread线程模型，在子线程计算的过程中，子线程计算时间的差异性和主线程接受子线程计算结束的信号标志的最大值都影响了程序的运行时间，故很难做到线性加速比，但是仍旧期待“线性加速比”。

本次实验，小组3位成员齐心协力，尽管遇到许多问题，但可以一同讨论解决问题。选择不同的机器进行测试，由于测试机器硬件和软件的差异性，对于同一个程序在不同机器上运行的时间结果的差异性也一定程度上影响了实验的推进。最后选择在实验室的机器上进行测试，在优化的过程中，小组成员尝试了多种解决方案，有的方案比较有效果，有的方案效果没有非常明显甚至会起到反向优化的作用，经过不断的尝试和总结，不断的进行改进，得到了上述的优化模型。本次实验小组成员利用所学知识，将其应用于实践，做到了“实践出真知”，不断尝试，追求更加卓越！