电子科技大学计算机科学与工程学院

实验报告

学　　号　2020080903009

姓　　名　李皓

（实验）课程名称　MPI 并行程序设计实验

教师　卢国明

电 子 科 技 大 学
实　验　报　告

学生姓名：李皓　　学号：2020080903009　　指导教师：卢国明

一、 实验名称: MPI 并行程序设计: 埃氏素数筛选算法并行及性能优化

二、 实验目的:

1） 使用 MPI 编程实现埃拉托斯特尼筛法并行算法。

2） 对程序进行性能分析以及调优。

三、 实验原理:

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。

这里，我们把 N 取 120 来举例说明埃拉托斯特尼筛法思想：

1） 首先将 2 到 120 写出

2） 在 2 上面画一个圆圈，然后划去 2 的其它倍数，这时划去的是除了 2 以外的其它偶数

3） 从 2 往后一个数一个数地去找，找到第一个没有被划去的数 3，将它画圈，再划去 3 的其它倍数（以斜线划去）

4） 再从 3 往后一个数一个数地去找，找到第一个没有被划去的数 5，将它画圈，再划去 5 的倍数（以交叉斜线划去）

5） 再往后继续找，可以找到 9、11、13、17、19、23、29、31、37、41、43、47，将它们分别画圈，并划去它们的倍数（可以看到，已经没有这样的数了）

6） 这时，小于或者等于 120 的各数都画上了圈或者被划去，被画圈的就是素数了。

四、 实验内容:

1） 完成 Eratosthenes 筛法实现

2） 进行并行程序的优化

-安装部署 MPI 实验环境，并调试完成基准代码，实测在不同进程规模（1，2，4，8，16）加速比，并分析原因（40 分）。

-完成优化 1，去除偶数优化，实测在不同进程规模（1，2，4，8，16）加速比，并分析原因（10 分）。

-完成优化 2，消除广播优化，实测在不同进程规模（1，2，4，8，16）加速比，并分析原因（15 分）。

-完成优化 3，cache 优化，实测在不同进程规模（1，2，4，8，16）加速比，并分析原因（10 分）。

-在完成优化 3 的基础上，可以利用课内外知识，全面优化代码性能。根据班优化 3 在目标机上实测性能，最高性能（最短执行时间）得分 25 分，最低性能得 0 分，其他按执行时间进行插值。（25 分）

五、 实验设备:

1） 操作系统：Windows 11 专业版

2） CPU: Ryzen 5-5700X

3） 编程环境：vscode

4） 执行环境：实验室服务器集群

5） Node-CPU：Intel(R) Xeon(R) Gold 5318Y x2

六、 实验步骤:

6.1 MPI 环境的配置和基本程序的运行

在 Windows 上下载安装 MPI 程序，以及最新版本的 vs stdio。随后进行环境变量的配置，以及新工程的创建。创建一个新的控制台应用，如下图所示：

图 1: 创建新的 vs 项目

随后进行环境变量和编译环境的设置，我们的素数筛选程序是一个 MPI 程序，因此运行时需要添加 msmpi 库的依赖，添加编译多线程运行的选项。完成上述配置后就可以将原始程序编译通过。

由于途中基准代码的部分变量是 int 类型，对于测试所需的 10^{10} 大小的数据可能存在溢出的风险。因此需要对原始的数据类型进行修改，如下所示：

```
int main(int argc, char* argv[])
{
    int    count;        /* Local prime count */
    double elapsed_time; /* Parallel execution time */
    int    first;        /* Index of first multiple */
    int    global_count=0; /* Global prime count */
    int    high_value;   /* Highest value on this proc */
    int    i;
    int    id=0;           /* Process ID number */
    int    index;        /* Index of current prime */
    int    low_value;    /* Lowest value on this proc */
    char* marked;       /* Portion of 2,...,'n' */
    char* marked_prime;       /* Portion of 2,...,'n' */
    int    n;            /* Sieving from 2, ..., 'n' */
    int    p;            /* Number of processes */
    int    proc0_size;   /* Size of proc 0's subarray */
    int    prime;        /* Current prime */
    int    size;         /* Elements in 'marked' */
```

图 2: 原始代码数据类型

使用 int 可能会造成溢出，导致结果计算的不正确，经过修改数据类型，以及分块计算的部分（图**??**）后即可编译运行，实测在不同进程规模（1，2，4，8，16）加速比。

```
low_value = 2 + id * (long long)(n - 1) / p;
high_value = 1 + (id + 1) * (long long)(n - 1) / p;
size = high_value - low_value + 1;
```

图 3: 经过修改后的分块计算

6.2 优化 I: 跳过偶数

跳过偶数的思想主要是在于一个朴素的思想：偶数一定不是质数。对于我们的任务而言，计算 N 以内的素数个数，有大约 $N/2$ 的数据不需要进行考虑。这一数据的减少，带来了存储数据的减少，因此需要对数据结构进行重构。在原始的算法中，用于标记素数的数组具有线性映射的关系，正如图（**??**）所示。而现在的数组和实际的素数构成了一个带有倍数关系的映射，原数据从 3 开始考虑素数的个数，这个数映射到 mask 的第 0 号。由于素数不予考虑，数字 5 映射到数组的第 1 位。

根据以上的映射关系，我们构造了从实际序数（ODD）到序号（index）的映射。为了计算的方便，使用宏定义的方式对运算进行替换：

```
#define ODD_TO_INDEX(odd) (((odd)-3)/2)
#define INDEX_TO_ODD(index) (2*(index)+3)    //对序号进行转换
```

在经过以上转换后，涉及 mask 数组的访问就使用 index 来进行，主要是进行下面的修改：

1） 对于搜寻素数的操作进行重构：

改动包括但是不限于将开始的素数由改为 3，对循环中的判断操作转化为 INDEX 的大小比较。改动后的核心代码如下：

```
    if (!id) index = 0;
    prime = 3;
    do {
        if (prime * prime > low_value)
            first = ODD_TO_INDEX( prime * prime) -ODD_TO_INDEX( low_value
                );
            // 从prime*prime 开始找，因为（prime-1）之前的因数已经找过了
            // 对于分段后的数据而言，是先前一段的内容都被找到了
        else {
            if (!(low_value % prime))
                first = 0;
            // 如果开始的数就是因子的整数倍，那么就从这个开始找
            else {
                first = prime - (low_value % prime);
                if (!((low_value + first)%2))
                    first += prime;
                    // 跳过偶数
                first /= 2;
                // 映射回index
            }
            // 如果不是，则为prime - (low_value % prime)
            // 假设 low_value 是 10 ，prime 为 3
            // first 就是 3-1= 2
            // 对应的就是mark[2]的位置，也就是 10，11，[12] 处
```

```
        }
        for (i = first; i < size; i += prime) marked[i] = 1;
        if (!id) {
            while (marked[++index]);
            // 找到下一个素数因子
            prime = INDEX_TO_ODD(index);
            // 从映射回序数
            // malloc 中的内容是从 0 开始索引，计算素数从2 开始。
        }
        if (p > 1) MPI_Bcast(&prime, 1, MPI_INT, 0, MPI_COMM_WORLD);
    } while (prime * prime <= n);
```

2） 对计数进行修正，由于计算素数个数时忽略所有二的倍数，包括 2 本身。因此在最后输出的时候需要 +1。

6.3 优化 II：取消并行通信

根据课堂的知识：计算过程的并行开销是制约并行程序发展的瓶颈。在原始的代码中，只有 0 号进程计算了素数，向其余的进程广播之后，其余的进程才能够继续进行计算。期间的通信活动使得程序的并行性能不佳。因此我们对此处进行重构，为每个进程分配一个数组各自进行素数的计算。

主要的改动在于在原始的数组 mask 之外，再开一个数组 marked_prime 来记录需要用到的素数。这个新开数组的大小只需 $\sqrt{n}$。

在一个素数标记后，后续的更新不需进行进程的通信，只需要在素数数组中寻找下一个未标记数即可。

6.4 优化 III：cache 优化

cache 优化的思路类似于进行并行化的程序设计。也就是假设原始的数据规模为 n 经过划分后，分配给 p 个进程进行计算。那么每个进程获得了 n/p 的数据。在每个进程中，若需要优化性能，那么可以根据 L3-cache 的大小进行分段，将数据分为若干段，每一段都和 cache 的大小相符合。

在进行素数计算时，完成了一段 cache 的访问后，再整体读取下一段 cache，这样可以极大优化程序的局部性，降低 cache-miss 的比例。

可以使用以下指令获取 cache 的大小：

```
getconf -a | grep CACHE
```

在实验室的机器上实验测得：

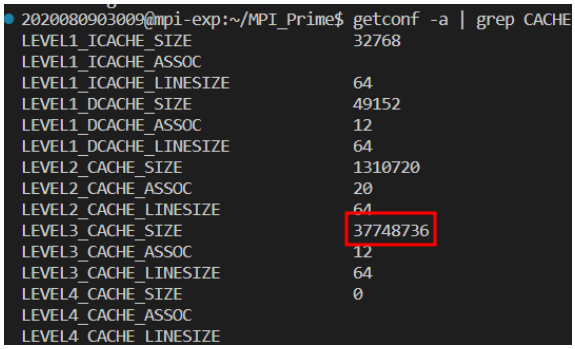

图 4: 测试缓存指令

因此我们可以使用这一输出结果来确定数组分片的大小。

6.5 优化 IV：自由优化

在上述的优化过程之外，我们还可以从以下的方法进行代码的重构和加速：

1） 对素数数组的大小进行调整，使其更为精简。

2） 考虑到计算时完成了许多序号到索引的转变，这些转变操作涉及到的乘法和除法可以换为更快的位移运算。

以下为素数程序进行计算的流程图：

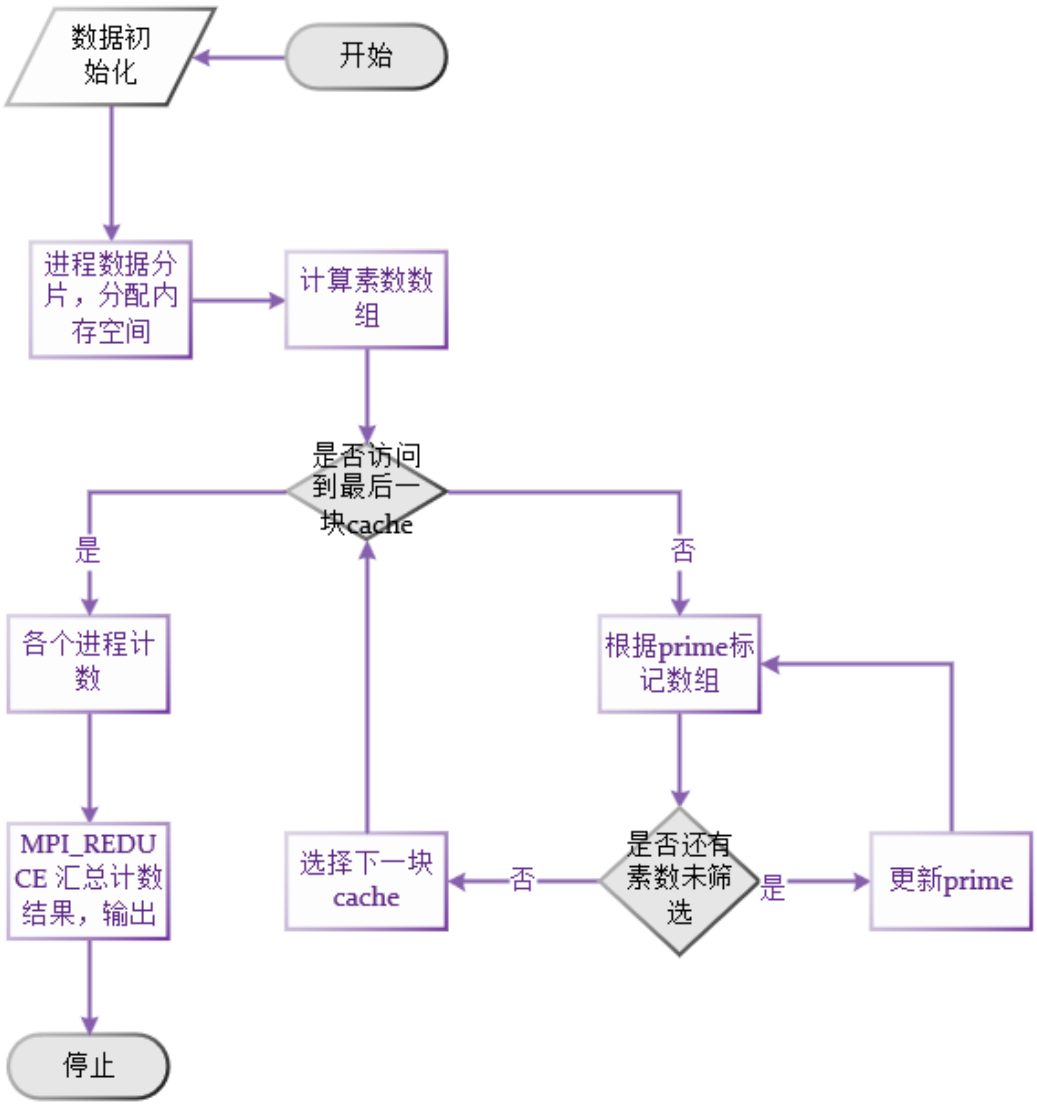

图 5: 埃氏素数筛法流程图

七、 实验结果与分析:

7.1 基准代码测试

基准代码的加速比如下所示：

表 1: 不同进程数下，不同规模下的基准程序加速比

进程数 \ n	10^3	10^4	10^5	10^6	10^7	10^8	10^9
1	**1**	**1**	**1**	**1**	**1**	**1**	**1**
2	**0.13699**	**0.50813**	**1.14061**	**1.35139**	**1.930511**	**2.434571**	**1.7950923**
4	**0.10989**	**0.30488**	**1.37152**	**2.8547**	**3.47912**	**4.687211**	**2.9950464**
8	**0.0678**	**0.29621**	**2.19395**	**6.40209**	**4.388913**	**8.180517**	**5.3209422**
16	**0.01372**	**0.13034**	**1.70304**	**10.6278**	**16.83087**	**12.67581**	**7.417538**

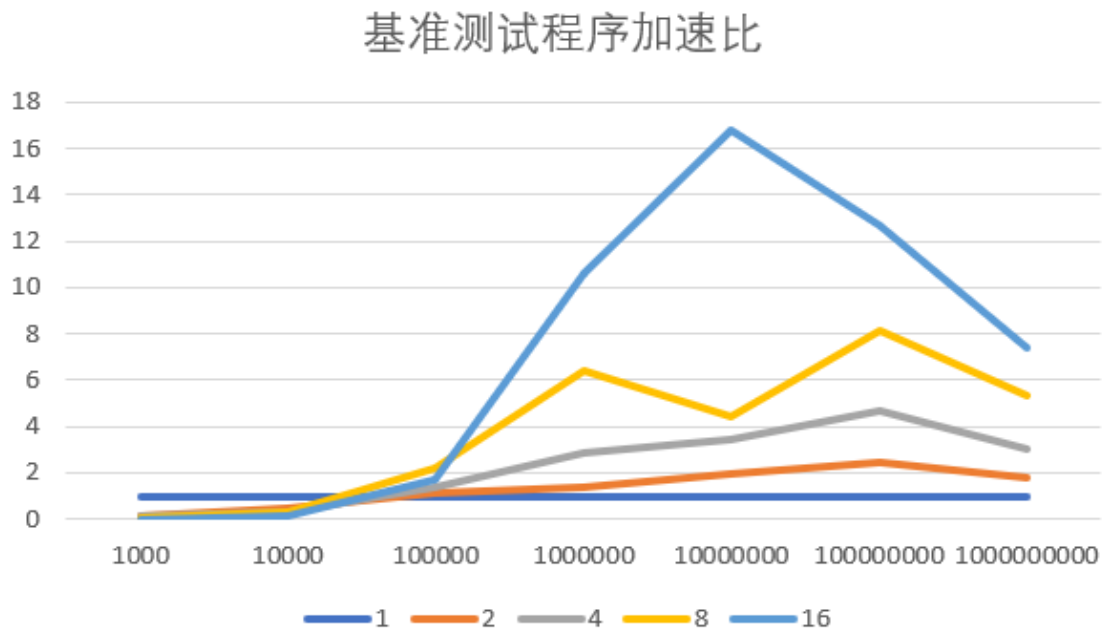

图 6: 基准程序加速比测试

分析该程序运行时的加速比可知，就整体趋势而言，加速比随着并行进程数量的增加而增加。在数字大小较小（$< 10^5$）时，由于任务本身不重，并行化开销比任务本身的串行开销要大，导致使用多线程时程序的加速比小于 1。随着数据 n 规模的扩大，导致进程之间通信的开销增加，导致加速比回落，但是仍然保持在较高的水平。

7.2 优化 I: 跳过偶数

以下为跳过偶数后的程序加速比。

表 2: 不同进程数下，不同规模下的跳过偶数后的加速比

进程数 \ n	10^3	10^4	10^5	10^6	10^7	10^8	10^9
1	**1**	**1**	**1**	**1**	**1**	**1**	**1**
2	**0.076142**	**0.313636**	**0.978784**	**1.104057**	**1.806307**	**1.9402998**	**1.90209853**
4	**0.044379**	**0.146497**	**1.797403**	**2.819246**	**3.755661**	**3.5750619**	**2.55833746**
8	**0.038961**	**0.167883**	**1.303202**	**3.660101**	**7.735794**	**6.6292152**	**5.53268197**
16	**0.029644**	**0.069347**	**0.773184**	**3.779198**	**14.45695**	**12.497346**	**6.6352114**

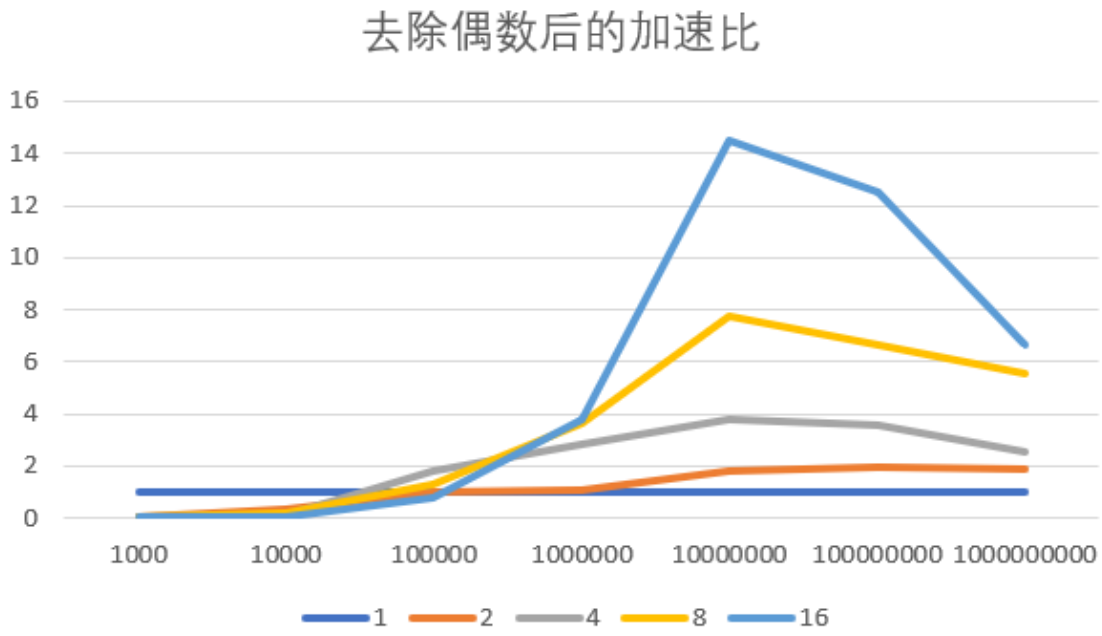

图 7: 去除偶数后的加速比测试

与上一次进行对比可以发现，加速比的提升并不明显，而整体的趋势还是保持一致。这是由于消除偶数的行为只是去除了偶数，并且增加了一部分位置映射的计算，较之于并行开销而言，提升并不明显。但是在与基准代码的纵向对比中也可以看出，在同一运行条件下，优化 I 代码的运行时间平均只有基准代码的一半，带来了数值上的绝对提升。

7.3 优化 II：取消并行通信

以下为取消并行通信后的程序加速比。

表 3: 不同进程数下，不同规模下的跳过进程通信后的加速比

进程数 \ n	10^3	10^4	10^5	10^6	10^7	10^8	10^9
1	**1**	**1**	**1**	**1**	**1**	**1**	**1**
2	**0.714286**	**0.88**	**1.005626**	**0.615108**	**1.840823**	**2.0596689**	**2.20605995**
4	**0.5**	**1.54386**	**2.344262**	**2.987882**	**3.920519**	**4.1683542**	**3.75156371**
8	**0.714286**	**1.833333**	**5.070922**	**6.341564**	**8.611069**	**8.556684**	**6.48140427**
16	**0.438596**	**0.473118**	**3.14978**	**11.5**	**13.20497**	**14.377272**	**9.16428213**

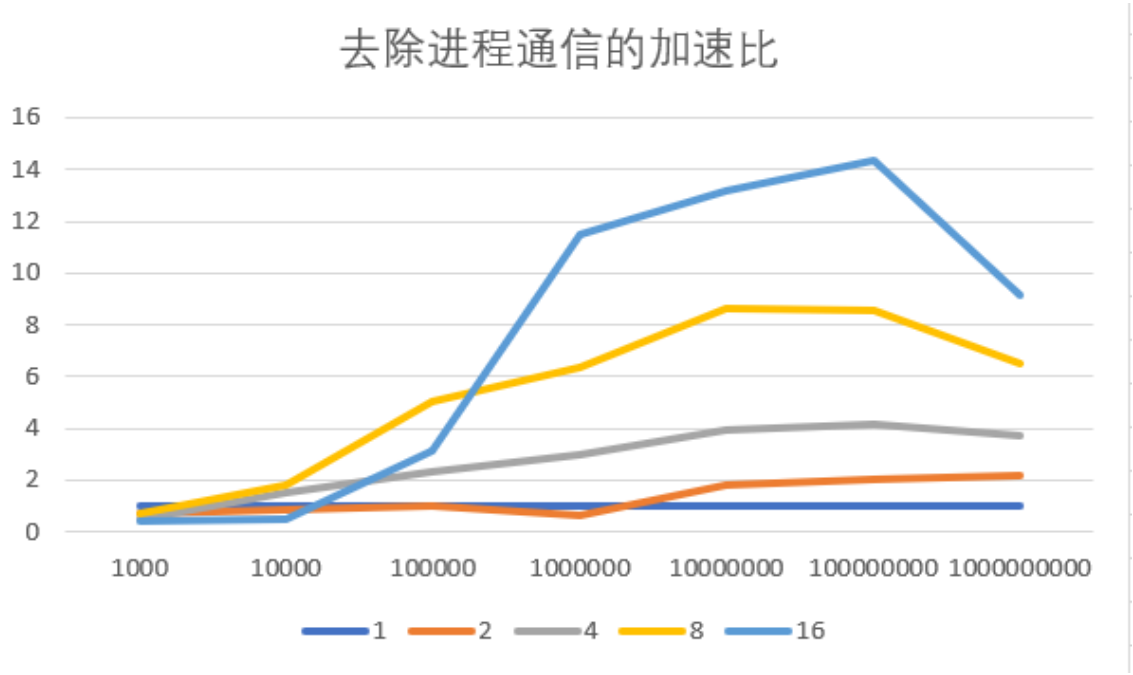

图 8: 去除进程通信后的加速比

在去除并行通信后，并行通信就只有最后汇总计数的开销，通信的次数和数据量都在下降。反映到加速比可知：在 n 十分大的情况下，加速比的损失降低，这正是优化通信后的结果。

7.4 优化 III：cache 优化

我们首先使用 perf 命令来查看优化 II 中 cache 的命中情况：

```
There are 455052511 primes less than or equal to 10000000000
SIEVE (16)  35.954861

 Performance counter stats for 'mpirun -np 16 ./Opt3 10000000000':

     5,871,368,960      cache-misses              #   60.194 % of all cache refs
     9,754,077,781      cache-references
   318,566,049,594      instructions              #    0.13  insn per cycle
 2,503,144,791,582      cycles

      36.041281328 seconds time elapsed

     539.960702000 seconds user
       4.867028000 seconds sys
```

图 9: 优化 II 的 cache-miss 情况

由图可知，大约三分之二的 cache 都处于未命中的状态。这极大制约了程序的效率，我们对程序进行优化后得到以下的加速比：

表 4: 不同进程数下，不同规模下进行 cache 优化后的加速比

进程数 \ n	10^3	10^4	10^5	10^6	10^7	10^8	10^9
1	**1**	**1**	**1**	**1**	**1**	**1**	**1**
2	**1.953856**	**1.433752**	**1.872933**	**1.936872**	**1.64537**	**2.0102914**	**2.0628349**
4	**1.982039**	**3.766319**	**3.759906**	**4.144591**	**4.085892**	**4.1677506**	**4.22854376**
8	**7.806353**	**8.564617**	**8.893048**	**9.547098**	**9.273755**	**8.9036151**	**8.62631875**
16	**5.42771**	**17.11815**	**18.04168**	**19.0874**	**6.868363**	**17.295912**	**16.7828432**

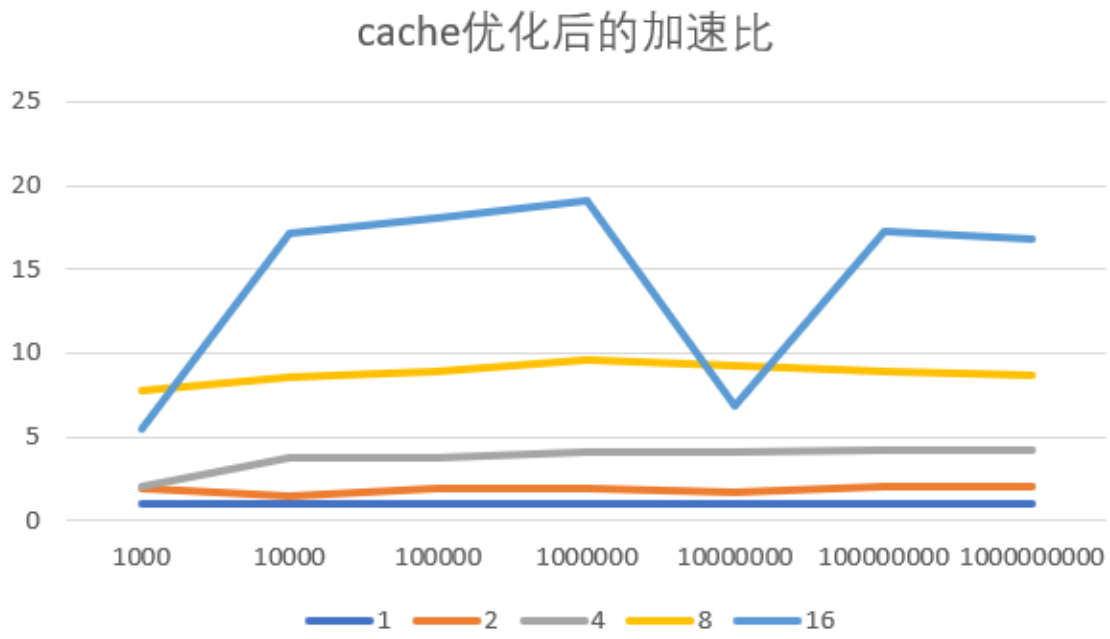

图 10: 进行 cache 优化后的加速比测试

可见优化后的程序局部性非常好，在部分 p 和 n 的组合下能够达到大于进程数量的加速比。其原因在于优化 III 限制了访问内存的范围不大于 cache, 对于计算机而言更有助于进行 cache 的刷新。由于计算机上不止一个进程在运行，cache 的占用状态也有所不同。因此 cache 带来的加速效果也不稳定，在 16 个进程的情况下某次测试进程加速比有很大的波动。

7.5 优化 IV：自由优化

进行自由优化后，程序的效率了显著提升：

表 5: 不同优化下的程序运行时间

进程数 \ 优化	base	优化 1	优化 2	优化 3
10^9	**1.654583**	**1.564225**	**1.501048**	**0.54903**
10^{10}	**23**	**14.86948**	**14.06185**	**4.878388**

在同一运行条件下，优化 3 较之于基准代码快了近 5 倍，反映出对并行程序的合理设计的加速效果。

八、 总结及心得体会:

在本次实验中，我基本掌握和复现了 MPI 并行程序设计的方法。并且通过四次程序的优化，直观认识了不同性能优化方法的效果以及特点。

经过这次实验我对并行程序设计的优化有了新的认识，在后续的实践中可以对这些技巧灵活应用。

九、 对本实验过程及方法、手段的改进建议:

在实验时需要一些代码调优的工具，如 perf 和 tau 等，但是网上似乎没有相应的资源，导致瓶颈分析困难。

报告评分：

本人签字：

指导教师签字：

十、 代码清单

1） 代码 I

```
#include "mpi.h"
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <iostream>
#define MIN(a,b)  ((a)<(b)?(a):(b))
#define ODD_TO_INDEX(odd) (((odd)-3)/2)      //Odd-to-index
#define INDEX_TO_ODD(index) (2*(index)+3)    //对序号进行转换
#define BLOCK_LOW(id,p,n) (id * (long long)(n) / p)
#define BLOCK_HIGH(id,p,n) ((id+1) * (long long)(n) / p -1)


int main(int argc, char* argv[])
{
    int    count;         /* Local prime count */
    double elapsed_time; /* Parallel execution time */
    int    first;         /* Index of first multiple */
    int    global_count=0; /* Global prime count */
    int    high_value;    /* Highest value on this proc */
    int    i;
    int    id=0;           /* Process ID number */
    int    index;         /* Index of current prime */
    int    low_value;     /* Lowest value on this proc */
    char* marked;       /* Portion of 2,...,'n' */
    int    n;             /* Sieving from 2, ..., 'n' */
    int    m;             // 将实际的编号 ODD 转化为序号 INDEX
    int    p;             /* Number of processes */
    int    proc0_size;    /* Size of proc 0's subarray */
    int    prime;         /* Current prime */
    int    size;          /* Elements in 'marked' */

    MPI_Init(&argc, &argv);

    /* Start the timer */

    MPI_Comm_rank(MPI_COMM_WORLD, &id);
    MPI_Comm_size(MPI_COMM_WORLD, &p);
    MPI_Barrier(MPI_COMM_WORLD);
    elapsed_time = -MPI_Wtime();
    if (argc != 2) {
        printf("argc is %d", argc);
        if (!id) printf("Command line: %s <m>\n", argv[0]);
        MPI_Finalize();
        exit(1);
    }

    //{//Just for debugger.if not,comment it.
    //    int temp;
    //    if (id == 0)
    //    {
    //        std::cin >> temp;
    //    }
    //    MPI_Barrier(MPI_COMM_WORLD); // All threads will wait here until you
    //    //give thread 0 an input
    //}

    n = atoi(argv[1]);
    m = ODD_TO_INDEX(n) +1;// +1 确保了不出问题
    // printf("m+n %d , %d \n",m,n);


    /* Figure out this process's share of the array, as
       well as the integers represented by the first and
       last array elements */
    // 对数组的长度进行修改
```

```
66      low_value = INDEX_TO_ODD(BLOCK_LOW(id,p,m));
67      high_value = INDEX_TO_ODD(BLOCK_HIGH(id,p,m));
68      size = (high_value - low_value)/2 + 1;
69      // printf("low_val:%d, high:%d, size:%d \n",low_value,high_value,size);
70
71
72      /* Bail out if all the primes used for sieving are
73         not all held by process 0 */
74
75      proc0_size = m / p;
76
77      if (INDEX_TO_ODD(proc0_size-1) < (int)sqrt((double)n)) {
78          if (!id) printf("Too many processes\n");
79          MPI_Finalize();
80          exit(1);
81      }
82
83      /* Allocate this process's share of the array. */
84
85      marked = (char*)malloc(size);
86
87      if (marked == NULL) {
88          printf("Cannot allocate enough memory\n");
89          MPI_Finalize();
90          exit(1);
91      }
92
93      for (i = 0; i < size; i++) marked[i] = 0;
94      if (!id) index = 0;
95      prime = 3;
96      do {
97          if (prime * prime > low_value)
98              first = ODD_TO_INDEX( prime * prime) -ODD_TO_INDEX( low_value);
99              // 从prime*prime 开始找，因为（prime-1）之前的因数已经找过了
100             // 对于分段后的数据而言，是先前一段的内容都被找到了
101         else {
102             if (!(low_value % prime))
103                 first = 0;
104             // 如果开始的数就是因子的整数倍，那么就从这个开始找
105             else {
106                 first = prime - (low_value % prime);
107                 if (!((low_value + first)%2))
108                     first += prime;
109                     // 跳过偶数
110                 first /= 2;
111                 // 映射回index
112             }
113             // 如果不是，则为prime - (low_value % prime)
114             // 假设 low_value 是 10 ，prime 为 3
115             // first 就是 3-1= 2
116             // 对应的就是mark[2]的位置，也就是 10，11，[12] 处
117         }
118         for (i = first; i < size; i += prime) marked[i] = 1;
119         if (!id) {
120             while (marked[++index]);
121             // 找到下一个素数因子
122             prime = INDEX_TO_ODD(index);
123             // 从映射回序数
124             // malloc 中的内容是从 0 开始索引，计算素数从2 开始。
125         }
126         if (p > 1) MPI_Bcast(&prime, 1, MPI_INT, 0, MPI_COMM_WORLD);
127     } while (prime * prime <= n);
128     count = 0;
129     for (i = 0; i < size; i++)
130         if (!marked[i]) count++;
131
132     // printf("%d",count);
133
134     MPI_Reduce(&count, &global_count, 1, MPI_INT, MPI_SUM,
```

```
        0, MPI_COMM_WORLD);

    /* Stop the timer */

    elapsed_time += MPI_Wtime();


    /* Print the results */

    if (!id) {
        printf("There are %d primes less than or equal to %d\n",
            global_count+1, n);
        printf("SIEVE (%d) %10.6f\n", p, elapsed_time);
    }
    MPI_Finalize();
    return 0;
}
```

2） 代码 II

```
#include "mpi.h"
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <iostream>
#define MIN(a,b)  ((a)<(b)?(a):(b))
#define ODD_TO_INDEX(odd) (((odd)-3)/2)      //Odd-to-index
#define INDEX_TO_ODD(index) (2*(index)+3)    //对序号进行转换
#define BLOCK_LOW(id,p,n) (id * (long long)(n) / p)
#define BLOCK_HIGH(id,p,n) ((id+1) * (long long)(n) / p -1)


int main(int argc, char* argv[])
{
    int    count;        /* Local prime count */
    double elapsed_time; /* Parallel execution time */
    long long    first;        /* Index of first multiple */
    int    global_count=0; /* Global prime count */
    long long    high_value;   /* Highest value on this proc */
    long long    i;
    int    id=0;           /* Process ID number */
    long long    index;        /* Index of current prime */
    long long    low_value;    /* Lowest value on this proc */
    char*   marked;       /* Portion of 2,...,'n' */
    char*   marked_prime; // 用于记录串行计算的素数结果
    long long    n;            /* Sieving from 2, ..., 'n' */
    long long    m;            // 将实际的编号 ODD 转化为序号 INDEX
    int    p;            /* Number of processes */
    int    proc0_size;   /* Size of proc 0's subarray */
    long long    prime;        /* Current prime */
    long long    size;         /* Elements in 'marked' */
    long long    prime_size;   // to record prime size.

    MPI_Init(&argc, &argv);

    /* Start the timer */

    MPI_Comm_rank(MPI_COMM_WORLD, &id);
    MPI_Comm_size(MPI_COMM_WORLD, &p);
    MPI_Barrier(MPI_COMM_WORLD);
    elapsed_time = -MPI_Wtime();
    if (argc != 2) {
        printf("argc is %d", argc);
        if (!id) printf("Command line: %s <m>\n", argv[0]);
        MPI_Finalize();
        exit(1);
    }

    //{//Just for debugger.if not,comment it.
```

```
50      //     int temp;
51      //     if (id == 0)
52      //     {
53      //         std::cin >> temp;
54      //     }
55      //     MPI_Barrier(MPI_COMM_WORLD); // All threads will wait here until you
56      //     //give thread 0 an input
57      //}
58
59      n = std::stoll(argv[1]);
60      m = ODD_TO_INDEX(n) +1;// +1 确保了不出问题
61      // printf("m: %lld ,n: %lld \n",m,n);
62
63
64      /* Figure out this process's share of the array, as
65         well as the integers represented by the first and
66         last array elements */
67      // 对数组的长度进行修改
68      low_value = INDEX_TO_ODD(BLOCK_LOW(id,p,m));
69      high_value = INDEX_TO_ODD(BLOCK_HIGH(id,p,m));
70      size = (high_value - low_value)/2 + 1;
71      prime_size = ODD_TO_INDEX(sqrt(n))+1;
72      // printf("low_val:%lld, high:%lld, size:%lld \n",low_value,high_value,
            size);
73
74
75      /* Bail out if all the primes used for sieving are
76         not all held by process 0 */
77
78      proc0_size = m / p;
79
80      if (INDEX_TO_ODD(proc0_size-1) < (int)sqrt((double)n)) {
81          if (!id) printf("Too many processes\n");
82          MPI_Finalize();
83          exit(1);
84      }
85
86      /* Allocate this process's share of the array. */
87
88      marked = (char*)malloc(size);
89      marked_prime = (char*)malloc(prime_size);
90      // printf(" primesize: %lld  \n",prime_size);
91      if (marked == NULL||marked_prime== NULL) {
92          printf("Cannot allocate enough memory\n");
93          MPI_Finalize();
94          exit(1);
95      }
96
97      for (i = 0; i < prime_size;i++) marked_prime[i] = 0;
98      for (i = 0; i < size; i++) marked[i] = 0;
99
100     index = 0;
101     prime = 3;
102     do{
103         for (i = ODD_TO_INDEX(prime * prime); i < prime_size; i += prime)
104             marked_prime[i] = 1;
105         while (marked_prime[++index]);
106         prime = INDEX_TO_ODD(index);
107     }while(prime*prime<=sqrt(n));
108
109     index = 0;
110     prime = 3;
111     do {
112         if (prime * prime > low_value)
113             first = ODD_TO_INDEX( prime * prime) -ODD_TO_INDEX( low_value);
114             // 从prime*prime 开始找, 因为 (prime-1) 之前的因数已经找过了
115             // 对于分段后的数据而言, 是先前一段的内容都被找到了
116         else {
117             if (!(low_value % prime))
118                 first = 0;
```

```
            // 如果开始的数就是因子的整数倍，那么就从这个开始找
            else {
                first = prime - (low_value % prime);
                if (!((low_value + first)%2))
                    first += prime;
                    // 跳过偶数
                first /= 2;
                // 映射回index
            }
            // 如果不是，则为prime - (low_value % prime)
            // 假设 low_value 是 10 ，prime 为 3
            // first 就是 3-1= 2
            // 对应的就是mark[2]的位置，也就是 10，11，[12] 处
        }
        for (i = first; i < size; i += prime) marked[i] = 1;

            while (marked_prime[++index]);
            // 找到下一个素数因子
            prime = INDEX_TO_ODD(index);
            // 从映射回序数
            // malloc 中的内容是从 0 开始索引，计算素数从2 开始。

    } while (prime * prime <= n);
    count = 0;
    for (i = 0; i < size; i++)
        if (!marked[i]) count++;

    MPI_Reduce(&count, &global_count, 1, MPI_INT, MPI_SUM,
        0, MPI_COMM_WORLD);

    /* Stop the timer */

    elapsed_time += MPI_Wtime();


    /* Print the results */

    if (!id) {
        printf("There are %d primes less than or equal to %lld\n",
            global_count+1, n);
        printf("SIEVE (%d) %10.6f\n", p, elapsed_time);
    }
    MPI_Finalize();
    return 0;
}
```

3） 代码 III

```
#include "mpi.h"
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <iostream>
#define MIN(a, b) ((a) < (b) ? (a) : (b))
#define ODD_TO_INDEX(odd) (((odd)-3) / 2)      // Odd-to-index
#define INDEX_TO_ODD(index) (2 * (index) + 3) // 对序号进行转换
#define BLOCK_LOW(id, p, n) (id * (long long)(n) / p)
#define BLOCK_HIGH(id, p, n) ((id + 1) * (long long)(n) / p - 1)

int main(int argc, char *argv[])
{
    int count;            /* Local prime count */
    double elapsed_time;  /* Parallel execution time */
    long long first;      /* Index of first multiple */
    int global_count = 0; /* Global prime count */
    long long high_value; /* Highest value on this proc */
    long long i;
    int id = 0;           /* Process ID number */
    long long index;      /* Index of current prime */
```

```
22      long long low_value;  /* Lowest value on this proc */
23      char *marked;         /* Portion of 2,...,'n' */
24      char *marked_prime;   // 用于记录串行计算的素数结果
25      long long n;          /* Sieving from 2, ..., 'n' */
26      long long m;          // 将实际的编号 ODD 转化为序号 INDEX
27      int p;                /* Number of processes */
28      int proc0_size;       /* Size of proc 0's subarray */
29      long long prime;      /* Current prime */
30      long long size;       /* Elements in 'marked' */
31      long long prime_size; // to record prime size.
32      int cache_size;       // cache大小，用于优化数组的信息
33      long long cache_low;        // 记录低位信息
34
35      MPI_Init(&argc, &argv);
36
37      /* Start the timer */
38
39      MPI_Comm_rank(MPI_COMM_WORLD, &id);
40      MPI_Comm_size(MPI_COMM_WORLD, &p);
41      MPI_Barrier(MPI_COMM_WORLD);
42      elapsed_time = -MPI_Wtime();
43      cache_size = 1 << 15;
44      // 设置cache的大小
45      if (argc != 2)
46      {
47          printf("argc is %d", argc);
48          if (!id)
49              printf("Command line: %s <m>\n", argv[0]);
50          MPI_Finalize();
51          exit(1);
52      }
53
54      //{//Just for debugger.if not,comment it.
55      //    int temp;
56      //    if (id == 0)
57      //    {
58      //        std::cin >> temp;
59      //    }
60      //    MPI_Barrier(MPI_COMM_WORLD); // All threads will wait here until you
61      //    //give thread 0 an input
62      //}
63
64      n = std::stoll(argv[1]);
65      m = ODD_TO_INDEX(n) + 1; // +1 确保了不出问题
66      // printf("m: %lld ,n: %lld \n",m,n);
67
68      /* Figure out this process's share of the array, as
69         well as the integers represented by the first and
70         last array elements */
71      // 对数组的长度进行修改
72      low_value = INDEX_TO_ODD(BLOCK_LOW(id, p, m));
73      high_value = INDEX_TO_ODD(BLOCK_HIGH(id, p, m));
74      size = (high_value - low_value) / 2 + 1;
75      prime_size = ODD_TO_INDEX(sqrt(n)) + 1;
76      // printf("low_val:%lld, high:%lld, size:%lld \n",low_value,high_value,
            size);
77
78      /* Bail out if all the primes used for sieving are
79         not all held by process 0 */
80
81      proc0_size = m / p;
82
83      if (INDEX_TO_ODD(proc0_size - 1) < (int)sqrt((double)n))
84      {
85          if (!id)
86              printf("Too many processes\n");
87          MPI_Finalize();
88          exit(1);
89      }
```

```
90
91      /* Allocate this process's share of the array. */
92
93      marked = (char *)malloc(size);
94      marked_prime = (char *)malloc(prime_size);
95      // printf(" primesize: %lld  \n",prime_size);
96      if (marked == NULL || marked_prime == NULL)
97      {
98          printf("Cannot allocate enough memory\n");
99          MPI_Finalize();
100         exit(1);
101     }
102
103     for (i = 0; i < prime_size; i++)
104         marked_prime[i] = 0;
105     for (i = 0; i < size; i++)
106         marked[i] = 0;
107
108     index = 0;
109     prime = 3;
110     do
111     {
112         for (i = ODD_TO_INDEX(prime * prime); i < prime_size; i += prime)
113             marked_prime[i] = 1;
114         while (marked_prime[++index])
115             ;
116         prime = INDEX_TO_ODD(index);
117     } while (prime * prime <= sqrt(n));
118
119     // 按照cache大小进行循环分段
120     for (i = 0; i < size; i += cache_size)
121     {
122         index = 0;
123         prime = 3;
124         cache_low = INDEX_TO_ODD(ODD_TO_INDEX(low_value)+i);
125         // printf("cache_low %lld\n",cache_low);
126         do
127         {
128             if (prime * prime > cache_low)
129                 first = ODD_TO_INDEX(prime * prime) - ODD_TO_INDEX(cache_low);
130             // 从prime*prime 开始找，因为（prime-1）之前的因数已经找过了
131             // 对于分段后的数据而言，是先前一段的内容都被找到了
132             else
133             {
134                 if (!(cache_low % prime))
135                     first = 0;
136                 // 如果开始的数就是因子的整数倍，那么就从这个开始找
137                 else
138                 {
139                     first = prime - (cache_low % prime);
140                     if (!((cache_low + first) % 2))
141                         first += prime;
142                     // 跳过偶数
143                     first /= 2;
144                     // 映射回index
145                 }
146                 // 如果不是，则为prime - (low_value % prime)
147                 // 假设 low_value 是 10 ，prime 为 3
148                 // first 就是 3-1= 2
149                 // 对应的就是mark[2]的位置，也就是 10，11，[12] 处
150             }
151             for (i = first; i < size; i += prime)
152                 marked[i] = 1;
153
154             while (marked_prime[++index])
155                 ;
156             // 找到下一个素数因子
157             prime = INDEX_TO_ODD(index);
158             // 从映射回序数
```

```
            // malloc 中的内容是从 0 开始索引，计算素数从2 开始。

        } while (prime * prime <= n);
    }

    count = 0;
    // printf("size:%lld",size);
    for (i = 0; i < size; i++)
        if (!marked[i])
            count++;

    MPI_Reduce(&count, &global_count, 1, MPI_INT, MPI_SUM,
               0, MPI_COMM_WORLD);

    /* Stop the timer */

    elapsed_time += MPI_Wtime();

    /* Print the results */

    if (!id)
    {
        printf("There are %d primes less than or equal to %lld\n",
               global_count + 1, n);
        printf("SIEVE (%d) %10.6f\n", p, elapsed_time);
    }
    MPI_Finalize();
    return 0;
}
```

4） 代码 IV

```
#include "mpi.h"
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <iostream>
#define MIN(a, b) ((a) < (b) ? (a) : (b))
#define ODD_TO_INDEX(odd) (((odd)-3) >>1 )      // Odd-to-index
#define INDEX_TO_ODD(index) (((index)<<1) + 3) // 对序号进行转换
#define BLOCK_LOW(id, p, n) (id * (long long)(n) / p)
#define BLOCK_HIGH(id, p, n) ((id + 1) * (long long)(n) / p - 1)
// #pragma GCC optimize(3)

int main(int argc, char *argv[])
{
    int block_count=0;
    int count=0;              /* Local prime count */
    double elapsed_time;  /* Parallel execution time */
    long long first;      /* Index of first multiple */
    int global_count = 0; /* Global prime count */
    long long high_value; /* Highest value on this proc */
    long long i;
    long long j;
    int id = 0;           /* Process ID number */
    long long index;      /* Index of current prime */
    long long low_value;  /* Lowest value on this proc */
    char *marked;         /* Portion of 2,...,'n' */
    char *marked_prime;   // 用于记录串行计算的素数结果
    long long n;          /* Sieving from 2, ..., 'n' */
    long long m;          // 将实际的编号 ODD 转化为序号 INDEX
    int p;                /* Number of processes */
    int proc0_size;       /* Size of proc 0's subarray */
    long long prime;      /* Current prime */
    long long size;       /* Elements in 'marked' */
    long long prime_size; // to record prime size.
    int cache_size;       // cache大小，用于优化数组的信息
    long long cache_low;        // 记录低位信息
    long long s_n; // sqrt n
```

```
38      long long ss_n; // sqrt sqrt n
39
40
41      MPI_Init(&argc, &argv);
42
43      /* Start the timer */
44
45      MPI_Comm_rank(MPI_COMM_WORLD, &id);
46      MPI_Comm_size(MPI_COMM_WORLD, &p);
47      MPI_Barrier(MPI_COMM_WORLD);
48      elapsed_time = -MPI_Wtime();
49      cache_size = 37748736/4;
50      // cache_size = 50;
51      int block_size = cache_size/p;
52      // 设置cache的大小
53      if (argc != 2)
54      {
55          printf("argc is %d", argc);
56          if (!id)
57              printf("Command line: %s <m>\n", argv[0]);
58          MPI_Finalize();
59          exit(1);
60      }
61
62      //{//Just for debugger.if not,comment it.
63      //    int temp;
64      //    if (id == 0)
65      //    {
66      //        std::cin >> temp;
67      //    }
68      //    MPI_Barrier(MPI_COMM_WORLD); // All threads will wait here until you
69      //    //give thread 0 an input
70      //}
71
72      n = std::stoll(argv[1]);
73      m = ODD_TO_INDEX(n) + 1; // +1 确保了不出问题
74      // printf("m: %lld ,n: %lld \n",m,n);
75
76      /* Figure out this process's share of the array, as
77         well as the integers represented by the first and
78         last array elements */
79      // 对数组的长度进行修改
80      s_n = sqrt(n);
81      ss_n = sqrt(s_n);
82      low_value = INDEX_TO_ODD(BLOCK_LOW(id, p, m));
83      high_value = INDEX_TO_ODD(BLOCK_HIGH(id, p, m));
84      size = (high_value - low_value) / 2 + 1;
85      prime_size = ODD_TO_INDEX(s_n) + 1;
86      // printf("low_val:%lld, high:%lld, size:%lld \n",low_value,high_value,
            size);
87
88      /* Bail out if all the primes used for sieving are
89         not all held by process 0 */
90
91      proc0_size = m / p;
92
93      if (INDEX_TO_ODD(proc0_size - 1) < s_n)
94      {
95          if (!id)
96              printf("Too many processes\n");
97          MPI_Finalize();
98          exit(1);
99      }
100
101     /* Allocate this process's share of the array. */
102
103     marked = (char *)malloc(block_size);
104     marked_prime = (char *)malloc(prime_size);
105     // printf(" primesize: %lld  \n",prime_size);
106     if (marked == NULL || marked_prime == NULL)
```

```
107     {
108         printf("Cannot allocate enough memory\n");
109         MPI_Finalize();
110         exit(1);
111     }
112 
113     for (i = 0; i < prime_size; i++)
114         marked_prime[i] = 0;
115 
116     index = 0;
117     prime = 3;
118     do
119     {
120         for (i = ODD_TO_INDEX(prime * prime); i < prime_size; i += prime)
121             marked_prime[i] = 1;
122         while (marked_prime[++index])
123             ;
124         prime = INDEX_TO_ODD(index);
125     } while (prime * prime <= ss_n);
126 
127     // 按照cache大小进行循环分段
128     i = 0;
129     int block_num = size/block_size;
130     int block_tail = size % block_size;
131     // printf("ID:%d: size:%lld  bn:%d,bt:%d\n\n",id,size,block_num,block_tail
        );
132     long long cache_high;
133     for (int block_id = 0; block_id<=block_num; block_id ++)
134     {
135         // printf("i:%lld,block_size:%d,size:%lld\n",i,block_size,size);
136         // i 是 块的 id
137         for (j = 0; j < block_size; j++)
138         marked[j] = 0;
139         index = 0;
140         prime = 3;
141 
142         cache_low = INDEX_TO_ODD(ODD_TO_INDEX(low_value)+block_id*block_size);
143         if (block_id==block_num){
144             cache_high = INDEX_TO_ODD((ODD_TO_INDEX(low_value)+block_id*
                block_size+block_tail-1));
145         }  else{
146             cache_high = INDEX_TO_ODD((ODD_TO_INDEX(low_value)+(block_id+1)*
                block_size-1));
147         }
148 
149 
150         do
151         {
152             if (prime * prime > cache_low)
153                 first = ODD_TO_INDEX(prime * prime) - ODD_TO_INDEX(cache_low);
154             // 从prime*prime 开始找，因为（prime-1）之前的因数已经找过了
155             // 对于分段后的数据而言，是先前一段的内容都被找到了
156             else
157             {
158                 if (!(cache_low % prime))
159                     first = 0;
160                 // 如果开始的数就是因子的整数倍，那么就从这个开始找
161                 else
162                 {
163                     first = prime - (cache_low % prime);
164                     if (!((cache_low + first) & 1))
165                         first += prime;
166                     // 跳过偶数
167                     first /= 2;
168                     // 映射回index
169                 }
170                 // 如果不是，则为prime - (low_value % prime)
171                 // 假设 low_value 是 10 ，prime 为 3
172                 // first 就是 3-1= 2
```

```
173                 // 对应的就是mark[2]的位置，也就是 10，11，[12] 处
174             }
175             for (j = first; j < block_size; j += prime)
176                 marked[j] = 1;
177 
178             while (marked_prime[++index])
179                 ;
180             // 找到下一个素数因子
181             prime = INDEX_TO_ODD(index);
182             // 从映射回序数
183             // malloc 中的内容是从 0 开始索引，计算素数从2 开始。
184 
185         } while (prime * prime <= cache_high);
186         // printf("sz:%d diff:%lld \n",block_size,cache_high-cache_low);
187         // printf("sz:%d indexdiff:%lld \n",block_size,ODD_TO_INDEX(
                cache_high)-ODD_TO_INDEX(cache_low));
188 
189         for (j = 0; j < (cache_high-cache_low)/2+1; j++)
190         if (!marked[j])
191             block_count++;
192 
193         count+=block_count;
194         // printf("ID:%d: blockID:%d low_val:%lldlow:%lld,high:%lldblock_count
                :%d\n",id,block_id,low_value,cache_low,cache_high,block_count);
195         block_count=0;
196         // printf("%dcouter:%d\n",id,count);
197     }
198 
199 
200     MPI_Reduce(&count, &global_count, 1, MPI_INT, MPI_SUM,
201                0, MPI_COMM_WORLD);
202 
203     /* Stop the timer */
204 
205     elapsed_time += MPI_Wtime();
206 
207     /* Print the results */
208 
209     if (!id)
210     {
211         printf("There are %d primes less than or equal to %lld\n",
212                global_count + 1, n);
213         printf("SIEVE (%d) %10.6f\n", p, elapsed_time);
214     }
215     MPI_Finalize();
216     return 0;
217 }
```