《高性能地理计算》
实习报告

学　　院：　遥感信息工程学院

班　　级：　2006

学　　号：　2020302131043

姓　　名：　吴斌文

指导教师：　乐鹏

2022 年 11 月 12 日

目录

1. 实习内容

1.1 实习目的

本次实习的主要目的在于帮助同学们熟悉高性能计算的相关方法与理论，至少了解一种并行计算优化方法，具体实现对某一算法的并行化设计，将理论课程学习到的专业知识应用到本次实践中来，深入了解高性能计算的基本原理和操作方法。并能够对并行化设计程序的结果进行合理的解释与分析。

1.2 实习要求

可以下面三选一，也可以选择 GPGPU 等其他与课程相关的技术：

①基于 MPI 并行编程实验

②基于 OpenMP 并行编程实验

③基于 MapReduce 云计算编程实验

注意事项：

①可使用单机或者集群环境

②主要考察对算法的并行化设计，不必对算法本身做过多的研究

③需要对并行化结果进行分析（可使用图表等方式）

1.3 个人选题介绍

①选题：

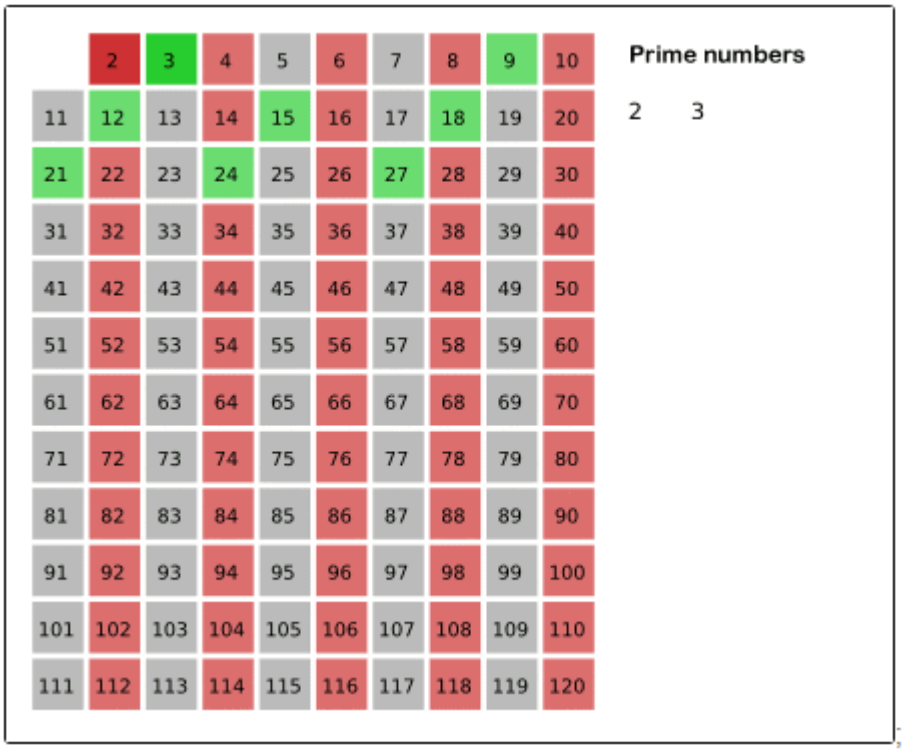
《基于 MPI 的埃拉托斯特尼筛法的并行化设计、实现与结果分析》

②算法介绍

埃拉托斯特尼是一位古希腊数学家，他在寻找整数 N 以内的素数时，采用了一种与众不同的方法：先将 2～N 的各个数写在纸上：

在 2 的上面画一个圆圈，然后划去 2 的其他倍数；第一个既未画圈又没有被划去的数是 3，将它画圈，再划去 3 的其他倍数；现在既未画圈又没有被划去的第一个数是 5，将它画圈，并划去 5 的其他倍数……依此类推，一直到所有小于或等于 N 的各数都画了圈或划去为止。这时，画了圈的以及未划去的那些数正好就是小于 N 的素数。

如下图所示：



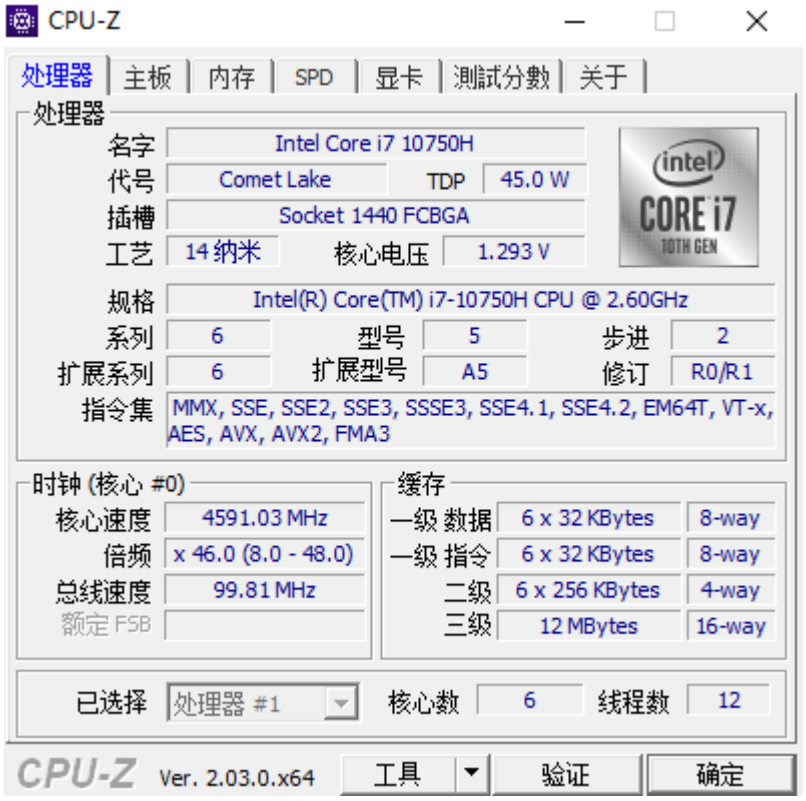
其伪代码如下所示：

```
Input: an integer n > 1

Let A be an array of Boolean values, indexed by integers 2 to n,
initially all set to true.

 for i = 2, 3, 4, ..., not exceeding √n:
  if A[i] is true:
```

```
8.      for j = i2, i2+i, i2+2i, i2+3i, ..., not exceeding n :
9.        A[j] := false
10.
11. Output: all i such that A[i] is true.
```

1.4 实验环境

Windows：

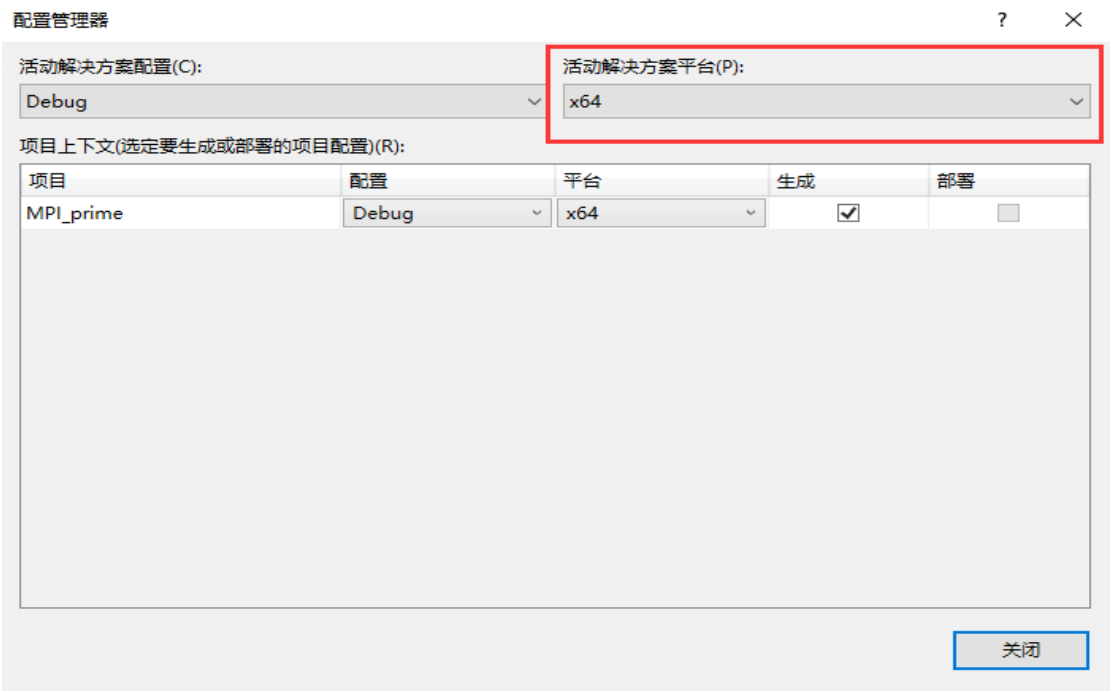


开发环境： Visual Studio 2019, MSMPI v10.0

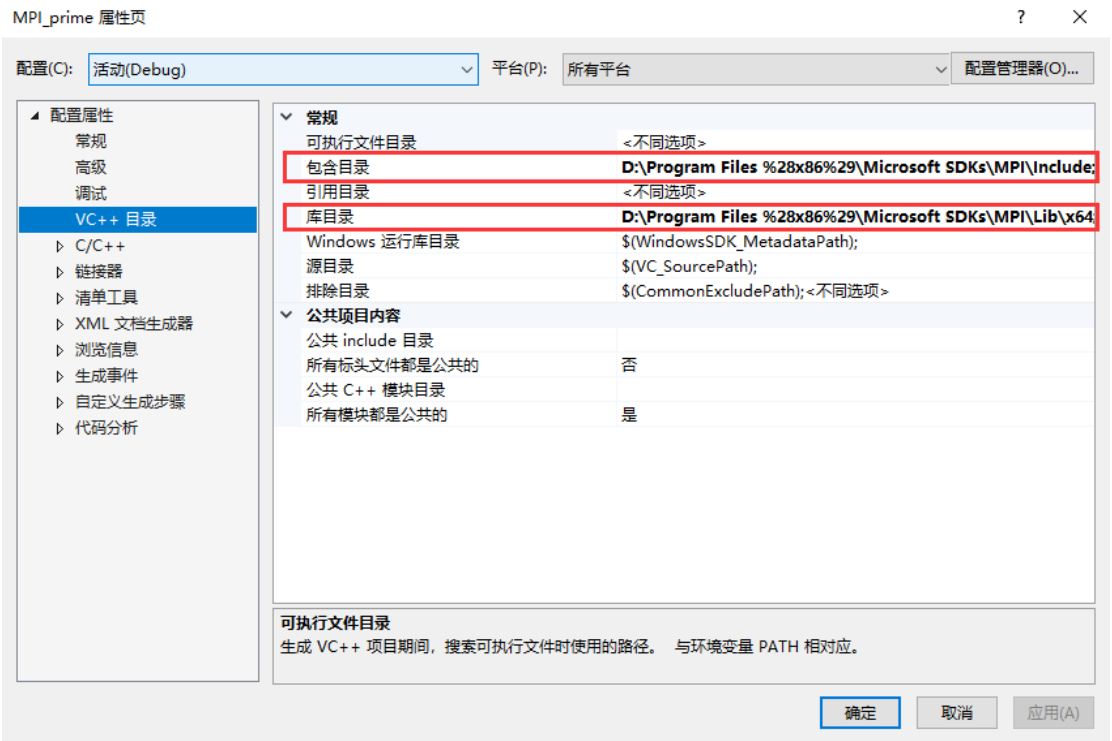
MPI 环境配置（本次实验为 Windows 环境）：

windows 下运行 mpi 首推微软的 msmp，因为比较简单，下载地址为：https://docs.microsoft.com/en-us/message-passing-interface/microsoft-mpi，将两个安装包 msmpisdk.msi 和 msmpisetup.exe 分别下载然后安装完成后即可，下面是在 VS2019 中引入 MSMPI 的步骤：

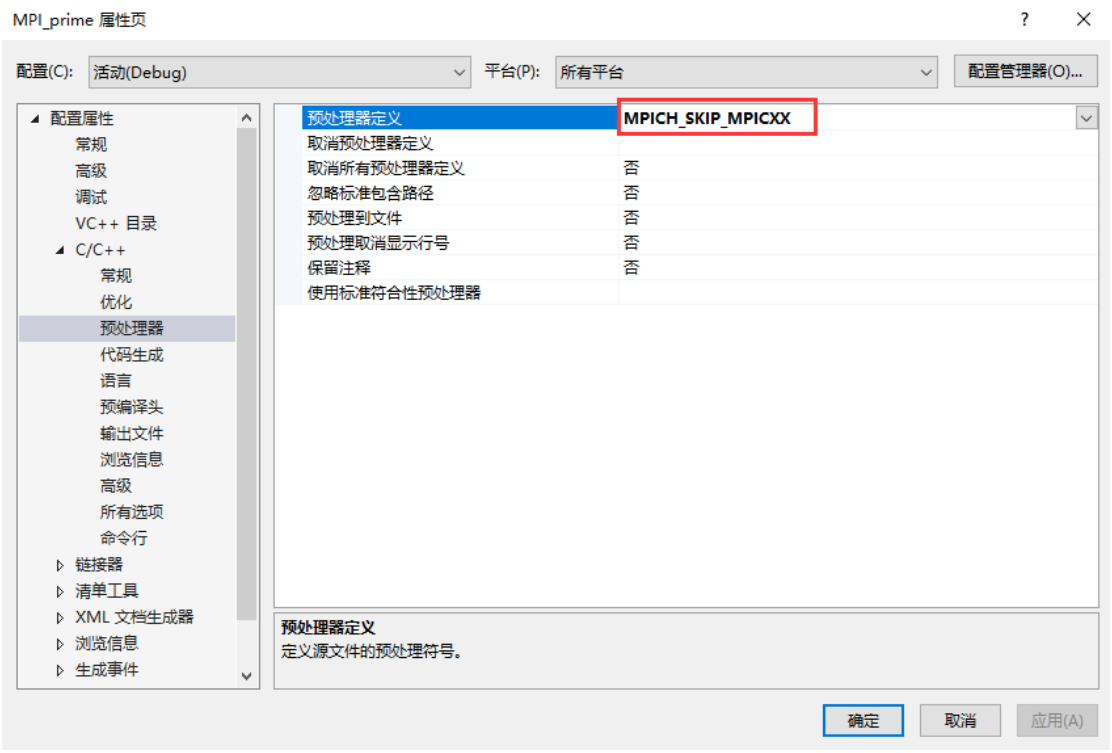
①在 VS 中新建 C++控制台应用程序，将项目编译改为 X64



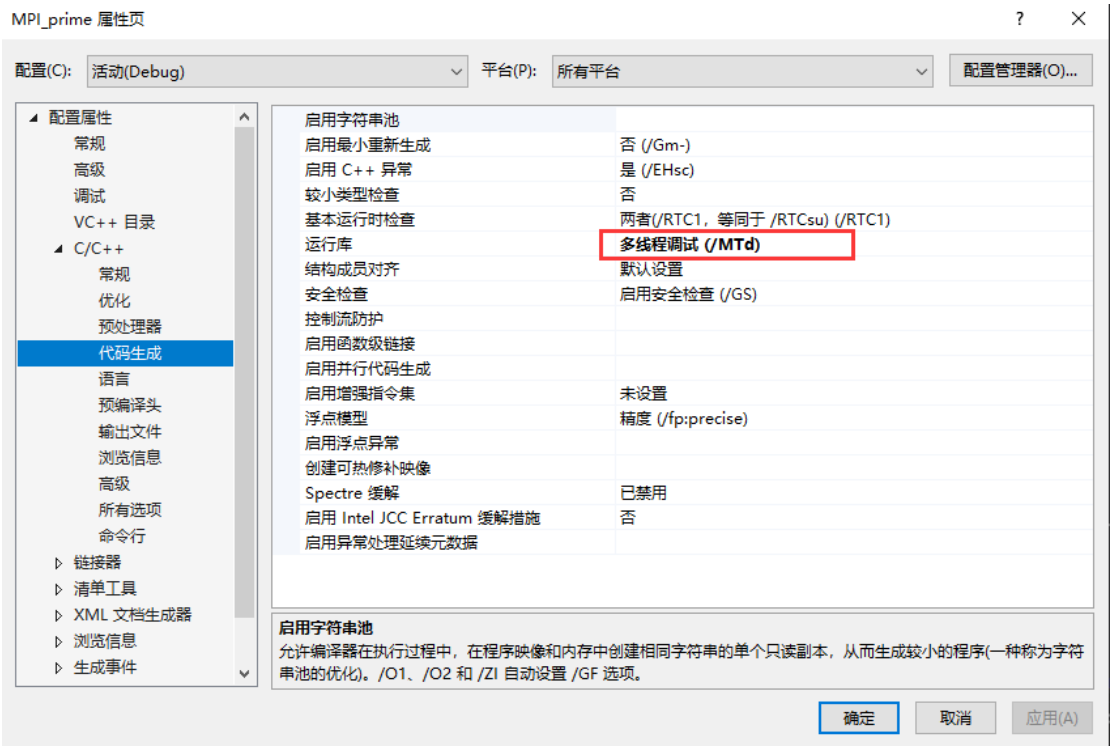
②去安装的 SDK 目录，找到 include 与 lib 文件夹右键项目 -- 属性 -- vc++ 目录中包含目录添加 include 文件夹路径，库目录中添加 lib 文件夹路径。



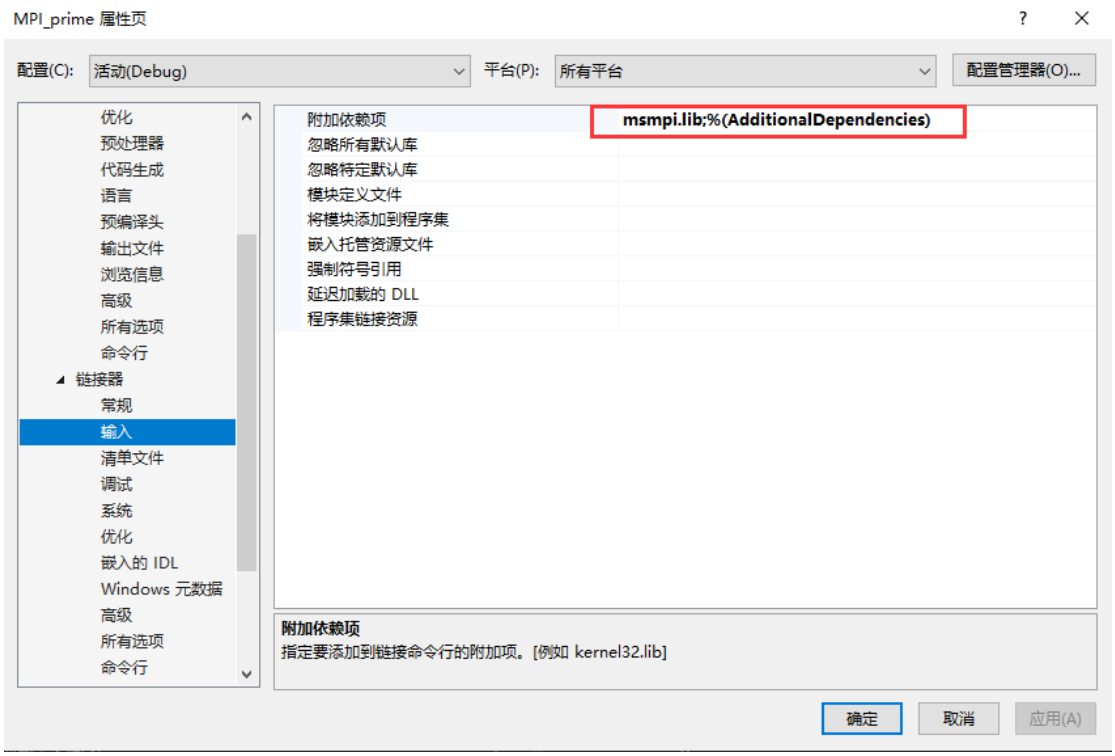
③C/C++ ->预处理器 -> 预处理器定义 -> 添加 MPICH_SKIP_MPICXX



④C/C++ -> 代码生成 -> 运行库 ->选择：多线程调试(/MTD)



⑤属性 -- 链接器 -- 输入 -- 附加依赖项中添加 msmpi.lib;



Linux 下配置 MPICH

①安装：

```
1. sudo apt-get install mpic
```

②CmakeLists.txt 下配置

```
1. cmake_minimum_required(VERSION 3.13)
2. project(MPI)
3. set(CMAKE_CXX_STANDARD 17)
4. find_package(MPI REQUIRED)
5. include_directories(${MPI_INCLUDE_PATH})
6. set(CMAKE_CXX_COMPILER mpicxx)
7. set(CMAKE_C_COMPILER mpicc)
8. add_executable(MPI main.cpp)
```

2. 具体流程

2.1 串行

2.1.1 源码

```
#include<stdio.h>
#include<stdlib.h>
#include <iostream>
#include <time.h>
using namespace std;
int main() {
std::vector<int> Prime(int n){
    if(n<=0){
        std::cout<<"n<=0"<<std::endl;
        exit(-1);
    }
    bool isPrime[n];
    std::fill(&isPrime[0],&isPrime[0]+n,true);
    int n1=sqrt(n);
    isPrime[0]=isPrime[1]=false;
    for(int k=2;k<=n1;k++){
        for(int j=k*k;j<=n;j+=k){
            isPrime[j]=false;
        }
    }
    std::vector<int> v;
    for(int k=2;k<n;++k){
        if(isPrime[k]){
            v.push_back(k);
        }
    }
    return v;
}
```

2.1.2 时间复杂度 $O(nln\,n)$

内层循环一共计算$|n \div k| - k + 1$，循环总数$A \approx n\left(\frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{\sqrt{n}}\right) - \sqrt{n}(\sqrt{n} - 1)/2$，又因为$\lim_{n\to\infty} \sum_k^n \frac{1}{k} - \ln n = \gamma$，当 n->∞ 时，

$$A = \Theta(n(\tfrac{1}{2}\ln n + \gamma - 1) - \tfrac{1}{2}(n - \sqrt{n})) = \Theta(n \ln n)$$

2.2 MPI 优化

在进行优化前，让我们再回顾一下 Eratosthenes 筛法的步骤：

1、创建一个自然数 2，3，4，…，n 的列表，其中所有自然数都被标记为 0。

2、令 k=2，它是列表中第一个未被标记的自然数。

3、重复下面步骤，直到$k^2 > n$为止。

（a）找出k^2和 n 之间的是 k 的倍数的数并标记为 1。

（b）找到比 k 大的未被标记的数中最小的那个，令 k 等于这个数。

4、列表中未被标记的数就是质数。

2.2.1 优化思路

下面使用 MPI 并行计算加快这一算法，将数组分为 p 个连续的块，每个块的大小基本相等，为了平衡负载，要给每个进程分配 $[\frac{n}{p}]$ 个元素，我们考虑使用下面的实现方法：

1、首先计算$r = n \bmod p$，前 r 个进程分配$[\frac{n}{p}]$个元素，后面 p-r 个进程分配$[\frac{n}{p}]$个元素。

2、进程 i 控制的第一个元素是$i\left[\frac{i}{p}\right] + \min(i, r)$，最后一个元素是$(i+1)\left[\frac{n}{p}\right] + \min(i+1, r) - 1$，对特定元素 j，控制它的进程是$\min\left(\left[\frac{j}{[\frac{n}{p}]+1}\right], \left[\frac{j-r}{[\frac{n}{p}]}\right]\right)$。

```
int N = n - 1;
low_index = id * (N / p) + MIN(id, N % p); // 进程的第一个数的索引
high_index = (id + 1) * (N / p) + MIN(id + 1, N % p) - 1; // 进程的最后一个数的索引
low_value = 2 + low_index; //进程的第一个数
high_value = 2 + high_index;//进程的最后一个数
size = high_value - low_value + 1;     //进程处理的数组大小
```

3、0 进程分到的数据块的大小$proc0_{size} = \left[\frac{n-1}{p}\right]$，我们用 0 进程来存储步骤 3（b）中用于筛选的 k 值(即 2 到$\sqrt{n}$的质数)，所以程序运行的前提要求是$2 + proc0_{size} \geq (int)sqrt((double)n)$

```
proc0_size = (n - 1) / p;
// 如果有太多进程
if ((2 + proc0_size) < (int)sqrt((double)n)) {
    if (!id) printf("Too many processes\n");
    MPI_Finalize();
    exit(1);
}
```

4、对每个进程都提供一个 marked[size]这样的数组，prime 保存当前用于筛选的质数，first 表示进程 id 中第一个要求被筛掉的数对应的 marked 数组的下标。index 用于步骤 3（b）中找到的比 prime 大的未被标记的数中最小的那个数，index 为 0 进程专属。核心部分的程序为：

```
    // 先假定所有的整数都是素数
    for (int i = 0; i < size; i++) marked[i] = 0;
    // 索引初始化为 0
    if (!id) index = 0;//0 进程专属
    // 从 2 开始搜寻
    prime = 2;
    do {
        /*确定该进程中素数的第一个倍数的下标 */
       // 如果该素数 n*n>low_value，n*(n-i)都被标记了
       // 即 n*n 为该进程中的第一个素数
       // 其下标为 n*n-low_value
        if (prime * prime > low_value)
            first = prime * prime - low_value;
        else {
            // 若最小值 low_value 为该素数的倍数
           // 则第一个倍数为 low_value，即其下标为 0
            if (!(low_value % prime)) first = 0;
            // 若最小值 low_value 不是该素数的倍数
               // 那么第一个倍数的下标为该素数减去余数的值
```

```
            else first = prime - (low_value % prime);
        }
        // 从第一个素数开始，标记该素数的倍数为非素数
        for (int i = first; i < size; i += prime) marked[i] = 1;
        // 只有 id=0 的进程才调用，用于找到下一素数的位置
        if (!id) {
            while (marked[++index]); // 先自加再执行
            prime = index + 2;// 起始加偏移
        }
        // 只有 id=0 的进程才调用，用于将下一个素数广播出去
        if (p > 1) MPI_Bcast(&prime, 1, MPI_INT, 0, MPI_COMM_WORLD);
    } while (prime * prime <= n);
```

其中 MPI_Bcast(&buffer,count,datatype,root,comm)表示标号为 root 的进程发送相同的消息给通信域 comm 中所有的进程，消息的内容为三元组：&buffer,count,datatype。

2.2.2 源码

```
#include "mpi.h"
#include <math.h>
#include <stdio.h>
#include<stdlib.h>
using namespace std;
/*****************************************************
MPI_BCAST(buffer,count,datatype,root,comm)
     IN/OUT  buffer      通信消息缓冲区的起始地址(可变)
     IN       count       通信消息缓冲区中的数据个数(整型)
     IN       datatype    通信消息缓冲区中的数据类型(句柄)
     IN       root        发送广播的根的序列号(整型)
     IN       comm        通信子(句柄)
int MPI_Bcast(void* buffer,int count,MPI_Datatype datatype,int root, MPI_Comm
comm)

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.

规约函数 MPI_Reduce()，将通信子内各进程的同一个变量参与规约计算，并向指定的进程输出计
算结果
MPI_METHOD MPI_Reduce(
```

```
    _In_range_(!= , recvbuf) _In_opt_ const void* sendbuf,  // 指向输入数据的指针
    _When_(root != MPI_PROC_NULL, _Out_opt_) void* recvbuf, // 指向输出数据的指针，即计算结果存放的地方
    _In_range_(>= , 0) int count,                           // 数据尺寸，可以进行多个标量或多个向量的规约
    _In_ MPI_Datatype datatype,                             // 数据类型
    _In_ MPI_Op op,                                         // 规约操作类型
    _mpi_coll_rank_(root) int root,                         // 目标进程号，存放计算结果的进程
    _In_ MPI_Comm comm                                      // 通信子
);
*******************************************/
#define BLOCK_LOW(id, p, n) ((id) * (n) / (p))
#define BLOCK_HIGH(id, p, n) (BLOCK_LOW((id) + 1, p, n) - 1)
#define BLOCK_SIZE(id, p, n) (BLOCK_LOW((id) + 1) - BLOCK_LOW(id))
#define BLCOK_OWNER(index, p, n) (((p)* (index) +1 ) -1 / (n))
#define MIN(a, b) ((a)<(b)?(a):(b))
void write_txtfile(double k, int num_pro, int num)
{
    FILE* pFile = fopen("../../../result/result_Initial.txt", "a+");
    if (NULL == pFile)
    {
        printf("error");
        return;
    }
    fprintf(pFile, "Thread num:%d,Arrange:%d, Time:%10.6f\n", num_pro, num, k);
    fclose(pFile);
    return;
}
int main(int argc, char* argv[])
{
    long long int    count;        /* Local prime count */
    double elapsed_time;           /* Parallel execution time */
    long long int    first;        /* Index of first multiple */
    long long int    global_count; /* Global prime count */
    long long int    high_value;   /* Highest value on this proc */
    long long int    i;
    int    id;                     /* Process ID number */
    long long int    index;        /* Index of current prime */
    long long int    low_value;    /* Lowest value on this proc */
    char* marked;                  /* Portion of 2,...,'n' */
    long long int    n;            /* Sieving from 2, ..., 'n' */
```

```
60.     int    p;                       /* Number of processes */
61.     long long int    proc0_size;    /* Size of proc 0's subarray */
62.     long long int    prime;         /* Current prime */
63.     long long int    size;          /* Elements in 'marked' */
64.     int low_index;          /* Lowest index on this proc */
65.     int high_index;         /* Highest index on this proc */
66.     // 初始化
67.     // MPI 程序启动时“自动”建立两个通信器：
68.     // MPI_COMM_WORLD:包含程序中所有 MPI 进程
69.     // MPI_COMM_SELF：有单个进程独自构成，仅包含自己
70.     MPI_Init(&argc, &argv);
71.     // MPI_COMM_RANK 得到本进程的进程号，进程号取值范围为 0, …, np-1
72.     MPI_Comm_rank(MPI_COMM_WORLD, &id);
73.     // MPI_COMM_SIZE 得到所有参加运算的进程的个数
74.     MPI_Comm_size(MPI_COMM_WORLD, &p);
75.     // MPI_Barrier 是 MPI 中的一个函数接口
76.     // 表示阻止调用直到 communicator 中所有进程完成调用
77.     MPI_Barrier(MPI_COMM_WORLD);
78.     // MPI_WTIME 返回一个用浮点数表示的秒数
79.     // 它表示从过去某一时刻到调用时刻所经历的时间
80.     elapsed_time = -MPI_Wtime();
81.     // 参数个数为 2：文件名以及问题规模 n
82.     if (argc != 2) {
83.         if (!id) printf("Command line: %s <m>\n", argv[0]);
84.         // 结束 MPI 系统
85.         MPI_Finalize();
86.         exit(1);
87.     }
88.     // 表示找 <= n 的素数
89.     n = atoi(argv[1]);
90.     /* Figure out this process's share of the array, as
91.        well as the integers represented by the first and
92.        last array elements */
93.     int N = n - 1;
94.     low_index = id * (N / p) + MIN(id, N % p); // 进程的第一个数的索引
95.     high_index = (id + 1) * (N / p) + MIN(id + 1, N % p) - 1; // 进程的最后一个
    数的索引
96.     low_value = 2 + low_index; //进程的第一个数
97.     high_value = 2 + high_index;//进程的最后一个数
98.     size = high_value - low_value + 1;    //进程处理的数组大小
99.     /* Bail out if all the primes used for sieving are not all held by process
     0 */
100.        proc0_size = (n - 1) / p;
101.        // 如果有太多进程
```

```
102.     if ((2 + proc0_size) < (int)sqrt((double)n)) {
103.         if (!id) printf("Too many processes\n");
104.         MPI_Finalize();
105.         exit(1);
106.     }
107.     /* Allocate this process's share of the array. */
108.     marked = (char*)malloc(size);
109.     if (marked == NULL) {
110.         printf("Cannot allocate enough memory\n");
111.         MPI_Finalize();
112.         exit(1);
113.     }
114.     // 先假定所有的整数都是素数
115.     for (int i = 0; i < size; i++) marked[i] = 0;
116.     // 索引初始化为 0
117.     if (!id) index = 0;//0 进程专属
118.     // 从 2 开始搜寻
119.     prime = 2;
120.     do {
121.         /*确定该进程中素数的第一个倍数的下标 */
122.        // 如果该素数 n*n>low_value，n*(n-i)都被标记了
123.        // 即 n*n 为该进程中的第一个素数
124.        // 其下标为 n*n-low_value
125.         if (prime * prime > low_value)
126.             first = prime * prime - low_value;
127.         else {
128.             // 若最小值 low_value 为该素数的倍数
129.            // 则第一个倍数为 low_value，即其下标为 0
130.             if (!(low_value % prime)) first = 0;
131.             // 若最小值 low_value 不是该素数的倍数
132.                // 那么第一个倍数的下标为该素数减去余数的值
133.             else first = prime - (low_value % prime);
134.         }
135.         // 从第一个素数开始，标记该素数的倍数为非素数
136.         for (int i = first; i < size; i += prime) marked[i] = 1;
137.
138.         // 只有 id=0 的进程才调用，用于找到下一素数的位置
139.         if (!id) {
140.             while (marked[++index]); // 先自加再执行
141.             prime = index + 2;// 起始加偏移
142.         }
143.
144.         // 只有 id=0 的进程才调用，用于将下一个素数广播出去
145.         if (p > 1) MPI_Bcast(&prime, 1, MPI_INT, 0, MPI_COMM_WORLD);
```

```
146.        } while (prime * prime <= n);
147.        // 将标记结果发给 0 号进程
148.        count = 0;
149.        for (int i = 0; i < size; i++)
150.            if (!marked[i]) count++;
151.        if (p > 1) MPI_Reduce(&count, &global_count, 1, MPI_INT, MPI_SUM,
152.            0, MPI_COMM_WORLD);
153.        else global_count = count;
154.        /* Stop the timer */
155.        elapsed_time += MPI_Wtime();
156.        MPI_Finalize();
157.        // print the results
158.        if (!id) {
159.            printf("***************| All %ld |**************\n", n);
160.            printf("***************| SIEVE(%d) |************** \n ************
    ***| it tastes:time: %10.6fs |***************\n***************| Find %d primes
     |***************\n", p, elapsed_time, global_count);
161.        }
162.        if (!id) write_txtfile(elapsed_time, p, n);
163.        return 0;
164.    }
```

2.2.3 时间复杂度

χ：执行二元操作所需要的时间。

λ：经由通道将一个整数传到另一个通道所需要的时间。

素数定理：$\pi(x)\sim\frac{x}{lnx}$，其中$\pi(x)$表示不超过 x 的素数的个数。

2 到$\sqrt{n}$内，找到一个质数需要一个广播，所以通信的预期时间为$\left(\frac{\sqrt{n}}{\ln\sqrt{n}}\right)\lambda\log_2 p$，对 n 个质数，p 个进程，需要筛选$\frac{n}{p}$次，一共需要$\sum_p\frac{n}{p}$，又因为素数的倒数和为$\sum_p\frac{1}{p}\sim O(\ln\ln n)$，所以 p 个进程的筛选的预期时间粗略估计为$\chi(n\ln\ln n)/p$，总预期时间为$\sum_p\frac{1}{p}\sim O(\ln\ln n)+\left(\frac{\sqrt{n}}{\ln\sqrt{n}}\right)\lambda\log_2 p$。

2.3 去偶数优化

2.3.1 优化思路

利用已知除 2 以外的所有偶数都不是素数的常识，可以将待筛选数字总量减半，从而提高筛选效率。关键在于数组减半，找到新的索引映射，以及首个倍数（非素数）的位置。这样筛选的预期时间约为 $\chi(n \ln \ln n)/(2p)$

2.3.2 关键代码

```
1.      int N = (n - 1) / 2;
2.      low_index = id * (N / p) + MIN(id, N % p); // 进程的第一个数的索引
3.      high_index = (id + 1) * (N / p) + MIN(id + 1, N % p) - 1; // 进程的最后一个
    数的索引
4.      low_value = low_index * 2 + 3; //进程的第一个数
5.      high_value = (high_index + 1) * 2 + 1;//进程的最后一个数
6.      size = (high_value - low_value) / 2 + 1;    //进程处理的数组大小
7.      // Bail out if all the primes used for sieving are not all held by process
    0
8.      proc0_size = (n - 1) / p;
9.      // 如果有太多进程
10.     if ((2 + proc0_size) < (int)sqrt((double)n)) {
11.         if (!id) printf("Too many processes \n");
12.         MPI_Finalize();
13.         exit(1);
14.     }
15.     // allocate this process 's share of the array
16.     marked = (char*)malloc(size);
17.     if (marked == NULL) {
18.         printf("Cannot allocate enough memory \n");
19.         MPI_Finalize();
20.         exit(1);
21.     }
22.     // 先假定所有的整数都是素数
23.     for (int i = 0; i < size; i++) marked[i] = 0;
24.     // 索引初始化为 0
```

```
25.     if (!id) index = 0;
26.     // 从 3 开始搜寻，first 为第一个不是素数的位置
27.     prime = 3;
28.     do {
29.         /*确定该进程中素数的第一个倍数的下标 */
30.        // 如果该素数 n*n>low_value，n*(n-i)都被标记了
31.        // 即 n*n 为该进程中的第一个素数
32.        // 其下标为 n*n-low_value，并且由于数组大小减半所以除以 2
33.         if (prime * prime > low_value) {
34.             first = (prime * prime - low_value) / 2;
35.         }
36.         else {
37.             // 若最小值 low_value 为该素数的倍数
38.           // 则第一个倍数为 low_value，即其下标为 0
39.             if (!(low_value % prime)) first = 0;
40.             // 若最小值 low_value 不是该素数的倍数
41.             // 但是其余数为偶数，那么第一个非素数的索引为该素数剪去求余除以 2
42.             else if (low_value % prime % 2 == 0)
43.                  first = prime - ((low_value % prime) / 2);
44.             // 若最小值 low_value 不是该素数的倍数
45.             // 那么第一个倍数的下标为该素数减去余数的值，并且由于数组大小减半所以除以
    2
46.             else first = (prime - (low_value % prime)) / 2;
47.         }
48.         // 从第一个素数开始，标记该素数的倍数为非素数
49.         for (int i = first; i < size; i += prime)
50.             marked[i] = 1;
51.         // 只有 id=0 的进程才调用，用于找到下一素数的位置
52.         if (!id) {
53.             while (marked[++index]);
54.             prime = index * 2 + 3; // start of 3
55.         }
56.         // 只有 id=0 的进程才调用，用于将下一个素数广播出去
57.         if (p > 1) {
58.             MPI_Bcast(&prime, 1, MPI_INT, 0, MPI_COMM_WORLD);
59.         }
60.     } while (prime * prime <= n);
```

①当 prime = 3，考虑序列 15 17 19 21 23，则满足 low_value % prime == 0，所以索引 0 即第一个数 15 就是 prime 的倍数，这个很好理解。

②当 prime = 3，考虑序列 3 5 7 9 11 13，则满足 prime * prime > low_value，那么 first = (prime * prime - low_value) / 2 即第一个非素数索引为 3 即值为 9，这个和前面的一样，主要是减去一个偏移，不难理解。

比较难以理解的是后面两个判断：

③当 prime = 3，考虑序列 17 19 21 23 25，则满足 prime * prime <= low_value，并且满足 low_value % prime % 2 == 0，那么 first = prime - ((low_value % prime) / 2)，即第一个非素数的位置为 0，即值为 15。

④当 prime = 3，考虑序列 11 13 15 17 19，则满足 prime * prime <= low_value，并且满足 low_value % prime % 2 != 0，那么 first = (prime - (low_value % prime)) / 2，即第一个非素数索引为 2，即值为 15。

⑤我理解了很久，我认为主要是由于每次遇到 prime 与 2 的公倍数的时候性质会发生改变，当 prime = 3，考虑原有序列 7 8 9 10 11 12 13 14 15 16 17，其中 12 为 2 和 3 的公倍数，排除除 12 以外的其他偶数，该序列为 7 9 11 12 13 15 17，再排除 3 的倍数，该序列为 7 11 12 13 17，问题的本质在于小于等于该数(7,11,13,17)最近的并且为 prime(3)和 2 的公倍数 A 是否为偶数。

⑥对于 11，17 而言，即满足条件 3，A 为 9，15 为奇数，偏差 bias（求余）为 2，所以可以更进一步判断 bias 是否为偶数，由于其下

一个 prime 倍数的位置只能取 [0, prime - 1]，故而 prime 下一个倍数所在的位置为 prime - bias / 2。

⑦对于 7，13 而言，即满足条件 4，A 为 6，12 为偶数，偏差 bias（求余）为 1，同样由于其下一个 prime 倍数的位置只能取 [0, prime - 1]，故而 prime 下一个倍数所在的位置为 (prime - bias) / 2。

2.4 去广播优化

2.4.1 优化思路

原理：初始的代码是通过进程 0 广播下一个筛选倍数的素数。进程之间需要通过 MPI_Bcast 函数进行通信。通信就一定会有开销，特别是在分布式计算机架构上，因此我们让每个进程都各自找出它们的前 sqrt(n) 个数中的素数，再通过这些素数筛选剩下的素数，这样一来进程之间就不需要每个循环广播素数了，只需要最后通信归总结果，这样只需要$\lambda[\log_2 p]$的通信时间。与去偶数优化结合，则一共只需要 $\frac{\chi(n\ln\ln n)}{2p}+\frac{\chi(n\ln\ln n)}{p}+\lambda[\log_2 p]$ 。

2.4.2 关键代码

```
1.  //小数组
2.  char* NewMarked;              /*portion of 2...(in newarray)*/
3.  int low_NewArray_index;       /* Lowest index on NewArray*/
4.  int low_NewArray_value;       /* Lowest valuue on NewArray*/
5.  int high_NewArray_index;      /* highest index on NewArray*/
6.  int high_NewArray_value;      /* highest value on NewArray*/
```

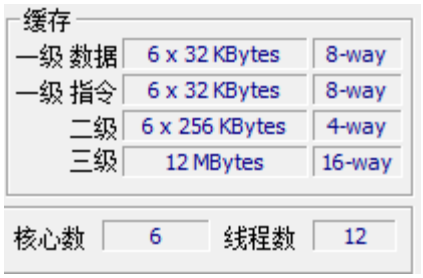
```
/* 广播优化  */
    //先找前 sqrt(n)内的素数,再通过这些素数筛选后续素数
    int sqrt_N = (((int)sqrt((double)n)) - 1) / 2;
    low_NewArray_value = MIN(0, sqrt_N % p) * 2 + 3; //进程的第一个数
    high_NewArray_value = ((sqrt_N / p) + MIN(1, sqrt_N % p)) * 2 + 1;//进程的最后一个数
    // Bail out if all the primes used for sieving are not all held by process 0
    proc0_size = (sqrt_N - 1) / p;
    // 如果有太多进程
    if ((2 + proc0_size) < (int)sqrt((double)sqrt_N)) {
        if (!id) printf("Too many processes \n");
        MPI_Finalize();
        exit(1);
    }
    NewMarked = (char*)malloc(sqrt_N);
    if (NewMarked == NULL) {
        printf("Cannot allocate enough memory \n");
        MPI_Finalize();
        exit(1);
    }
    // 先假定所有的整数都是素数
    for (int i = 0; i < sqrt_N; i++)
        NewMarked[i] = 0;
    // 索引初始化为 0
    index = 0;
    prime = 3;
    do {
        // 从小数组开始标只会命中第一个条件
        first = (prime * prime - low_NewArray_value) / 2;
        // 从第一个素数开始，标记该素数的倍数为非素数
        for (int i = first; i < sqrt_N; i += prime) {
            NewMarked[i] = 1;
        }
        while (NewMarked[++index]);
        prime = index * 2 + 3;// 起始加偏移
    } while (prime * prime <= sqrt_N);
```

2.5 分块筛选，提高 Cache 命中率

对于第三个有两个优化思路，第一个是基于 cache_linesize 的优化，

另外一个是基于 cache_size 的优化。对于 windows，在任务管理器的性能中能看到相关信息，也可以通过软件 CPU-Z 来查看。

Cache 配置如图：



2.5.1 优化思路

对于大规模的数组来说 cache_line 的优化效果并不是很明显，所以下面只针对 cache 做优化，本优化方法需每个进程都各计算出前 sqrt(n) 个数中的素数，(对应去广播优化)。

由于从 cache 读取的速度远高于从内存中处理，所以 cache_size 优化的思路在于每次处理 cache 大小的数组，之前我们已经将 n 内分成大小约为 n/p 的块给每个进程处理，然后再在每个进程中将 n/p 大小块按照 cache_size 进行分块，在此之前我们需要对 cache 的大小从 byte 转化为 int，64 位系统即除以 8，以上图 windows 为例，L1，L2，L3 缓存分别可以存 48KB，0.1875MB，1.5MB 个 int，而单机（注意是单机，如果是分布式计算机理论上可以占满所有的 cache）中每个进程又将划分其中的 cache，如果对于 L3 而言如果分配 2 个进程则每个进程能够得到 0.75 MB 个 int 进程处理，而实际中由于计算机中有其他进程也会使用 cache，所以

在实际中这个数还要小。另外根据测试的时候 n 大小进行选择 cache 的级别，比如我测试的是亿级的数据，远超过 cache 的大小，所以直接对 L3 级别的 cache 进行分块，当然选择 L3 并不一定是最优策略，需要多次实证才能知道。关键在于进程内分块。

2.5.2 关键代码

```
#define CACHE_SIZE 12*1024*1024  //Cache 大小，在此为我的三级缓存 12mB
#define CACHELINE_SIZE 256   //CACHELINE 256B
//#define L1_CACHE_SIZE 1048576//L1cache 大小默认值，1M
#define MIN(a, b) ((a)<(b)?(a):(b))
new_int Block_pos_first; /*Index of first multiple(in L3 or L2 Cache) */
new_int Block_pos_last; /* Index of last multiple(in L3 or L2 Cache) */
new_int Block_low_value;/* Lowest value on this CacheBLock(in L3 or L2 Cache)
*/
new_int Block_high_value;/* Highest value on this CacheBLock(in L3 or L2 Cache
)*/
int Cache_linenum_pro = CACHE_SIZE / (CACHELINE_SIZE * p);//每个进程占有的
cacheline
int CacheBlock_size = Cache_linenum_pro * 8;//每个进程获得的用于存取 long long 的
块大小
int Block_N = CacheBlock_size - 1;
int line_need = size / CacheBlock_size;//每个进程一共需要多少块
int line_rest = size % CacheBlock_size;//多出来的 cacheline
int time_UseCache = 0;
// allocate this process 's share of the array
    marked = (char*)malloc(CacheBlock_size);
    if (marked == NULL) {
        printf("Cannot allocate enough memory \n");
        MPI_Finalize();
        exit(1);
    }
    //总计数
    count = 0;
    while (time_UseCache <= line_need) {
        //cache 更新；
        Block_pos_last.value = (time_UseCache + 1) * Block_N + MIN(time_UseCac
he + 1, line_rest) - 1 + (id * (N / p) + MIN(id, N % p));
```

```
27.         Block_pos_first.value = time_UseCache * Block_N + MIN(time_UseCache, l
    ine_rest) + (id * (N / p) + MIN(id, N % p));
28.         Block_low_value.value = Block_pos_first.value * 2 + 3;
29.         if (time_UseCache == line_need) {
30.             Block_high_value.value = high_value;
31.             Block_pos_last.value = (id + 1) * (N / p) + MIN(id + 1, N % p) - 1
    ;
32.             CacheBlock_size = ((Block_high_value.value - Block_low_value.value
    ) >> 1) + 1;
33.         }
34.         else {
35.             Block_high_value.value = (Block_pos_last.value + 1) * 2 + 1;
36.         }
37.         // 索引初始化为 0
38.         index = 0;
39.         // 从 3 开始搜寻，first 为第一个不是素数的位置
40.         prime.value = 3;
41.         // 块计数
42.         count_cacheBlock = 0;
43.         // 先假定块中的整数都是素数
44.         for (int i = 0; i < CacheBlock_size; i++) marked[i] = 0;
45.         // 在块内找素数
46.         do {
47.             /*确定该进程中素数的第一个倍数的下标 */
48.            // 如果该素数 n*n>low_value，n*(n-i)都被标记了
49.            // 即 n*n 为该进程中的第一个素数
50.            // 其下标为 n*n-low_value，并且由于数组大小减半所以除以 2
51.             if (prime.value * prime.value > Block_low_value.value) {
52.                 first.value = (prime.value * prime.value - Block_low_value.val
    ue) >> 1;
53.             }
54.             else {
55.                 // 若最小值 low_value 为该素数的倍数
56.              // 则第一个倍数为 low_value，即其下标为 0
57.                 if (!(Block_low_value.value % prime.value)) first.value = 0;
58.                 // 若最小值 low_value 不是该素数的倍数
59.                 // 但是其余数为偶数，那么第一个非素数的索引为该素数剪去求余除以 2
60.                 else if (Block_low_value.value % prime.value % 2 == 0) first.v
    alue = prime.value - ((Block_low_value.value % prime.value) >> 1);
61.                 // 若最小值 low_value 不是该素数的倍数
62.                 // 那么第一个倍数的下标为该素数减去余数的值，并且由于数组大小减半所
    以除以 2
63.                 else first.value = (prime.value - (Block_low_value.value % pri
    me.value)) >> 1;
```

```
            }
            // 从第一个素数开始，标记该素数的倍数为非素数
            for (int i = first.value; i < CacheBlock_size; i += prime.value) {
                marked[i] = 1;
            }
            // 用于找到下一素数的位置
            while (NewMarked[++index]);
            prime.value = index * 2 + 3;// 起始加偏移
        } while (prime.value * prime.value <= Block_high_value.value);
        // 统计块内计数
        for (int i = 0; i < CacheBlock_size; i++) {
            if (marked[i] == 0) {
                count_cacheBlock++;
            }
        }
        // 汇总总体计数
        count += count_cacheBlock;
        // 处理下一个块
        time_UseCache++;
    }
```

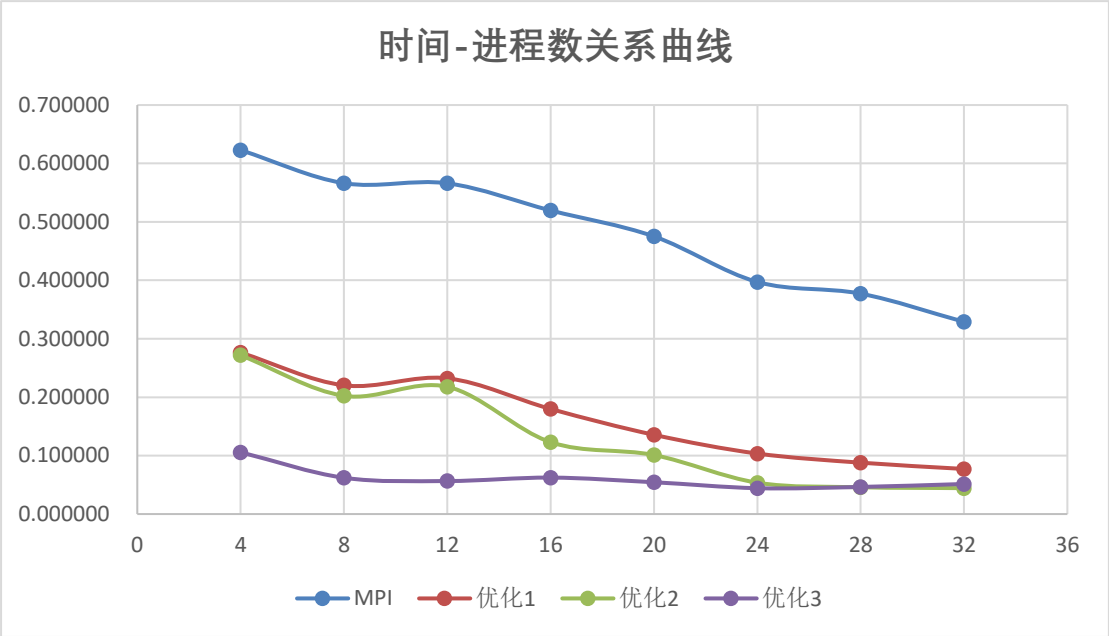
3. 实验结果分析

3.1 实验组一（1,0000,0000）

①实验组一为在 windows 环境下，在规模 1,0000,0000 下从 4 到 32 个进程分别进行 5 次测试取运行时间平均值，得出的结果如下图：

	运行时间汇总			
进程数	MPI	优化 1	优化 2	优化 3
4	0.622856	0.276238	0.271789	0.105658
8	0.566249	0.220299	0.202315	0.062081
12	0.566094	0.232320	0.217845	0.056504
16	0.519578	0.179844	0.122979	0.062379
20	0.475152	0.135575	0.100967	0.054397
24	0.397196	0.103553	0.053613	0.044237
28	0.377199	0.087921	0.045886	0.046401
32	0.329006	0.077203	0.044334	0.051442

运行时间-进 程数关系曲线如图：



横向对比可以看出通过优化一去掉偶数可以使待处理数组减半，故而时间减少一半左右；通过优化二去掉通信在单机上进程数量较少的时候优化效果不明显，随着进程数量增加，优化效果有所提升；通过优化三增加 cache 命中率可以显著提升优化效果。

②加速比

1）.绝对加速比

将最好的串行算法与并行算法相比较（这里的“最好的“不是绝对的，有时处理最快是最好的，有时得到最优解的是最好的，有时又把两者结合起来考虑，所以具体问题具体分析，本次实习都采用处理最快的为最好的）

$$S=\frac{T_{best}}{T(N)}$$

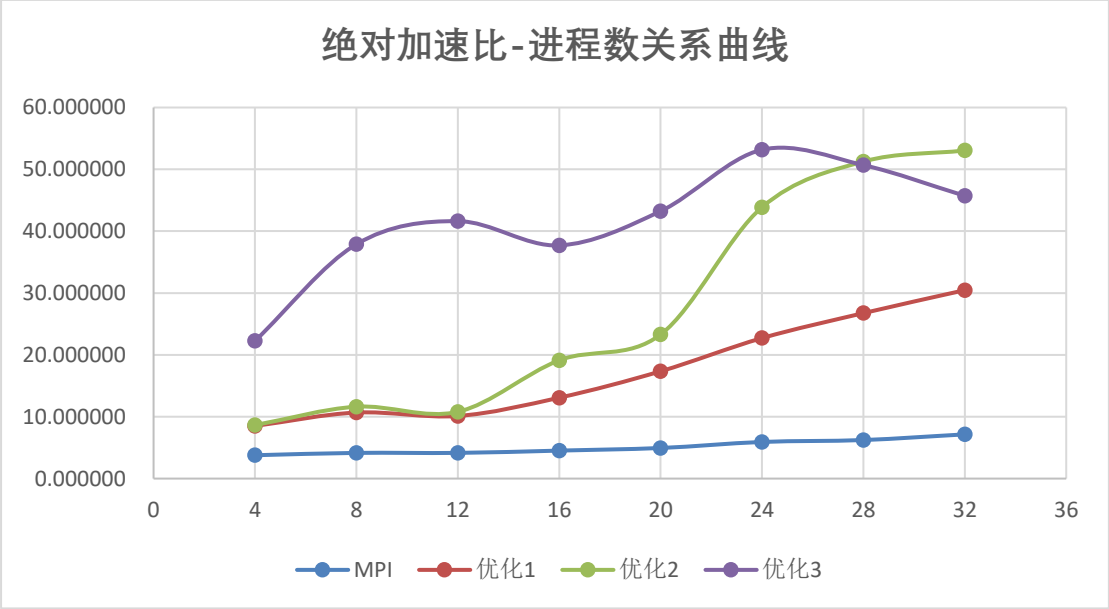
最好的串行算法的运行时间如图：

单行执行						
进程数	次数 1	次数 2	次数 3	次数 4	次数 5	平均耗时
1	2.328	2.329	2.326	2.421	2.356	2.352

绝对加速比计算结果如图：

绝对加速比汇总				
进程数	MPI	优化 1	优化 2	优化 3
4	3.776151	8.514397	8.653772	22.260459
8	4.153648	10.676399	11.625459	37.886110
12	4.154787	10.123976	10.796648	41.625372
16	4.526752	13.078030	19.125217	37.705118
20	4.949999	17.348280	23.294694	43.237838
24	5.921516	22.712919	43.869793	53.168403
28	6.235442	26.751349	51.257911	50.688126
32	7.148814	30.465295	53.051834	45.721395

绝对加速比-进程数关系曲线



可以发现通过优化 1 去除偶数，在进程数较少时，加速比表现为原始版本的两倍左右，在进程数大于 12 时，加速比继续稳定增长。

而通过优化 2 去除广播通信在进程数较少时优化效果不明显，但在进程较多的时候优化效果明显。

通过 Cache 优化对加速比提升明显，但在进程数不断增加时，加速比增长不够稳定，甚至出现了进程数增加，加速比下降的情况。

当 $S_p=p$ 时，S_p 便可以称为“线性加速比”，即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，但是在规模 1,0000,0000 的情况下除了 Cache 优化和去广播优化，其他两种优化并没有表现这种趋势，初步分析认为可能是 i/o 占据了程序的一部分。

2）. 相对加速比

同一并行算法在单节点上运行时间与在多个相同节点构成的处理机系统上的运行时间之比。这种定义侧重于描述算法和并行计算机本身的可扩展性。

$$S=\frac{T(1)}{T(N)}$$

由 S 表示形式，可以看出随着处理器数量 N 的增加，加速比 S 在增大，如果这种增长呈现线性关系，就称之为线性加速比；如果这种增长速度呈现超线性关系，就称之为超线性加速比；如果 S 增长速度逐渐呈现递减关系，就称之为病态加速比。

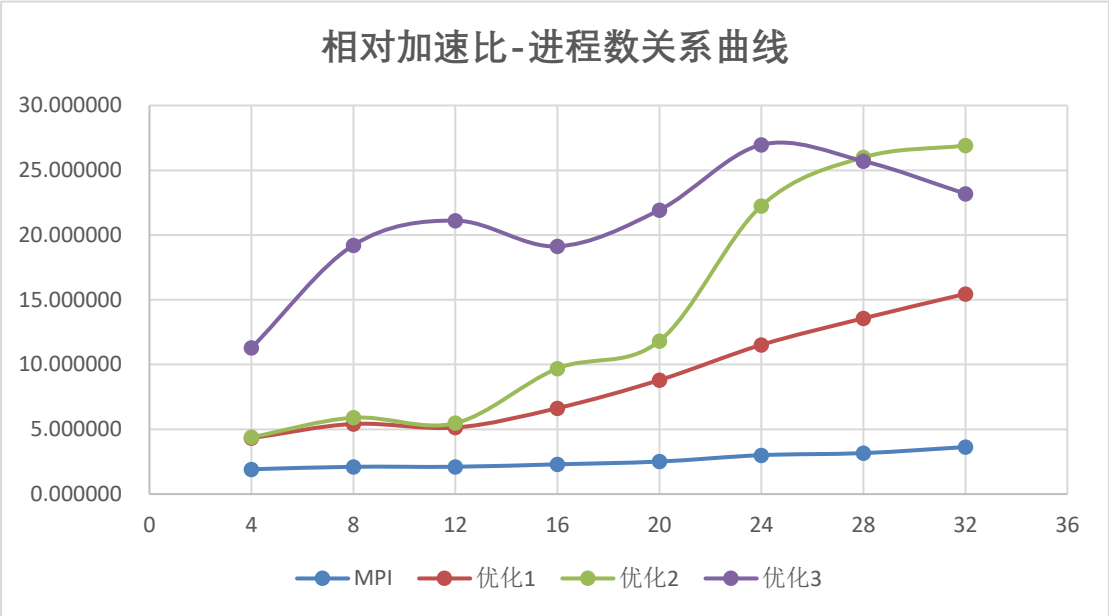
线性加速比：中间开销小，通信少，弱耦合计算

超线性加速比：当应用需要很大内存时可能出现

病态加速比：加速比递减，可能是计算量太小

注：由于本次实习中，在计算相对加速比和绝对加速比时，分母采用的数据完全一致，只是改变了分子的取值，因此曲线图形状与绝对加

速比完全一致，只是取值范围有差别。相对加速比-进程数关系曲线如图：（后续将只对绝对加速比进行分析）



不难看出，在原始 MPI 和优化 1 中，上述三种加速比情况均未出现；而在优化 2（去广播优化）中，在 12-16 进程阶段出现了线性加速比，在 20-24 进程阶段出现了超线性加速比，而在 8-12 进程阶段出现了病态加速比；在优化 3（Cache 优化）中，在 4-8、20-24 进程阶段出现了线性加速比，而在 12-16、24-32 进程阶段出现了病态加速比。

对于超线性加速比的情况，通过维基百科得知，超线性加速比有几种可能的成因，如现代计算机的存储层次不同所带来的“高速缓存效应”；具体来说，较之顺序计算，在并行计算中，不仅参与计算的处理器数量更多，不同处理器的高速缓存也集合使用。而有鉴于此，集合的缓存便足以提供计算所需的存储量，算法执行时便不必使用速度较慢的内存，因而存储器读些时间便能大幅降低，这便对实际计算产生了额外的加速效果。

③并行效率

并行效率是由加速比派生出的效率，是量度性能的指标，定义为：

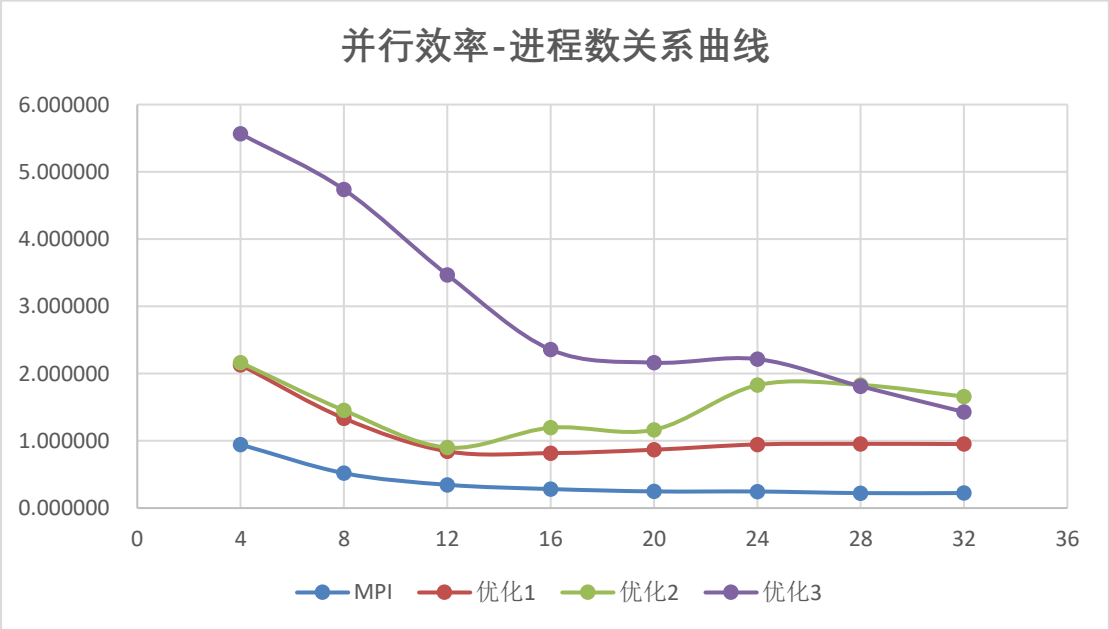
$$E_p = \frac{S_p}{p} = \frac{T_1}{pT_p}.$$

其中 p 为并行计算的核心数

各优化并行效率计算结果如下：

并行效率汇总				
进程数	MPI	优化 1	优化 2	优化 3
4	0.944038	2.128599	2.163443	5.565115
8	0.519206	1.334550	1.453182	4.735764
12	0.346232	0.843665	0.899721	3.468781
16	0.282922	0.817377	1.195326	2.356570
20	0.247500	0.867414	1.164735	2.161892
24	0.246730	0.946372	1.827908	2.215350
28	0.222694	0.955405	1.830640	1.810290
32	0.223400	0.952040	1.657870	1.428794

绘制的并行效率与执行进程数的曲线如下：



可以看到在没有使用 Cache 优化的时候，并行效率一般介于 0-2 之间，

通过 Cache 优化在进程数较少时，并行效率提升明显，但随着进程数的增加，并行效率呈现急速下降并趋于平缓。

3.2 实验组二（10,0000,0000）

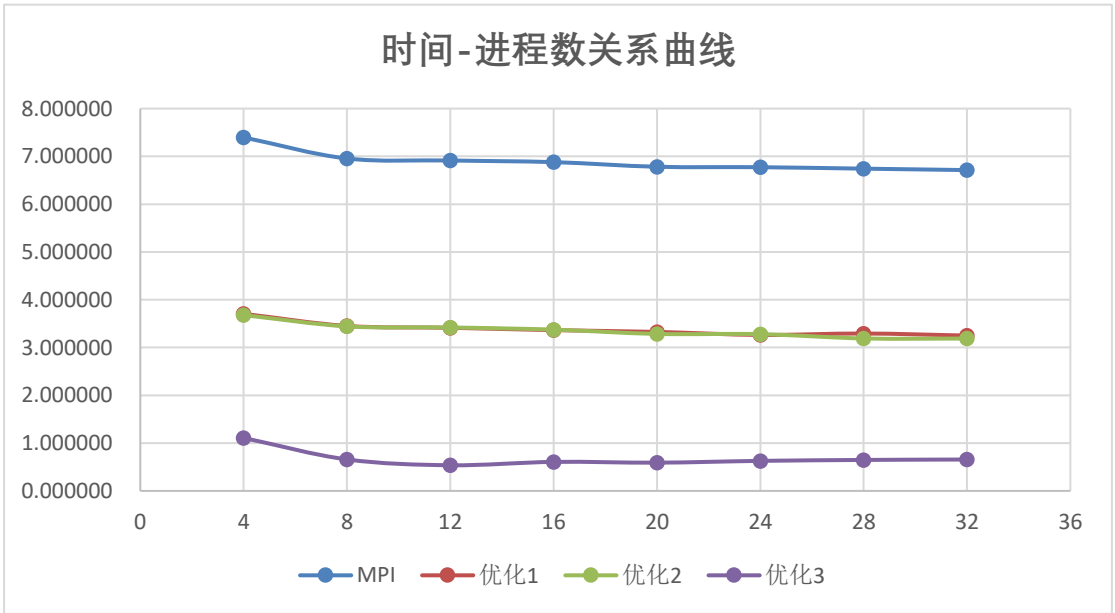
为了减少 i/o 占比，让计算占据程序更大的部分，我们将规模增加为 10,0000,0000。

①运行时间

运行时间汇总表：

	运行时间汇总			
进程数	MPI	优化 1	优化 2	优化 3
4	7.396894	3.705501	3.678613	1.108405
8	6.956320	3.452338	3.443471	0.658759
12	6.913996	3.413446	3.419979	0.537246
16	6.879428	3.364131	3.374694	0.607570
20	6.783076	3.326852	3.285012	0.591393
24	6.773436	3.264225	3.280095	0.628525
28	6.743607	3.293699	3.190327	0.648143
32	6.713557	3.250400	3.188245	0.657364

时间-进程数关系曲线：



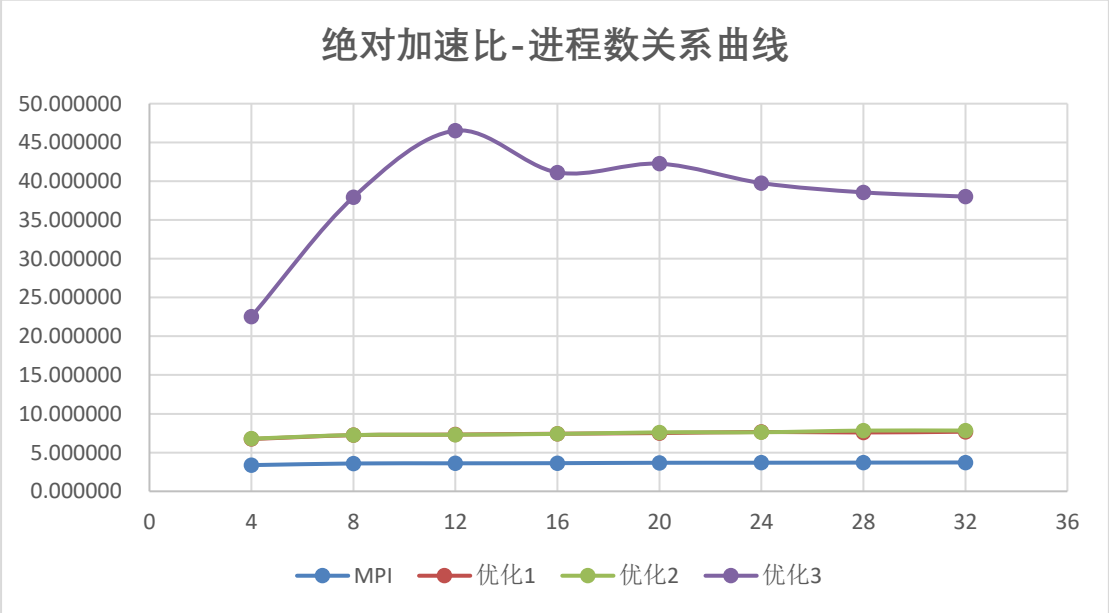
除优化 2 以外，其他优化方案均有明显的提升效果，但随着进程数的增加，各优化方案提升都不明显。

②绝对加速比

绝对加速比汇总表：

绝对加速比汇总				
进程数	MPI	优化 1	优化 2	优化 3
4	3.378580	6.744296	6.793593	22.546809
8	3.592560	7.238862	7.257503	37.936484
12	3.614552	7.321341	7.307354	46.516883
16	3.632715	7.428663	7.405412	41.132695
20	3.684317	7.511906	7.607583	42.257827
24	3.689560	7.656029	7.618987	39.761359
28	3.705880	7.587519	7.833367	38.557825
32	3.722468	7.688592	7.838482	38.017012

绝对加速比-进程数关系曲线：



在 10,0000,0000 数量级下，去除偶数优化对原始 MPI 小有提升，而去广播优化则几乎没有明显效果，前三种优化方案在进程数增加时，加速比增长都不明显。对比下，Cache 优化在前 12 核加速比按照处理器核数线性增加，而后续则出现波动，并不在呈现线性趋势，因为

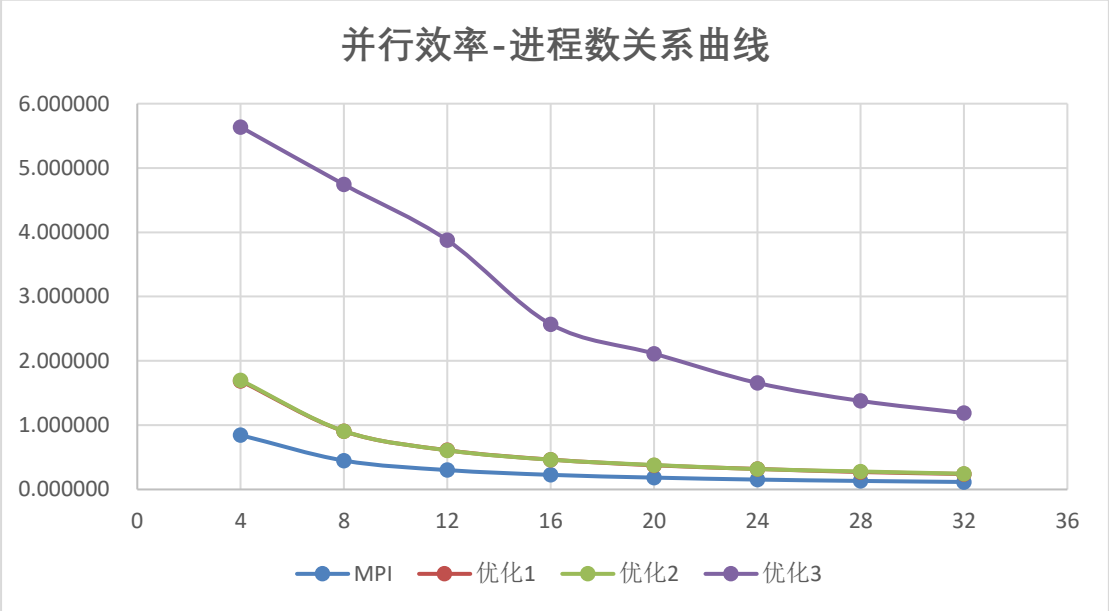
处理器有限不能同时处理多个任务，这与之前查看 windows 系统 CPU 配置的 6 核 12 处理器相符。

③并行效率

并行效率汇总表：

并行效率汇总				
进程数	MPI	优化 1	优化 2	优化 3
4	0.844645	1.686074	1.698398	5.636702
8	0.449070	0.904858	0.907188	4.742060
12	0.301213	0.610112	0.608946	3.876407
16	0.227045	0.464291	0.462838	2.570793
20	0.184216	0.375595	0.380379	2.112891
24	0.153732	0.319001	0.317458	1.656723
28	0.132353	0.270983	0.279763	1.377065
32	0.116327	0.240268	0.244953	1.188032

并行效率-进程数关系曲线：



可以看到在没有使用 Cache 优化的时候，并行效率一般介于 0-2 之间，通过 Cache 优化在进程数较少时，并行效率提升明显，但随着进程数的增加，并行效率呈现下降趋势。

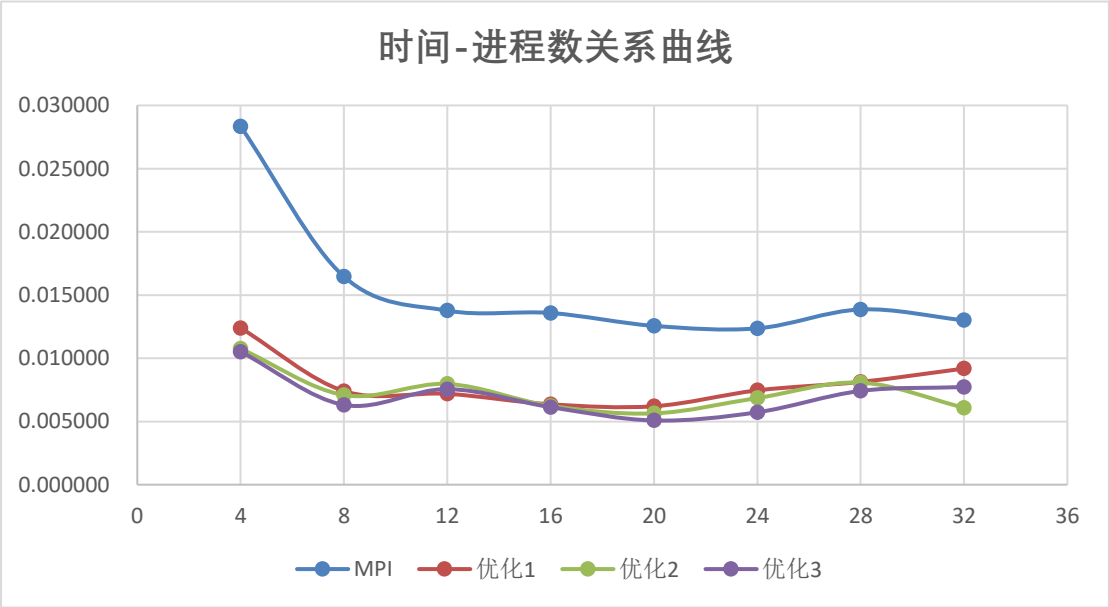
3.3 实验组三（10,000,000）

①运行时间

运行时间汇总表：

运行时间汇总				
进程数	MPI	优化 1	优化 2	优化 3
4	0.028340	0.012395	0.010752	0.010517
8	0.016458	0.007405	0.007090	0.006311
12	0.013770	0.007191	0.007968	0.007559
16	0.013583	0.006359	0.006231	0.006132
20	0.012564	0.006206	0.005640	0.005078
24	0.012364	0.007460	0.006857	0.005727
28	0.013863	0.008131	0.008076	0.007419
32	0.013006	0.009184	0.006090	0.007724

时间-进程数关系曲线：



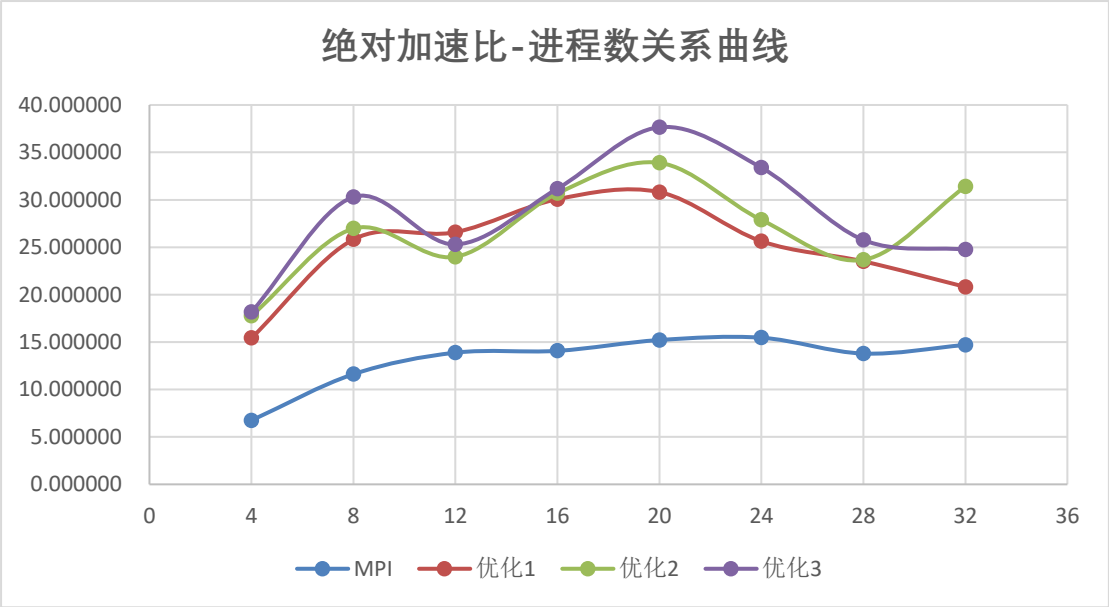
可以看到在规模较少的时候优化 1 和 2 对原始 MPI 的提升效果都不明显，同时 Cache 优化的效果也不及数量级大的情况。

②绝对加速比

绝对加速比汇总表：

绝对加速比汇总				
进程数	MPI	优化 1	优化 2	优化 3
4	6.746743	15.425077	17.782738	18.180435
8	11.617592	25.819694	26.966799	30.296308
12	13.885258	26.590271	23.994779	25.295020
16	14.076626	30.068566	30.683314	31.179675
20	15.218083	30.810880	33.898305	37.651136
24	15.463751	25.629340	27.882288	33.384551
28	13.792506	23.513786	23.675673	25.771667
32	14.700455	20.819269	31.394700	24.754013

绝对加速比-进程数关系曲线：



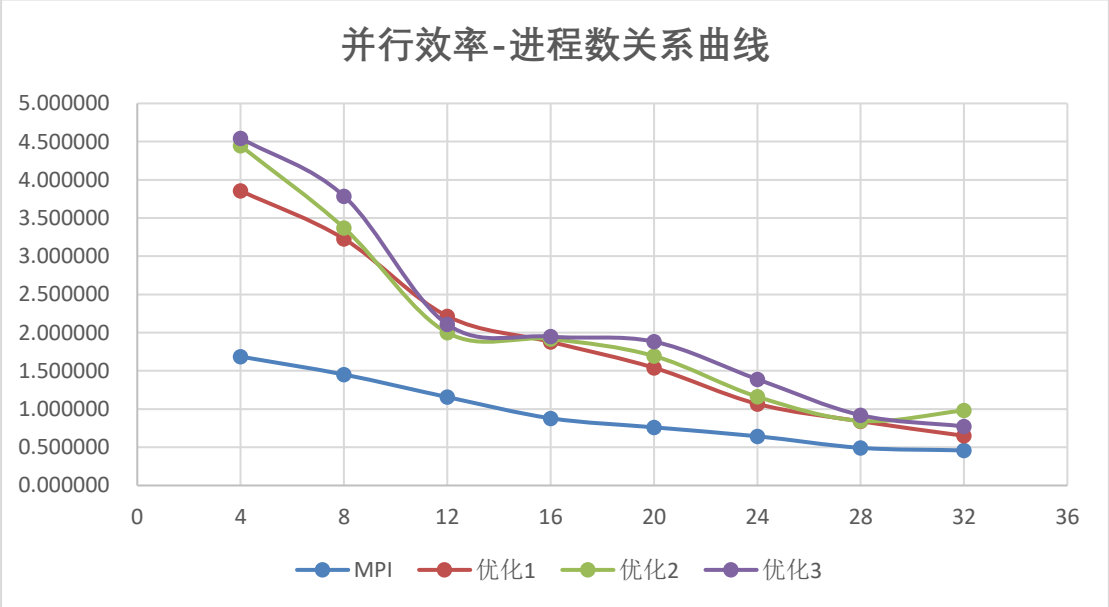
在数量级较小时，优化 1、优化 2、优化 3 相较于原始 MPI 都有明显的提升，但之间相差不大，并随着进程数的增加，绝对加速比呈现波动趋势。同时，主要的优化提升都是优化 1（去偶数优化）所致，优化 2、3 的效果不明显。

③并行效率

并行效率汇总表：

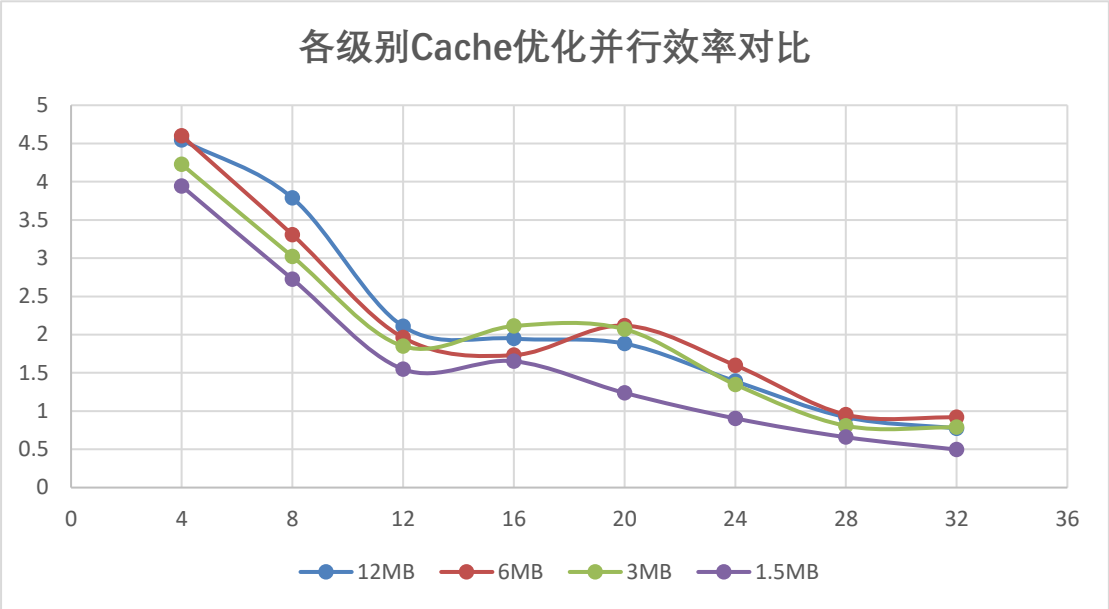
	并行效率汇总			
进程数	MPI	优化 1	优化 2	优化 3
4	1.686686	3.856269	4.445685	4.545109
8	1.452199	3.227462	3.370850	3.787039
12	1.157105	2.215856	1.999565	2.107918
16	0.879789	1.879285	1.917707	1.948730
20	0.760904	1.540544	1.694915	1.882557
24	0.644323	1.067889	1.161762	1.391023
28	0.492590	0.839778	0.845560	0.920417
32	0.459389	0.650602	0.981084	0.773563

并行效率-进程数关系曲线：

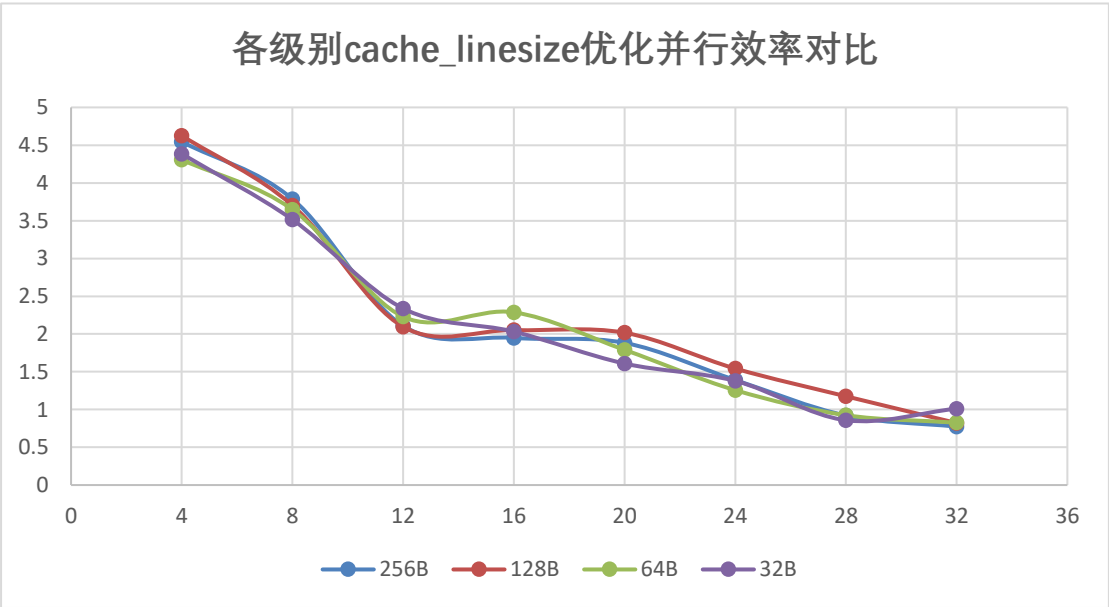


分析情况与绝对加速比分析一致。我认为导致的原因之一是数量级小，i/o 占程序的比重比计算大，另一个原因可能是 Cache 分块太大导致的，本次实习采用的均为 12MB，即 12*1024*1024，默认使用的是 L3 的 cache。因此考虑将分块减小，再分析一下结果。

分块减小后的并行效率-进程数关系曲线：



但是从结果上看，对 Cache 分块大小减小得到的并行效率并没有明显的改变。因此，我们继续在 Cache 分块大小为 12MB 不变的情况下，改变 cache_linesize 的大小进行实验。得到的并行效率-进程数关系曲线如下图：



从结果上看，在不同的 cache_linesize 大小下并行效率并没有呈现明显的差别，说明应该也不是 cache_linesize 的大小导致

的。因此可以推测是由 i/o 占比增大导致。

4.结论与改进思路

4.1 结论

(1).去偶数优化其实只是数量上线性地减少了问题规模(接近一半)，通信和广播模式没有改变，所有优化效果是线性，接近 2 倍，并行效率也相似；

(2).核心数少时广播代价并不大，随着核心数增加广播代价增大，所以在核心数很大时去广播优化就明显优于去偶数。

(3).显然存在计算代价、通信代价（或者广播代价、计算代价、通信代价），而通信（通信和广播）会随着核心数增加增大，所以 Cache 优化和去广播优化在加速计算时，如果通信代价过大，计算加速带来的收益就下降，加速比曲线存在峰；而通信代价与直接计算代价的相对比例在不同问题规模不同，使不同问题规模下最大加速比对应核心数不同；

(4).通信代价存在，导致并行效率随核心数增大而下降；

(5).问题规模溢出本人计算机性能，出现大规模加速比下降，反阿姆达尔效应。

(6).从相对加速比上不难得出，在进程数较少时，各数据级情况的程序均呈现良好的可扩展性，而在进程数较大时，更大数据级情况的程序的可扩展性会更好。

4.2 改进思路

算法并行设计的挑战：

①并行策略

②负载均衡

③并行 I/0

④通信

我们可以针对以上四个方面进行讨论

1、并行策略

本次实验中采用的是数据并行策略，因此可以考虑优化数据分配，通过资料查询，发现有另一种数据分配方法，其计算量较本次实验的方法更少。具体如下：

让进程 i 控制的第一个元素是$[\frac{in}{p}]$, 最后一个元素是$\left[\frac{(i+1)n}{p}\right]-1$，对于特定元素$j$，控制它的进程是$[\frac{p(j+1)-1}{n}]$。

2、负载均衡

负载均衡是通过调整计算在各个处理器上的分配，以充分发挥系统内处理器的计算能力，通常意味着各个处理器近似同时结束计算。负载均衡法分为两种，分别是静态负载均衡和动态负载均衡：

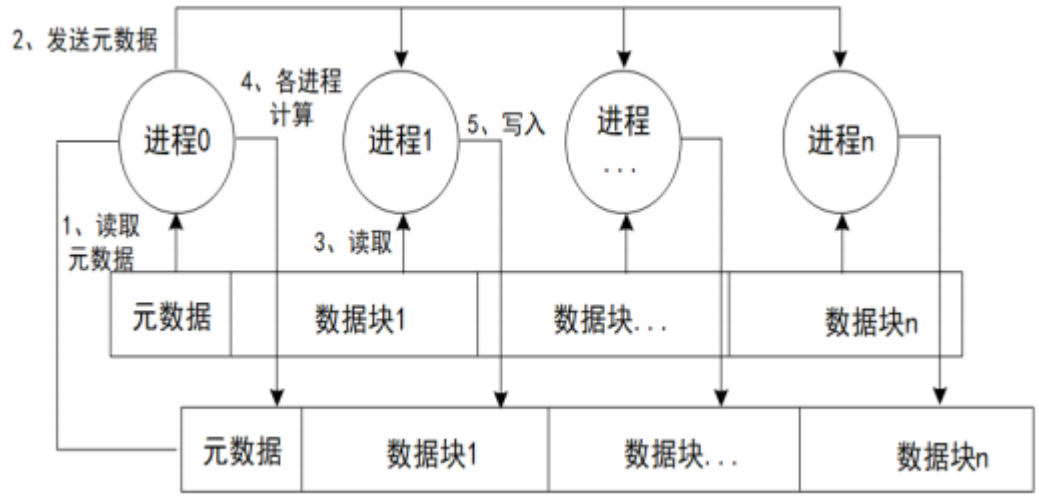
•静态负载均衡：指在程序运行前，开发人员已经将计算任务分割为多个部分并保证能够均匀地把各个部分计算分配给控制流运行

•动态负载均衡：指在程序运行过程中，重新调整任务的分布以达到负载均衡的目的

本次实验使用的是静态负载均衡，只在计算执行前进行了一次分配，因此可以考虑使用动态负载均衡优化程序。

3、并行 I/O

数据的串行 I/O 方式严重限制了并行计算性能的提升，这在我们的实验中也有体现。



解决方案：可以考虑使用 I/O 异步并行手段，即同时开展多个 I/O 操作(也被称为 overlapped I/O)。I/O 并行的美妙之处在于其伸缩性，在多核的环境下，如果可以充分利用计算资源，则通常会获得 2 倍甚至 8 倍的性能提高。

4、通信

在本次实习的优化 2（去广播优化）中，我们采用的是通过使用串行算法先求出一部分质数，然后最后再通信归总结果，这种做法避免了过多的广播通信，从结果上看，确实也是有效的。然而基于 Eratosthenes 算法本身的特点，我们实习中采用了 MPI_Barrier(MPI_Comm comm）函数：

```
1.  // MPI_Barrier 是 MPI 中的一个函数接口
2.  // 表示阻止调用直到 communicator 中所有进程完成调用
3.  MPI_Barrier(MPI_COMM_WORLD);
```

这个函数就像一道障碍，在操作中，通信子 comm 中的所有进程相互同步，即它们相互等待，直到所有的进程都执行了它们各自的 MPI_Barrier 函数，然后再各自接着开始执行后续的代码，这么做是为了保证控制执行的顺序，但一定程度上降低了程序的效能。因此是否可以考虑使用异步通信方案，并采用诸如“树形结构通信”来进一步减小通信的开支，从而达到更佳的优化结果，这也是一个可以值得尝试的方向。

5.实习总结与体会

①总的来说，在整个实习过程中，我先后经历了“选题犹豫不决”、“环境配置不好”等情况，但在这三周的课间实习当中，我在仔细学习老师下发的教程以及网络资源的帮助下对 MPI 并行计算的整个过程都有了全新的认识。

②此次实习是对《高性能地理计算》理论课程的实践，二者关系紧密，本次实习作为一个桥梁，将理论与实践有效结合，我对课上的知识也有了更加深入的理解。

③本次实习还让我对 MPI 并行编程技术有了初步的认识，在学习别人的算法优化程序的同时，我也仔细阅读并理解 MPI 的官方 API 示例，这种过程无疑是对我编程技术的提升，我也更好的掌握了 MPI 的各种优化方式在实际操作中的实现。

④当然，完成了程序的设计，对结果的分析也是至关重要，本次实习中我通过 excel 工具来进行数据的整理与绘图，这锻炼了我通过图表方式反映并分析实验结果的能力。

⑤此次实习设置为个人形式，这也锻炼了我独立解决问题、利用网络资源解决问题的能力。

吴斌文-2020302131043

2020 年 11 月 2 日