

电子科技大学

实验报告

学生姓名:	学号:
一、实验室名称: 主楼 A2-412	
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	
<p>三、实验原理:</p> <p>1. 埃拉托斯特尼筛法:</p> <p>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$2 \sim N$的各数写在纸上:</p> <p>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。</p> <p>这里,我们把N取120来举例说明埃拉托斯特尼筛法思想:</p> <ol style="list-style-type: none">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$7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47 \dots$将它们分别画圈,并划去它们的倍数。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。 <p>埃氏筛伪代码:</p> <ol style="list-style-type: none">1) 创建无标记的自然数列表 $2, 3, \dots, n$2) $k \leftarrow 2$	

3) 重复进行

(a) 标记 k^2 到 n 之间 k 的所有倍数

(b) 将最小的未标记的且大于 k 的数赋值给 k ,

直到 $k^2 > n$

4) 未标记的数就是素数

2. 块分配:

将需要筛选素数的范围根据并行进程数量进行划分, 这相较交错式(循环式)的划分方式能够平衡负载、容易标记倍数, 但是实现方式相对复杂。

实现细节

1) 用于筛分的最大素数是 \sqrt{n}

2) 第一个进程有 n/p 下取整个元素

3) if $p < \sqrt{n}$, 则第一个进程负责的范围有所有的素数

4) 第一进程总是广播下一个筛分素数

5) 不需要还原步骤

3. MPI 编程

MPI 是一个信息传递应用程序接口, 包括协议和语义说明, 他们指明其如何在各种实现中发挥其特性。MPI 的目标是高性能, 大规模性, 和可移植性。MPI 在今天仍为高性能计算的主要模型。

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. MPI 编程实验环境的搭建，及远程服务器的连接
2. 埃拉托斯特尼并行筛法的实现
3. 代码的优化
 - 1) 去掉偶数
 - 2) 消除广播
 - 3) cache 优化

六、实验器材（设备、元器件）：

目标机器配置：

浪潮 5280M4

CPU： E5-2660 v4 2 颗

内存： 256G

LEVEL2_CACHE_LINESIZE: 64 bytes

LEVEL3_CACHE_SIZE: 37748736 bytes

本机机器配置：

CPU： Intel(R) Core(TM) i5-10210U CPU @ 1.60GHz

内存： 16 GB DDR4

操作系统： Windows11

软件： VSCODE、REMOTE-SSH

七、实验步骤及操作：

1. 使用远程连接工具以 SSH 模式登录集群。

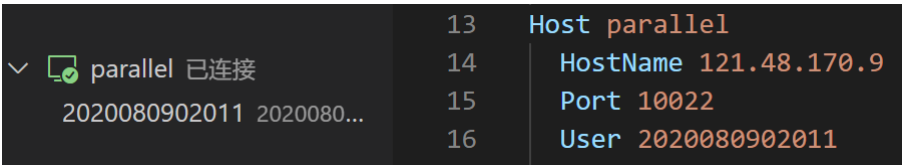


图 1 VSCODE 远程连接配置

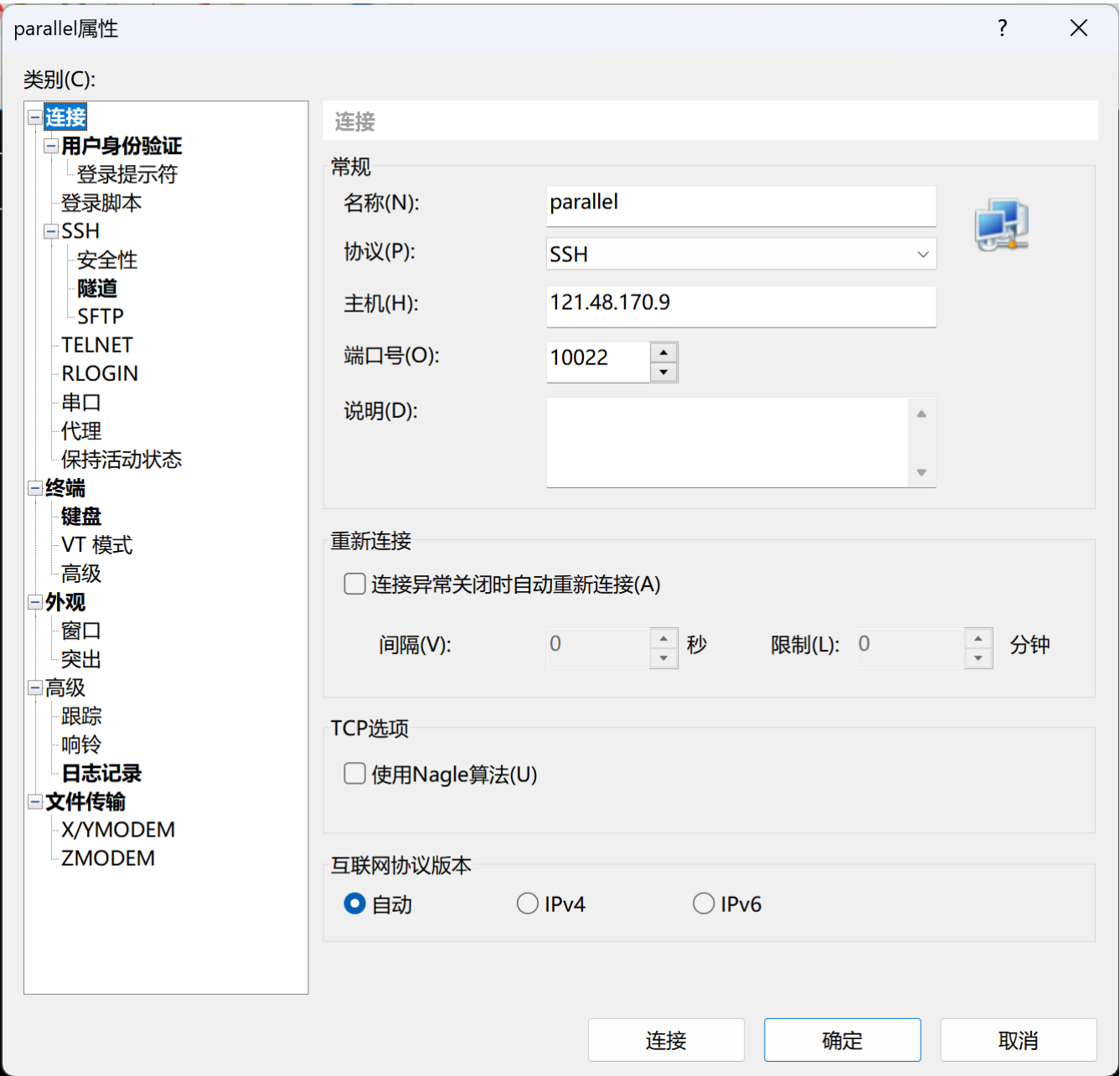


图 2 Xshell 远程连接配置

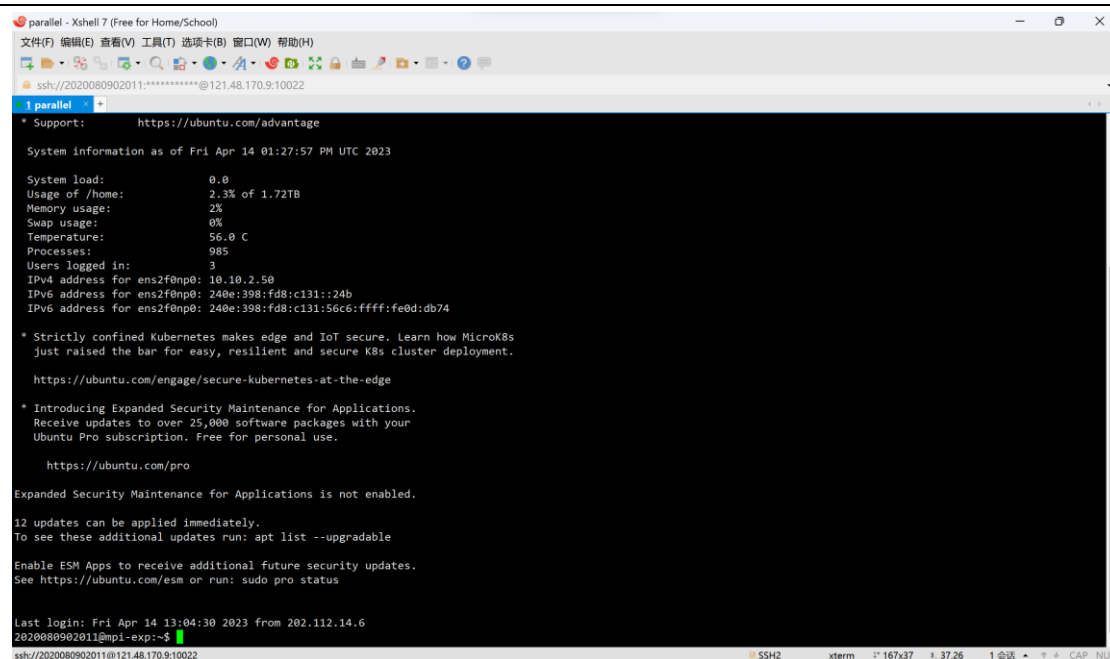


图 3 Xshell 远程连接成功

2. 基准代码的运行及程序说明。

将 $2 \sim n$ 的数字均匀地分配给 p 个进程，同时保证根号 n 落在 0 号进程范围内，这意味着需要被筛选倍数的素数在 0 号进程内可以找到并广播。

每个进程根据广播的待筛素数，找到进程范围内第一个该素数的倍数，并每次递增该素数的大小，将所负责范围内的所有倍数筛尽。重复进行上述过程，直到素数的平方大于 n ，此时每个进程中为被筛掉的素数集合就是 $2 \sim n$ 的素数集合。统计每个进程的素数个数，加和就是素数总个数。

基准代码及其关键部分的注释如下：

```
1. #include "mpi.h"
2. #include <math.h>
3. #include <stdio.h>
4. #define MIN(a,b) ((a)<(b)?(a):(b))
5. typedef long long LL; // 防止入参 n 在 1e10 的情况下爆 int
6.
7. int main (int argc, char *argv[])
8. {
9.     LL    count;        /* Local prime count */
10.    double elapsed_time; /* Parallel execution time */
11.    LL    first;         /* Index of first multiple */
12.    LL    global_count;  /* Global prime count */
13.    LL    high_value;    /* Highest value on this proc */
14.    LL    i;
15.    int   id;            /* Process ID number */
16.    LL    index;         /* Index of current prime */
17.    LL    low_value;     /* Lowest value on this proc */
18.    char *marked;        /* Portion of 2,...,'n' */
19.    LL    n;             /* Sieving from 2, ..., 'n' */
```

```

20.  int    p;                /* Number of processes */
21.  int    proc0_size;       /* Size of proc 0's subarray */
22.  LL     prime;            /* Current prime */
23.  LL     size;             /* Elements in 'marked' */
24.
25.  MPI_Init (&argc, &argv);
26.
27.  /* Start the timer */
28.
29.  MPI_Comm_rank (MPI_COMM_WORLD, &id);
30.  MPI_Comm_size (MPI_COMM_WORLD, &p);
31.  MPI_Barrier(MPI_COMM_WORLD);
32.  elapsed_time = -MPI_Wtime();
33.
34.  if (argc != 2) {
35.      if (!id) printf ("Command line: %s <m>\n", argv[0]);
36.      MPI_Finalize();
37.      exit (1);
38.  }
39.
40.  n = atoll(argv[1]);
41.
42.  /* Figure out this process's share of the array, as
43.     well as the integers represented by the first and
44.     last array elements */
45.
46.  low_value = 2 + (LL) id * (n - 1) / p;           // 使用块分配策略，每个进程得到自己负责的
value 上下界
47.  high_value = 1 + (LL) (id + 1) * (n - 1) / p;
48.  size = high_value - low_value + 1;               // size 为负责的范围大小
49.
50.  /* Bail out if all the primes used for sieving are
51.     not all held by process 0 */
52.
53.  proc0_size = (n - 1)/p;
54.
55.  if ((2 + proc0_size) < (int) sqrt((double) n)) { // 这里是检测需要的素数是否都在 0 号进程的范
围之内，若否则报错
56.      if (!id) printf ("Too many processes\n");
57.      MPI_Finalize();
58.      exit (1);
59.  }
60.
61.  /* Allocate this process's share of the array. */
62.
63.  marked = (char *) malloc (size);                 // 每个进程申请自己所需大小的空间
64.
65.  if (marked == NULL) {                             // 若无法申请到足够的空间，则报错
66.      printf ("Cannot allocate enough memory\n");
67.      MPI_Finalize();
68.      exit (1);
69.  }
70.
71.  for (i = 0; i < size; i++) marked[i] = 0;         // 将标记数组初始化
72.  if (!id) index = 0;                               // index == 0 对应着：当前的 prime == 2
73.  prime = 2;                                         // 2 为第一个素数
74.  do {
75.      // first 为每一个进程所负责的范围内的，第一个为当前素数的倍数

```

```

76.     if (prime * prime > low_value)
77.         first = prime * prime - low_value;
78.     else {
79.         if (!(low_value % prime)) first = 0;
80.         else first = prime - (low_value % prime);
81.     }
82.     for (i = first; i < size; i += prime) marked[i] = 1;    // 将当前素数的所有倍数都筛掉
83.     if (!lid) {
84.         while (marked[++index]);                            // 找到下一个素数
85.         prime = index + 2;
86.     }
87.     if (p > 1) MPI_Bcast (&prime, 1, MPI_LONG_LONG_INT, 0, MPI_COMM_WORLD);    // 广播该素
数
88.     } while (prime * prime <= n);
89.     count = 0;
90.     for (i = 0; i < size; i++)    // 统计所有进程中未被筛掉的数（素数）的个数
91.         if (!marked[i]) count++;
92.     if (p > 1) MPI_Reduce (&count, &global_count, 1, MPI_LONG_LONG_INT, MPI_SUM,    // 将其他进程
的 count 归约到 0 号进程
93.        0, MPI_COMM_WORLD);
94.     else global_count = count;
95.
96.     /* Stop the timer */
97.
98.     elapsed_time += MPI_Wtime();
99.
100.
101.     /* Print the results */
102.
103.     if (!lid) {
104.         printf ("There are %lld primes less than or equal to %lld\n",
105.             global_count, n);
106.         printf ("SIEVE (%d) %10.6f\n", p, elapsed_time);
107.     }
108.     MPI_Finalize ();
109.     return 0;
110. }

```

3. 代码优化及优化后代码的程序说明。

1) 去掉偶数

已知 2 是仅有的偶数素数，因此我们可以对其进行特判，并将标记数组中的其他偶数去除。这样无需存储偶数，算法空间可以减少一半。由于所有偶数均不考虑，因此算法时间也能减少一半。

由于去掉了偶数，进程的分配方式和映射关系需要改动。

变量 `low_value` 表示该进程范围下界的实际值，变量 `high_value` 表示该进程范围上界的实际值。变量 `low_index` 表示该进程范围下界在全局数组中的下标，变量 `high_index` 表示该进程范围上界在全局数组中的下标。`value` 和 `index` 是一一对应的，可以通过程序的两个宏定义

(VALUE_TO_INDEX、INDEX_TO_VALUE) 相互转换。例如：0 号进程中，low_value 为 3，对应的 low_index 为 0，当 value 为 5 时，index 为 1。

变量 first 是该进程范围内第一个待筛素数的倍数的局部数组的下标，每次递增该素数的大小，将该素数的倍数筛尽。

筛掉一个进程范围内的素数倍数主要有如下两个关键问题：

a. 如何找到该进程的 first 变量？

若待筛素数 prime 的平方大于进程范围上界实际值 high_value，则该进程无需筛去 prime 的任何倍数。

若 prime 的平方小于等于 high_value，则该进程可能需要筛去 prime 的倍数。若 low_value 为 prime 的倍数，low_value 需要被筛去，则 first 为 0；若 low_value 不为 prime 的倍数，令 t 为 $(\text{prime} - \text{low_value} \% \text{prime}) + \text{low_value}$ ，则 t 为该进程范围内的第一个 prime 倍数的实际值。

但是这里的 t 可能为奇数或偶数。若 t 为奇，VALUE_TO_INDEX(t) - low_index 即为 first；若 t 为偶，由于我们已经不考虑偶数，因此 t 需要加上一个 prime 成为奇数，VALUE_TO_INDEX(t + prime) - low_index 是 first。

b. 为什么去掉偶数之后，下标递加的还是素数的大小呢？

因为 first 对应的 value 必然是奇数，素数必然也是奇数，奇数之和为偶数，所以 value 就需要递加两倍的素数，对应地，index 递加一倍的素数。

去掉偶数优化的关键代码如下：

转换宏定义：

```
#define VALUE_TO_INDEX(x) ((x - 3) / 2)
#define INDEX_TO_VALUE(x) (x * 2 + 3)
```

偶数优化后的筛素数过程：

```
if (!id) index = 0;
prime = 3;
do {
    if (prime * prime > low_value) {
        first = VALUE_TO_INDEX(prime * prime) - low_index;
    } else {
        if (!(low_value % prime)) first = 0;
        else {
            LL t = (prime - low_value % prime) + low_value;
            if (t % 2) first = VALUE_TO_INDEX(t) - low_index;
            else first = VALUE_TO_INDEX(t + prime) - low_index;
        }
    }
}
```

```

    }
    // printf("prime = %d, id = %d, first = %d\n", prime, id, first);
    for (i = first; i < size; i += prime) marked[i] = 1;
    if (!id) {
        while (marked[++index]);
        prime = INDEX_TO_VALUE(index);
    }
    if (p > 1) MPI_Bcast (&prime, 1, MPI_LONG_LONG_INT, 0, MPI_COMM_WORLD);
} while (prime * prime <= n);

```

2) 消除广播

基准代码确定待筛素数 `prime` 是由第一个进程负责寻找并广播，若其他进程已经完成上一轮的工作，而第一个进程仍未广播下一个素数，则会其他进程会被阻塞，导致性能浪费。同时，`MPI_Bcast` 广播性能开销也较大。

因此我们的想法是：在偶数优化的基础上，将寻找素数这项工作让每个进程独立完成，无需通过 `MPI_Bcast` 广播同步。改动的代码量较小，只需修改寻找素数的部分即可。

消除广播优化的关键代码如下：

消除广播后的筛素数过程：

```

prime = 3;
do {
    if (prime * prime > low_value) {
        first = VALUE_TO_INDEX(prime * prime) - low_index;
    } else {
        if (!(low_value % prime)) first = 0;
        else {
            LL t = (prime - low_value % prime) + low_value;
            if (t % 2) first = VALUE_TO_INDEX(t) - low_index;
            else first = VALUE_TO_INDEX(t + prime) - low_index;
        }
    }

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

    // 进程单独计算下一个质数
    for (i = prime; i <= N; i += prime) cand[i] = 1;
    do prime++; while (prime % 2 == 0 || cand[prime]);
} while (prime * prime <= n);

```

3) cache 优化

我们的算法实际上是在一个巨大的数组上进行标记，由于标记的分散程度较大，导致 `cache`

命中率较低，因此我们的想法是，可以在一段时间内集中地标记数组的一个部分，这样能充分利用 cache，提高 cache 的命中率。在机器上使用如下命令可查看 cache 信息：

```
● 2020080902011@mpi-exp:~$ getconf -a | grep CACHE
LEVEL1_ICACHE_SIZE          32768
LEVEL1_ICACHE_ASSOC
LEVEL1_ICACHE_LINESIZE      64
LEVEL1_DCACHE_SIZE          49152
LEVEL1_DCACHE_ASSOC         12
LEVEL1_DCACHE_LINESIZE      64
LEVEL2_CACHE_SIZE           1310720
LEVEL2_CACHE_ASSOC           20
LEVEL2_CACHE_LINESIZE       64
LEVEL3_CACHE_SIZE           37748736
LEVEL3_CACHE_ASSOC           12
LEVEL3_CACHE_LINESIZE       64
LEVEL4_CACHE_SIZE            0
LEVEL4_CACHE_ASSOC
LEVEL4_CACHE_LINESIZE
```

图 4 cache 信息

可以知道该机器三级缓存空间有 37748736 字节，缓存行的大小为 64 字节。

我们需要重构循环。

重构前循环有两层，外层循环是 3~根号 n 之间的素数，内层循环是 3~n 之间的整数。

重构后循环是三层，第一层为 3~n 的整数分块，第二层为 3~根号 n 之间的素数，第三层为块内的所有整数。

由于重构的第二层需要用到 3~根号 n 之间的素数，这就要求我们每个进程在筛倍数前需要预处理出所有待筛素数。

需要确定分块的大小，经过多次尝试，使用 589824（37748736 / 64）作为块大小 block_size 时，运行时间最短，性能最优。

cache 优化的关键代码如下：

block_size 定义：

```
#define block_size 589824 // 37748736 / 64
```

进程独自预处理待筛素数过程：

```
for (i = 2; i <= N; i++) {
    if (st[i]) continue;
    primes[primes_cnt++] = i;
    for (j = i * i; j <= N; j += i) st[j] = 1;
}
```

cache 优化后的筛素数过程：

```
while (block_low_index <= high_index) {
```

```

if (block_high_index > high_index) block_high_index = high_index;

for (i = 1; i < primes_cnt && primes[i] * primes[i] <= high_value; i++) {
    LL prime_pos = primes[i];

    LL block_low_value = INDEX_TO_VALUE(block_low_index);
    LL block_high_value = INDEX_TO_VALUE(block_high_index);

    if (prime_pos * prime_pos > block_high_value) continue;

    if (prime_pos * prime_pos >= block_low_value)
        first = VALUE_TO_INDEX(prime_pos * prime_pos);
    else {
        if (!(block_low_value % prime_pos)) first = block_low_index;
        else {
            LL t = prime_pos - (block_low_value % prime_pos) + block_low_value;
            if (t % 2) first = VALUE_TO_INDEX(t);
            else first = VALUE_TO_INDEX(t + prime_pos);
        }
    }

    for (j = first; j <= MIN(block_high_index, block_low_index + size - 1); j += prime_pos) {
        marked[j - low_index] = 1;
    }
}

block_low_index = block_high_index + 1;
block_high_index += block_size;
}

```

4) 细节优化

a. 使用 memset 进行数字的初始化

关键代码：

```

// for (i = 0; i < size; i++) marked[i] = 0;
memset(marked, 0, size);

// for (i = 0; i < N; i++) st[i] = 0;
memset(st, 0, N);

```

b. 与 2 相关的乘除法运算改成移位运算

例如：

```

#define VALUE_TO_INDEX(x) ((x - 3) / 2)
#define INDEX_TO_VALUE(x) (x * 2 + 3)

```

八、调试说明：

在本次实验中，我遇到的绝大多数需要调试的情况都可以指定进程 id 号，并打印（printf）出该进程相关变量的信息进行 DEBUG。例如：

```
if (id == 1) printf("id = 1, prime_pos = %lld, first = %lld\n", prime_pos, first);
```

若遇到实在无法用 printf 解决的问题，可以使用 GDB 命令，用指定调试进程的 pid。注意调试的程序需要是 mpic++ 在调试模式下编译的（添加 -g 选项）。

九、实验数据及结果分析：

n 为十的十次方的结果如下：

```
● 2020080902011@mpi-exp:~$ mpiexec -n 16 ./get_primes_v0 10000000000
There are 455052511 primes less than or equal to 10000000000
SIEVE (16) 28.034435
```

图 5 基准代码的运行结果

```
● 2020080902011@mpi-exp:~$ mpiexec -n 16 ./get_primes_v1 10000000000
There are 455052511 primes less than or equal to 10000000000
SIEVE (16) 14.015778
```

图 6 去掉偶数的运行结果

```
● 2020080902011@mpi-exp:~$ mpiexec -n 16 ./get_primes_v2 10000000000
There are 455052511 primes less than or equal to 10000000000
SIEVE (16) 13.613911
```

图 7 消除广播的运行结果

```
● 2020080902011@mpi-exp:~$ mpiexec -n 16 ./get_primes_v3 10000000000
There are 455052511 primes less than or equal to 10000000000
SIEVE (16) 4.812951
```

图 8 cache 优化的运行结果

```
● 2020080902011@mpi-exp:~$ mpiexec -n 16 ./get_primes_v4 10000000000
There are 455052511 primes less than or equal to 10000000000
SIEVE (16) 3.732432
```

图 9 细节优化的运行结果

1. 去掉偶数优化的运行时间几乎为基准程序的一半，符合我们对该优化性能提高一倍的理论预期。

2. 消除广播优化的运行时间较去偶数优化的差别不大，但仍有一定提升。

3. cache 的优化极大地提升了并行化的性能。

4. 细节上的优化对整体性能影响不大。

综合运行结果，每一次的优化对程序性能都有一定的提高，其中 cache 优化的效果最好，对

性能的提升最大。

n 为十的九次方的详细结果如下：

	1	2	4	8	16
基准代码	2.093150 (s)	1.081433	0.576837	0.318080	0.213410
去掉偶数	0.834028	0.442868	0.234321	0.111315	0.088652
消除广播	0.836261	0.424034	0.214796	0.107629	0.061384
cache 优化	0.672976	0.344099	0.170782	0.085287	0.056114
细节优化	0.515648	0.263755	0.144837	0.065672	0.044363

每一份版本的代码都将进程规模为 1 的情况认为是串行算法执行时间，然后使用将串行算法执行时间除以并行算法执行时间得到加速比。

1. 基准代码在进程规模为 1、2、4、8、16 时，加速比为 1、1.936、3.629、6.581、8.941。随着进程规模的增加，加速比有所上升。但是由于程序运行必然有额外的开销，无法得到线性加速比。

2. 去掉偶数在进程规模为 1、2、4、8、16 时，加速比为 1、1.883、3.559、7.493、9.408。去掉偶数优化后的代码较基准代码的对加速比提升不是十分明显。

3. 消除广播在进程规模为 1、2、4、8、16 时，加速比为 1、1.972、3.893、7.770、13.623。由于消除了广播，进程无需因为等待 0 号进程广播素数而被阻塞，进程间的独立性更强了，因此加速比提高较大。

4. cache 优化在进程规模为 1、2、4、8、16 时，加速比为 1、1.955、3.941、7.891、11.993。cache 优化后的代码在进程规模为 1、2、4、8 的情况下，能得到近似线性加速比的加速比结果，但在进程规模为 16 的情况下，加速比相较线性加速比较大。

5. 细节优化在进程规模为 1、2、4、8、16 时，加速比为 1、1.955、3.560、7.852、11.623。

十、实验结论：

通过对基准代码采取一系列的优化措施，MPI 并行程序的性能得到了相当大的提升，其中 cache 优化效果最佳。

十一、总结及心得体会：

1. 通过本次实验，熟悉了 MPI 编程，对并行编程有了更加深刻的理解。最重要的是培养了调试并行代码的能力，学习到如何使用 GDB 调试工具。
2. 原来只认识埃氏筛的串行算法，在本次实验之后，学习到了埃氏筛的并行算法，这提高了我对算法的理解。
3. 收获了如何优化并程序的经验。过去我从没有想过 cache 居然能够在提高程序性能上起到如此巨大的作用，让我懂得从 cache 的新视角看计算机程序，体会到了并行编程之美。

十二、对本实验过程及方法、手段的改进建议：

1. 多进行类似有趣的并行编程实验，能提高同学的学习积极性。
2. 希望能提供更多 cache 优化相关的资料供同学们学习。

报告评分：

指导教师签字：