3.17 Design a parallel algorithm to find the first occurrence of the pattern in the text.

假设有 800GB 文本，8 个处理器，可以考虑把文本分成 100 个一级子任务，每个一级子任务 8GB（忽略 pattern 长度），分给 8 个处理器就是每个处理器可以分到每一份 8GB 文本中的 1GB 文本（二级子任务）。让 8 个处理器同时用 Boyer Moore 字符串匹配算法（需预先生成共享的好坏规则表）。只要某个处理器找到一个匹配，就可通知它后面的处理器结束任务，并把自己的结果发给头结点，然后只需要等它前面的处理器完成小于 1GB 的文本匹配。这避免了后面先找到答案的处理器等待太长时间，同时最多只会浪费小于 8GB（1%= $\frac{1}{\text{一级子任务数}}$）的匹配资源。当然，100 份这个迭代次数参数可以自由设置，控制每次迭代最大周期在一个较短的时间内。

3.18 Given a list of n keys, a[0], a[1],…, a[n-1], all with distinct values, design a parallel algorithm to find the second-largest key on the list.

设有$p = 2^k$个处理器，编号为 1，2，…，p（p≥2），第 1 号为 root task。考虑 Binomial Tree 的网络结构，给每个处理器分配 $\frac{n}{p}$份，各自计算自己那份数据中的最大值（互不相同），只需要 $\frac{\mathrm{n}}{\mathrm{p}} - 1$ 次比较，计算完后通过$\log_2 \mathrm{p}$次通信传值比较，求最大的两个数，最终在 1 号处理器上得到整个 list 的第二大的 key。

若 p 不是 2 的指数次方，则需要$\lfloor \log_2 \mathrm{p} \rfloor + 1$次通信和计算。二者合并起来就是需要$\lceil \log_2 \mathrm{p} \rceil$次通信和计算。

3.19 Given a list of n keys, a[0], a[1],…, a[n-1], design a parallel algorithm to find the second-largest key on the list. Keys do not necessarily have distinct values.

当存在相同值时，可能每个节点上得出来的最大值都一样，因此在每个节点上都必须记录前两个最大的值。如果把新值挨个与第二大、最大值比较取前 2 大，

最坏情况下比较$2\frac{n}{p}-3$次。可以考虑在每个节点上用分治法，把自己那份数据集一分为二，递归地先求出每份小数据集中的前两大数，最后归并，这样大概可以减少四分之一的比较次数。其他与上题思路一样。

4.2

241，128，99，13，127，0，1，1

```cpp
#include <iostream>
using namespace std;
typedef unsigned short us;
int main() {
    char a = 13, b = 22, c = 43, d = 64, e = 99;
    cout << (((us)(char)(a + b + c + d + e)) & (us)255) << endl;
    cout << (((us)(char)(a * b * c * d * e)) & (us)255) << endl;
    cout << (((us)(char)(a | b | c | d | e)) & (us)255) << endl;
    cout << (((us)(char)(a & b & c & d & e)) & (us)255) << endl;
    cout << (((us)(char)(a || b || c || d || e)) & (us)255) << endl;
    cout << (((us)(char)(a && b && c && d && e)) & (us)255) << endl;
    return 0;
}
```

4.8

（1） 代码链接

关键代码：主体算法还是按课件上的代码，下面列出不同的地方。

1. 在找 sieving primes 时丢弃偶数，但需要每块的数据个数也是偶数

```cpp
if (!id) { // 找下一个sieving primes
    while (marked[++index] || index % 2 == 0) // 是偶数就丢弃
        ;
```

2. 求每一块内部的奇数对数

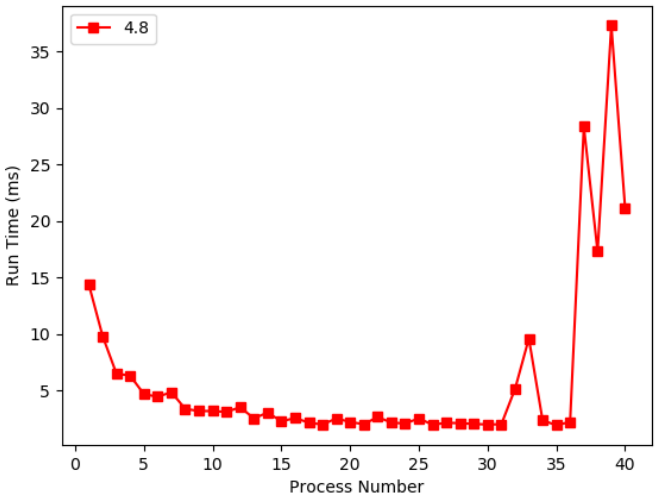
```
73      count = 0;
74      if (!id) {
75          // i=1代表3这个数
76          for (i = 1; i < size - 2; i += 2) // 假设每块都是偶数个，除了第0块
77              if (!marked[i] && !marked[i + 2])
78                  count++;
79      } else {
80          for (i = 0; i < size - 2; i += 2) // 第一个位置是奇数
81              if (!marked[i] && !marked[i + 2])
82                  count++;
83      }
```

3. 求边界情况，用 MPI_Send 和 MPI_Recv 函数传递所有边界结果（每个进程自己数据集的第一个和最后一个奇数是否为质数）到 0 号进程再统一做判断

```
86      if (!id) {
87          boundary = (int *)malloc(p * 2 * sizeof(int));
88          memset(boundary, 0, sizeof(boundary));
89          // 前p位存储p个进程的块的倒数第2个数是否是质数
90          // 后p位存储p个进程的块的顺数第1个数是否是质数，0号进程的没有用
91      }
92      MPI_Reduce(&count, &global_count, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);
93
94      if (id) {
95          int firstodd = marked[0],
96              lastodd = marked[size - 2]; // 转成int，因为不能传递bool变量！
97          // 把边界结果发送给0号进程
98          MPI_Send(&lastodd, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);
99          MPI_Send(&firstodd, 1, MPI_INT, 0, 1, MPI_COMM_WORLD);
100     }
101
102     if (!id) { // 边界情况
103         // 接收其他进程的边界结果
104         for (i = 1; i < p; i++) {
105             MPI_Recv(boundary + i, 1, MPI_INT, i, 0, MPI_COMM_WORLD,
106                      MPI_STATUS_IGNORE);
107             MPI_Recv(boundary + i + p, 1, MPI_INT, i, 1, MPI_COMM_WORLD,
108                      MPI_STATUS_IGNORE);
109         }
110
111         boundary[0] = marked[size - 2];
112         for (i = 0; i < p - 1; i++)
113             if (!boundary[i] && !boundary[p + 1 + i])
114                 global_count++;
115     }
```

(2) 实验结果

1. 5 个进程的时候运行时间为 4.329ms，一共有 8169 个连续质奇数对。
2. 分别用 1-100 个进程运行，测运行时间，结果如下：



进程达到 30 个后进程间通信的资源消耗限制了速度的进一步上升。

3. 根据教材上的代码，在运行 1-7 个进程时可以得到 100 万以内的正确质数个数，但当运行大于 8 个进程后就大部分结果都是错误的了，不清楚原因，21、192 个进程等时候是才对的。
4. 由于采用了偶数丢弃，只有在每一块数据集正好都是偶数个数的时候答案才是 8169。

4.11

(1) 代码链接

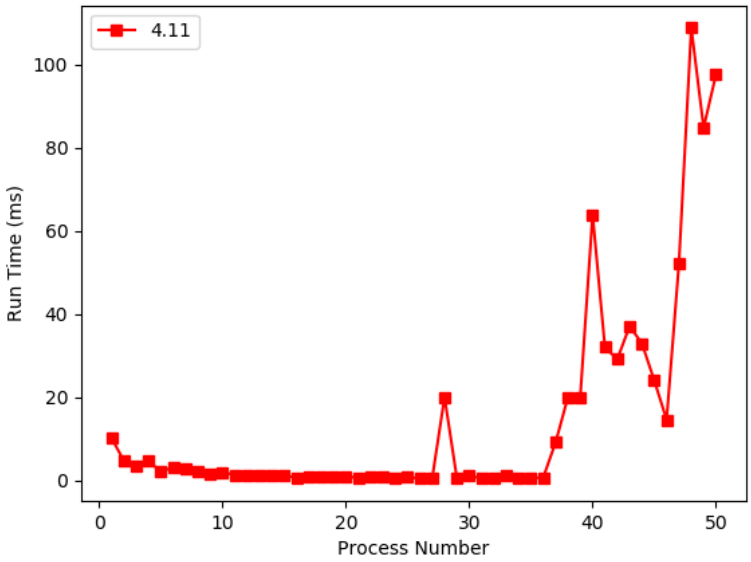
关键代码：单线程算法题目已给出，只需要完成分块和合并即可。

```
k = INTERVALS % p; // 多余的k个平均分给前k个进程
if (k) {
    if (id < k) { // 前k个进程每个多分一个数
        size = (INTERVALS + p) / p;
        begin = size * id;
    } else {
        size = INTERVALS / p;
        begin = size * id + k; // 前k个已分配完
    }
} else { // 没多余的数
    size = INTERVALS / p;
    begin = size * id;
}
```

```
MPI_Reduce(&area, &global_area, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
```

(2) 实验结果

1. 分别用 1，2，…,100 个进程运行，得到结果如下（51-100 的省略了）：



2. 34 个进程运行的时候最快，0.384ms，计算结果为 3.14159265358987527250，小数点后 12 位是对的。
3. 单个进程跑时达到了小数点后 13 位精度，其他进程跑时都是 12 位精度。
4. 达到 37 个进程时运行时间开始激增。