电 子 科 技 大 学

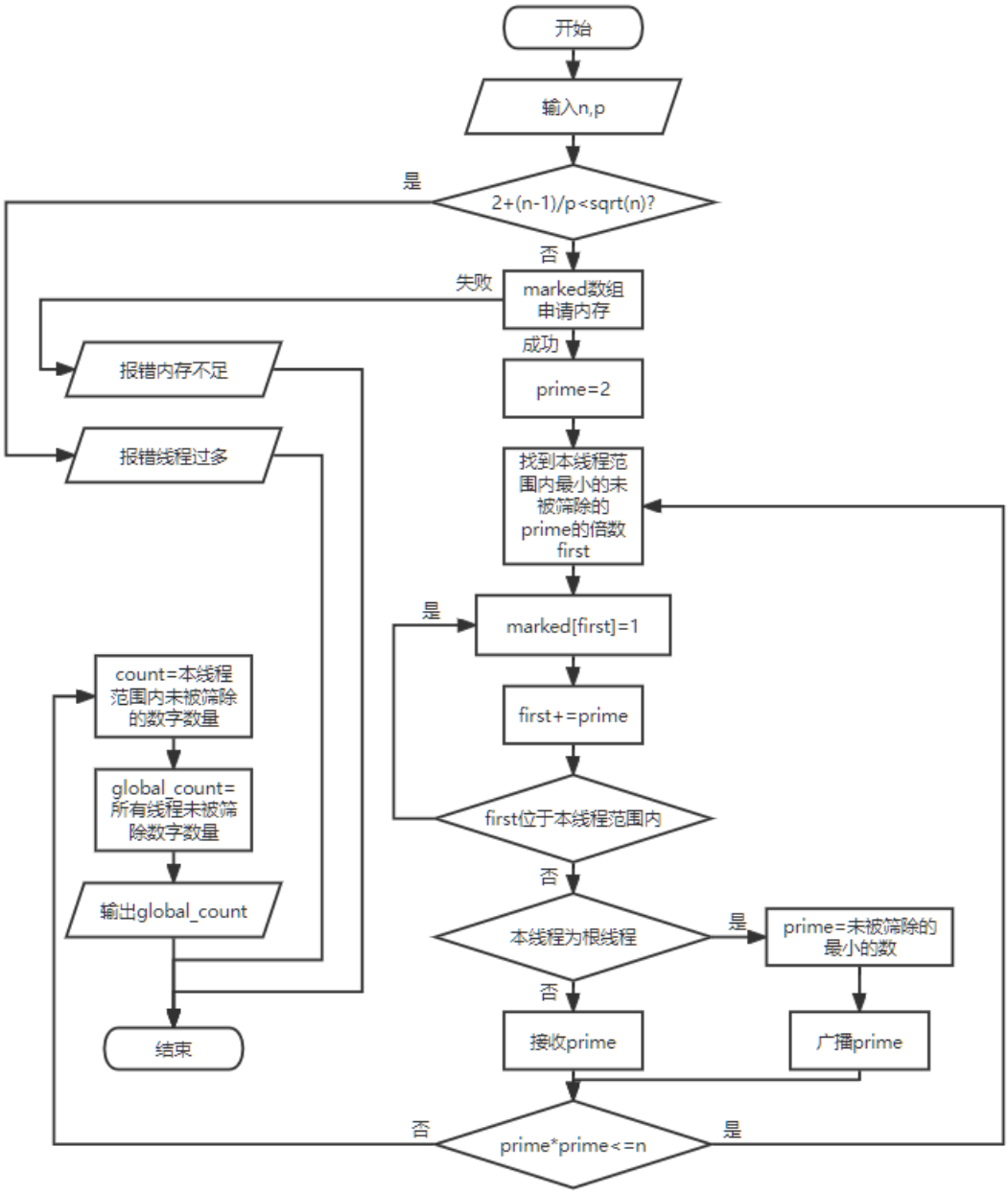
实 验 报 告

学生姓名：张镕麒	学 号：2019081301026

一、实验室名称：品学楼 A107

二、实验项目名称：埃拉托斯特尼素数筛选算法并行及性能优化

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。

本实验使用 C++中的 MPI 库实现并行运算，将埃氏筛中可并行运算的筛除部分进行并行化，并通过一系列优化手段降低程序的时空复杂度。

四、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。

五、实验内容：

1. Debug：对基础代码进行调试，找到错误。
2. 实现去偶优化：利用大于 2 的质数都是奇数的原理，去除掉所有偶数。
3. 实现去广播优化：让每个进程筛出所有必要的素数，减少广播通信时间。
4. 实现 cache 优化：进行循环重构等各种手段，提高 cache 利用率，提高效率。

六、实验器材（设备、元器件）：

服务器：浪潮 5280M4

操作系统：Linux

CPU： E5-2660 v4 2 颗

内存：64G

七、实验步骤及操作(逐句列出各个优化的实现步骤)：

1. Debug：

```
[std2019081301026@cu01 ~]$ mpic++ -o base base.cpp
base.cpp: In function 'int main(int, char**)':
base.cpp:36:14: error: 'exit' was not declared in this scope
       exit (1);
              ^
base.cpp:39:20: error: 'atoi' was not declared in this scope
    n = atoi(argv[1]);
                    ^
base.cpp:57:14: error: 'exit' was not declared in this scope
       exit (1);
              ^
base.cpp:62:34: error: 'malloc' was not declared in this scope
    marked = (char *) malloc (size);
                                  ^
base.cpp:67:14: error: 'exit' was not declared in this scope
       exit (1);
```

编译程序后报错如上所示，原因为代码中调用了 stdlib.h 头文件中的函数而并没有引用该头文件。

```
[std2019081301026@cu01 ~]$ mpic++ -o base base.cpp
[std2019081301026@cu01 ~]$ mpiexec -np 4 ./base 1000000
There are 78498 primes less than or equal to 1000000
SIEVE (4)   0.004831
```

加入该头文件后程序成功编译运行。

线程过多时，下面语句存在 int 溢出，需要将类型转化为 long long 类型。

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

在线程数为 1 时，该语句不会执行，global_count 不会被加，此时输出素数个数恒定为 1，需要删除 if(p>1)保证该语句被正确执行。

```
if (p > 1) MPI_Reduce (&count, &global_count, 1, MPI_INT, MPI_SUM,
   0, MPI_COMM_WORLD);
```

2. 实现去偶优化：

时间优化（对应 optimzer1）：

由于去除了偶数，原处理区间需要由[2,n]变为[3,n]，所以需要增加一位偏移量。

```
low_value = 3 + id*(n-2)/p;
high_value = 2 + (id+1)*(n-2)/p;
```

由于 2 已经被筛除，需要改变初始条件，以减少一次循环次数。

```
prime = 3;
```

first 需要找到大于等于 prime*prime 的在本线程范围内最小的 prime 的奇数倍数，故按 base 的方式找到的 first，若其对应 prime 倍数为偶数，根据非 2 偶数是合数原理已经被去除掉，则需要 first+=pirme 使得 first 成为合理的要被筛出的素数。

```
do {
    if (prime * prime > low_value)
        first = prime * prime - low_value;
    else {
        if (!(low_value % prime)) first = ((low_value / prime) & 1) ? 0 : prime;
        else first = prime - (low_value % prime) + (((low_value / prime) & 1) ? prime : 0);
    }
```

筛除过程中，需要从 first 开始，每次增加 2*prime 来找到下一个要被筛除的数。

```
for (i = first; i < size; i += prime << 1) marked[i] = 1;
```

根进程在找最小素数时，需要跳过所有偶数，此外由于区间范围改变，对应 prime 也需要偏移 1。

```
if (!id) {
    do {
        index += 2;
    } while (marked[index]);
    prime = index + 3;
}
```

找本进程所有未被筛除数字数量时，需要先找到第一个范围内的偶数，并跳过所有实为偶数的数字。

```
for (i = (low_value & 1) ? 0 : 1; i < size; i += 2)
    if (!marked[i]) count++;
```

输出过程中，由于没有计入 2 为素数，所以需要将 global_count 加 1。

```
printf ("There are %d primes less than or equal to %d\n",
        global_count+1, n);
```

该优化中，我们改进了所有可能的循环枚举过程，跳过了每一位实为偶数的地址空间，但我们还可以进一步设计空间映射，避免声明偶数部分对应的区间，从而减少内存空间开销，同时也因为 malloc 函数参数 size 的减少从而进一步降低时间复杂度。

空间优化（对应 optimzer2）：

我们进一步将[3,n]区间内的所有奇数映射到一段连续地址空间，具体实现为：

```
#define ID(x) (((x)-2)/2)
#define RID(x) (2*(x)+3)
```

其中 ID 实现逻辑地址到物理地址的转换，RID 实现物理地址到逻辑地址的转换。

声明 marked 的空间大小由本进程范围决定，id_low_value 和 id_high_value 表示进程对应的物理地址范围。此外由于去除了偶数，我们需要调整逻辑地址范围以使得左右端点为奇数以减少不必要的运算。

```
low_value = 3 + id*(n-2)/p;
low_value+= !(low_value & 1);
high_value = 2 + (id+1)*(n-2)/p;
high_value-= !(high_value & 1);
id_low_value = ID(low_value);
id_high_value = ID(high_value);
size = id_high_value - id_low_value + 1;
```

在筛除过程中，我们使用物理地址来控制循环，i 对应被筛除素数的物理地址对应值。逻辑地址中需要增加 2*prime，但由于物理地址相对逻辑地址对应为除 2 关系，所以最终为 i+=prime。又由于[id_low_value,id_high_value]在程序实现时再次需要映射到[0,size]（声明的 marked 数组需要从 0 开始），所以 i-id_low_value 表示逻辑值经过两次映射的真实物理地址。

```
for (i = ID(first+low_value); i <= id_high_value; i += prime) marked[i-id_low_value] = 1;
```

根进程寻找最小质数时同理，index 本需要加 2，但由于物理地址相对逻辑地址为除 2 关系，所以最终为++index。对应 prime 真值也需要从真实物理地址映射到相对物理地址，再从相对物理地址映射到逻辑地址。

```
if (!id) {
    while (marked[++index]);
    prime = RID(index + id_low_value);
}
```

寻找本进程未被筛除数字数量时同理，i 本需要加 2，但由于物理地址相对逻辑地址为除 2 关系，所以最终为 i++。

```
for (i = 0; i < size; i++)if (!marked[i]) count++;
```

在该优化中，我们设计了二重映射，一层为乘除 2 的逻辑与相对地址之间的映射，一层为加减 id_low_value 的相对绝对地址之间的映射。从而实现了对偶数数字地址空间的完全省去，减半了内存开销与申请内存对应的时间开销，性能得以再次优化。

3. 实现去广播优化（对应 optimzer3）：

在该优化中，我们需要先让每个线程筛除根号 n 以内的所有素数。使用 flag[]数组记录[0,sqrt(n)]内的筛除过程，此时需要在申请内存报错程序段中加入 flag 的判定。low_value 和 high_value0 为该数组对应的范围。

```
marked = (bool *) malloc (size);
flag = (bool *)malloc (size0);
if (marked == NULL || flag == NULL) {
    printf ("Cannot allocate enough memory\n");
    MPI_Finalize();
    exit (1);
}
```

```
low_value0 = 0;
high_value0 = (int) sqrt((double) n);
high_value0 -= !(high_value0 & 1);
size0 = high_value0 - low_value0 + 1;
```

此时素数不再由根进程提供，不需要根进程对应范围包括所有根号 n 内的素数，则可删去下面程序段。

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

我们使用 C++中的 vector 不定长数组记录根号内的素数 prime，需要引用头文件<vector>并使用命名空间 std。

```
#include <vector>
using namespace std;
```

```
vector<int> pri;
```

根号 n 内素数筛选代码段：

```
for(i = 2; i <= high_value0; i++)
{
    if(flag[i]) continue;
    pri.push_back(i);
    for(j = i * i; j < high_value0; j += i) flag[j] = 1;
}
```

后续筛选[low_value,high_value]中素数时，prime 不再由根进程广播提供，而是由 pri[]中获得。

```
for(int j = 1; j < pri.size(); j++)
{
    prime = pri[j];
```

至此，程序已经不需要依靠根进程实现 prime 的获得，减少了广播的开销。

4. 实现其他优化（对应 optimzer4）：

寄存器优化：

对常用变量（如枚举变量）加上 register 关键字，加快该变量的读取。

```
register int    i,j,k;          /* Iterative variable */
```

cache 优化：

使用 lscpu 指令查看 cache 大小，预估循环重构的范围。

```
Architecture:          x86_64
CPU op-mode(s):        32-bit, 64-bit
Byte Order:            Little Endian
CPU(s):                28
On-line CPU(s) list:   0-27
Thread(s) per core:    1
Core(s) per socket:    14
Socket(s):             2
NUMA node(s):          1
Vendor ID:             GenuineIntel
CPU family:            6
Model:                 79
Model name:            Intel(R) Xeon(R) CPU E5-2660 v4 @ 2.00GHz
Stepping:              1
CPU MHz:               2399.921
BogoMIPS:              3996.26
Virtualization:        VT-x
L1d cache:             32K
L1i cache:             32K
L2 cache:              256K
L3 cache:              35840K
NUMA node1 CPU(s):     0-27
```

将[id_low_value,id_high_value]划分为多段，使得该大小 sizep 符合 cache 大小，每次选取一段使用所有 prime 对该段内进行筛选，从而使得这部分 marked 数组长时间停留在 cache 区内，提高 cache 的利用率。

每部分的相对物理地址左端点为 L，r 为绝对地址右端点（其中 r 为重复利用变量，可提高 cache 利用率）。tL 为 L 的真值。将 r 的计算提出循环外是避免 MIN 的多次调用，先计算后使用仅使用一次 MIN 函数。first 的计算参数需要从 low_value 改变为 tL，即找到大于等于 tL 的 prime 的最小的奇数倍数。

```
for(L = ID(low_value); L <= id_high_value; L += sizep)
{
    tL = RID(L);
    for(j = 1; j < top; j++)
    {
        prime = pri[j];
        p2 = prime * prime;
        int first = 0;
        if (p2 > tL)
            first = p2 - tL;
        else {
            r = tL % prime, d = tL / prime;
            if (!r) first = (1 - (d & 1)) * prime;
            else first = prime - r + (d & 1) * prime;
        }
        r = MIN(L + sizep - 1, id_high_value) - id_low_value;
        for (i = ID(first + tL) - id_low_value; i <= r; i += prime) marked[i] = 1;
    }
}
```

运算优化：

将乘除 2 的运算改为左右移运算，位运算比十进制运算速度更快。

```
#define ID(x) (((x)-2)>>1)
#define RID(x) (((x)<<1)+3)
```

减少 if 的使用，使用运算语句实现条件语句。

```
low_value += !(low_value & 1);
```

在求 first 过程中，将除法和取余单独提取出来命名为寄存器变量（下图中为 p2,r,d），减少同一运算的反复执行。例如下图首先需要判断取余，如果余数非零，需要再次用到该余数，若未单独命为 r，则需要两次运算。

```
p2 = prime * prime;
int first = 0;
if (p2 > tL)
    first = p2 - tL;
else {
    r = tL % prime, d = tL / prime;
    if (!r) first = (1 - (d & 1)) * prime;
    else first = prime - r + (d & 1) * prime;
}
```

对自筛素数过程，使用去偶优化。但在实际测试中效果不明显，详见第八节。

```
/*delete even numbers
for(i = id_low_value0; i < id_high_value0; i++)
{
    if(marked[i])continue;
    pri[top++] = prime = RID(i);
    for(j = prime * prime; j < high_value0; j += prime << 1)marked[ID(j)] = 1;
}
*/
```

其他优化：

使用 int 数组代替 vector，降低时间复杂度的常数，并预先计算程序最大可接受的 n 的大小(1e9)，预先分配 pri[]的大小。

```
int             pri[33000];   /* Save prime numbers */
int             top;          /* Size of pri[] */
```

flag 和 marked 均为 bool 型变量，可以取两者较大者为 marked 数组的大小，两个数组使用过程存在先后，互不干扰，故可以省去 flag 的内存开销。

```
size = MAX(high_value0, id_high_value - id_low_value + 1);
/* Allocate this process's share of the array. */
marked = (bool *) malloc (size);
```

使用 string.h 头文件 memset 代替 for 循环对 marked 数组进行初始化，效率更高。

```
memset(marked, 0, size0);
```

使用 if 语句判断代替三目运算符，同时使用 inline 关键字，效率更高。

```
inline int MAX(int a,int b)
{
    if(a<b)return b;
    return a;
}
```

八、实验数据及结果分析：

1. 对自筛素数过程，使用去偶优化。但由于服务器抖动问题，该优化并不明显，优化时间之占很小一部分，而且由于坐标映射带来的额外运算与优化部分较为接近，故实际未采用该优化。（optimzer4 为未加优化，optimzer5 为加自筛去偶优化，打印出的时间为截止到自筛结束所用时间。由于服务器抖动，优化效果并不明显，故最终未使用该优化）

```
[std2019081301026@cu01 ~]$ mpiexec -np 4 ./optimzer5 1000000000
  0.047795
  0.048622
  0.048705
  0.048923
There are 50847534 primes less than or equal to 1000000000
SIEVE (4)   0.757989
```

```
[std2019081301026@cu01 ~]$ mpiexec -np 4 ./optimzer5 1000000000
  0.053039
  0.053471
  0.053627
  0.054185
There are 50847534 primes less than or equal to 1000000000
SIEVE (4)   0.761685
[std2019081301026@cu01 ~]$ mpiexec -np 4 ./optimzer4 1000000000
  0.047704
  0.047964
  0.048267
  0.048821
There are 50847534 primes less than or equal to 1000000000
SIEVE (4)   0.761297
```

2. 经多次测试后找到较优值，该值与其他值相较下速率较快。存在服务期波动因素影响，测试结果仅供参考，第三个参数为 sizep 大小，除 210000 外其他数字波动均较大，但只有 210000 基本对应最快速度。分析可知，该程序使用到了 cpu 的 L2 级 cache 存储 marked 数组，其他部分用于存储程序的其他变量。

```
sizep = 210000;
```

```
[std2019081301026@cu01 ~]$ mpiexec -np 4 ./optimzer4 1000000000 210000
There are 50847534 primes less than or equal to 1000000000
SIEVE (4)   0.738335
[std2019081301026@cu01 ~]$ mpiexec -np 4 ./optimzer4 1000000000 180000
There are 50847534 primes less than or equal to 1000000000
SIEVE (4)   0.751329
[std2019081301026@cu01 ~]$ mpiexec -np 4 ./optimzer4 1000000000 150000
There are 50847534 primes less than or equal to 1000000000
SIEVE (4)   0.752633
[std2019081301026@cu01 ~]$ mpiexec -np 4 ./optimzer4 1000000000 120000
There are 50847534 primes less than or equal to 1000000000
SIEVE (4)   0.741952
[std2019081301026@cu01 ~]$ mpiexec -np 4 ./optimzer4 1000000000 80000
There are 50847534 primes less than or equal to 1000000000
SIEVE (4)   0.753805
[std2019081301026@cu01 ~]$ mpiexec -np 4 ./optimzer4 1000000000 50000
There are 50847534 primes less than or equal to 1000000000
SIEVE (4)   0.786610
[std2019081301026@cu01 ~]$ mpiexec -np 4 ./optimzer4 1000000000 30000
There are 50847534 primes less than or equal to 1000000000
SIEVE (4)   0.761981
[std2019081301026@cu01 ~]$ mpiexec -np 4 ./optimzer4 1000000000 20000
There are 50847534 primes less than or equal to 1000000000
SIEVE (4)   0.847674
[std2019081301026@cu01 ~]$ mpiexec -np 4 ./optimzer4 1000000000 10000
There are 50847534 primes less than or equal to 1000000000
SIEVE (4)   1.243750
[std2019081301026@cu01 ~]$ mpiexec -np 4 ./optimzer4 1000000000 240000
There are 50847534 primes less than or equal to 1000000000
SIEVE (4)   0.738039
[std2019081301026@cu01 ~]$ mpiexec -np 4 ./optimzer4 1000000000 270000
There are 50847534 primes less than or equal to 1000000000
SIEVE (4)   0.756799
[std2019081301026@cu01 ~]$ mpiexec -np 4 ./optimzer4 1000000000 300000
There are 50847534 primes less than or equal to 1000000000
SIEVE (4)   0.791767
```

3. 基准代码运行结果（1,2,4,8,16 核）：

```
[std2019081301026@cu01 ~]$ mpiexec -np 1 ./base 10000000
There are 664579 primes less than or equal to 10000000
SIEVE (1)   0.226998
[std2019081301026@cu01 ~]$ mpiexec -np 2 ./base 10000000
There are 664579 primes less than or equal to 10000000
SIEVE (2)   0.119665
[std2019081301026@cu01 ~]$ mpiexec -np 4 ./base 10000000
There are 664579 primes less than or equal to 10000000
SIEVE (4)   0.060556
[std2019081301026@cu01 ~]$ mpiexec -np 8 ./base 10000000
There are 664579 primes less than or equal to 10000000
SIEVE (8)   0.030470
[std2019081301026@cu01 ~]$ mpiexec -np 16 ./base 10000000
There are 664579 primes less than or equal to 10000000
SIEVE (16)   0.015712
```

4. 优化代码一运行结果（1,2,4,8,16 核）：

```
[std2019081301026@cu01 ~]$ mpiexec -np 1 ./optimzer1 10000000
^[[AThere are 664579 primes less than or equal to 10000000
SIEVE (1)   0.146463
[std2019081301026@cu01 ~]$ mpiexec -np 2 ./optimzer1 10000000
^[[AThere are 664579 primes less than or equal to 10000000
SIEVE (2)   0.076696
[std2019081301026@cu01 ~]$ mpiexec -np 4 ./optimzer1 10000000
^[[AThere are 664579 primes less than or equal to 10000000
SIEVE (4)   0.039021
[std2019081301026@cu01 ~]$ mpiexec -np 8 ./optimzer1 10000000
^[[AThere are 664579 primes less than or equal to 10000000
SIEVE (8)   0.019679
[std2019081301026@cu01 ~]$ mpiexec -np 16 ./optimzer1 10000000
There are 664579 primes less than or equal to 10000000
SIEVE (16)   0.010530
```

5. 优化代码二运行结果（1,2,4,8,16 核）：

```
[std2019081301026@cu01 ~]$ mpiexec -np 1 ./optimzer2 10000000
There are 664579 primes less than or equal to 10000000
SIEVE (1)   0.097500
[std2019081301026@cu01 ~]$ mpiexec -np 2 ./optimzer2 10000000
There are 664579 primes less than or equal to 10000000
SIEVE (2)   0.052094
[std2019081301026@cu01 ~]$ mpiexec -np 4 ./optimzer2 10000000
There are 664579 primes less than or equal to 10000000
SIEVE (4)   0.025950
[std2019081301026@cu01 ~]$ mpiexec -np 8 ./optimzer2 10000000
^[[AThere are 664579 primes less than or equal to 10000000
SIEVE (8)   0.012865
[std2019081301026@cu01 ~]$ mpiexec -np 16 ./optimzer2 10000000
There are 664579 primes less than or equal to 10000000
SIEVE (16)   0.006892
```

6. 优化代码三运行结果（1,2,4,8,16 核）：

```
[std2019081301026@cu01 ~]$ mpiexec -np 1 ./optimzer3 10000000
^[[AThere are 663910 primes less than or equal to 10000000
SIEVE (1)   0.095941
[std2019081301026@cu01 ~]$ mpiexec -np 2 ./optimzer3 10000000
There are 663546 primes less than or equal to 10000000
SIEVE (2)   0.050287
[std2019081301026@cu01 ~]$ mpiexec -np 4 ./optimzer3 10000000
There are 662799 primes less than or equal to 10000000
SIEVE (4)   0.024548
[std2019081301026@cu01 ~]$ mpiexec -np 8 ./optimzer3 10000000
^[[AThere are 661391 primes less than or equal to 10000000
SIEVE (8)   0.012096
[std2019081301026@cu01 ~]$ mpiexec -np 16 ./optimzer3 10000000
There are 658515 primes less than or equal to 10000000
SIEVE (16)   0.005164
```

7. 优化代码四运行结果（1,2,4,8,16 核）：

```
[std2019081301026@cu01 ~]$ mpiexec -np 1 ./optimzer4 10000000
There are 664579 primes less than or equal to 10000000
SIEVE (1)   0.028109
[std2019081301026@cu01 ~]$ mpiexec -np 2 ./optimzer4 10000000
There are 664579 primes less than or equal to 10000000
SIEVE (2)   0.015003
[std2019081301026@cu01 ~]$ mpiexec -np 4 ./optimzer4 10000000
There are 664579 primes less than or equal to 10000000
SIEVE (4)   0.007444
[std2019081301026@cu01 ~]$ mpiexec -np 8 ./optimzer4 10000000
There are 664579 primes less than or equal to 10000000
SIEVE (8)   0.003984
[std2019081301026@cu01 ~]$ mpiexec -np 16 ./optimzer4 10000000
There are 664579 primes less than or equal to 10000000
SIEVE (16)   0.001994
```

此外，仅有优化代码四支持 n=1e9 的输入参数，其他程序均报错显示内存不足，说明该程序内存使用效率**最高**。

九、实验结论：

1. 基准代码加速比：1.00/1.90/3.75/7.45/14.45，该组代码最为接近线性加速，说明原基准程序并行化程度最高。
2. 优化代码一加速比：1.00/1.91/3.75/7.44/13.91，该组代码加速比比基准程序低，与优化代码二较为接近，因为二者优化手段接近，只有映射过程有区别。
3. 优化代码二加速比：1.00/1.87/3.76/7.58/14.15，该组代码加速比比基准程序低，与优化代码三较为接近，因为二者优化手段接近，只有映射过程有区别。
4. 优化代码三加速比：1.00/1.91/3.91/7.93/18.57，该组代码加速比为所有代码最高，其中 16 核的加速比为超线性加速，猜测为单核运行时受服务器波动影响导致速度变慢。
5. 优化代码四加速比：1.00/1.87/3.78/7.06/14.10，该组代码加速比为所有代码最低，说明优化代码四并行化程度最低。
6. **综上分析，时间优化越多的代码并行程度越低**。

十、总结及心得体会：

1. 所有程序的加速比均与核数较为接近，说明成功地使用 MPI 实现了代码的并行化。
2. 复习了 Linux 的使用，深刻体验到了跨系统开发的过程。
3. 体验了服务器在计算上的高性能，同时也因为多人共享计算资源而导致效率波动而影响后续设计。

十一、对本实验过程及方法、手段的改进建议：

可以对服务器的使用时间进行划分，使得计算资源不存在大范围被占用的情况。因为服务器的波动对实验的进行很大，经常是今天做完的优化，明天再来看反而更慢了。经常导致优化版本回溯，浪费很多时间。

报告评分：

指导教师签字：