电 子 科 技 大 学

实 验 报 告

学生姓名：	学 号：

一、实验室名称：

二、实验项目名称：基于 MPI 实现埃拉托斯特尼筛法及性能优化

三、实验原理：

1. 埃氏筛

   埃氏筛是一种朴素的质数筛选法。其内容即通过将质数从小到大找出并标记其所有倍数来筛掉合数。可以证明，筛出N以内的所有质数的时间复杂度是$O(N\log\log N)$。

2. 区间筛

   原始的埃氏筛仅用于筛出[1,N]区间内的质数。而在大规模并行埃氏筛时单个程序需要处理的区间往往是[L,R]，即左端点不是 1 的区间。为了避免对[1,L-1]重复筛选，只需要提前筛出$\sqrt{R}$内的质数并将其在[L, R]内的倍数筛掉。时间复杂度是$O\left(\left(\sqrt{R}+R-L\right)\log\log R\right)$。

3. 埃氏筛的并行化

   若有P个进程在并行计算N以内的质数个数，可将[1,N]尽量平均分成P个区间，分得区间左端点为1的进程将不断寻找小于等于$\sqrt{\mathrm{N}}$的质数并将其广播给其他进程。

设广播代价为$\lambda\log P$，单次标记代价为χ，则时间复杂度为$O\left((\sqrt{N}/\log N)\lambda\log P+(\chi N\log\log N)/P\right)$。

4. 埃式筛的优化

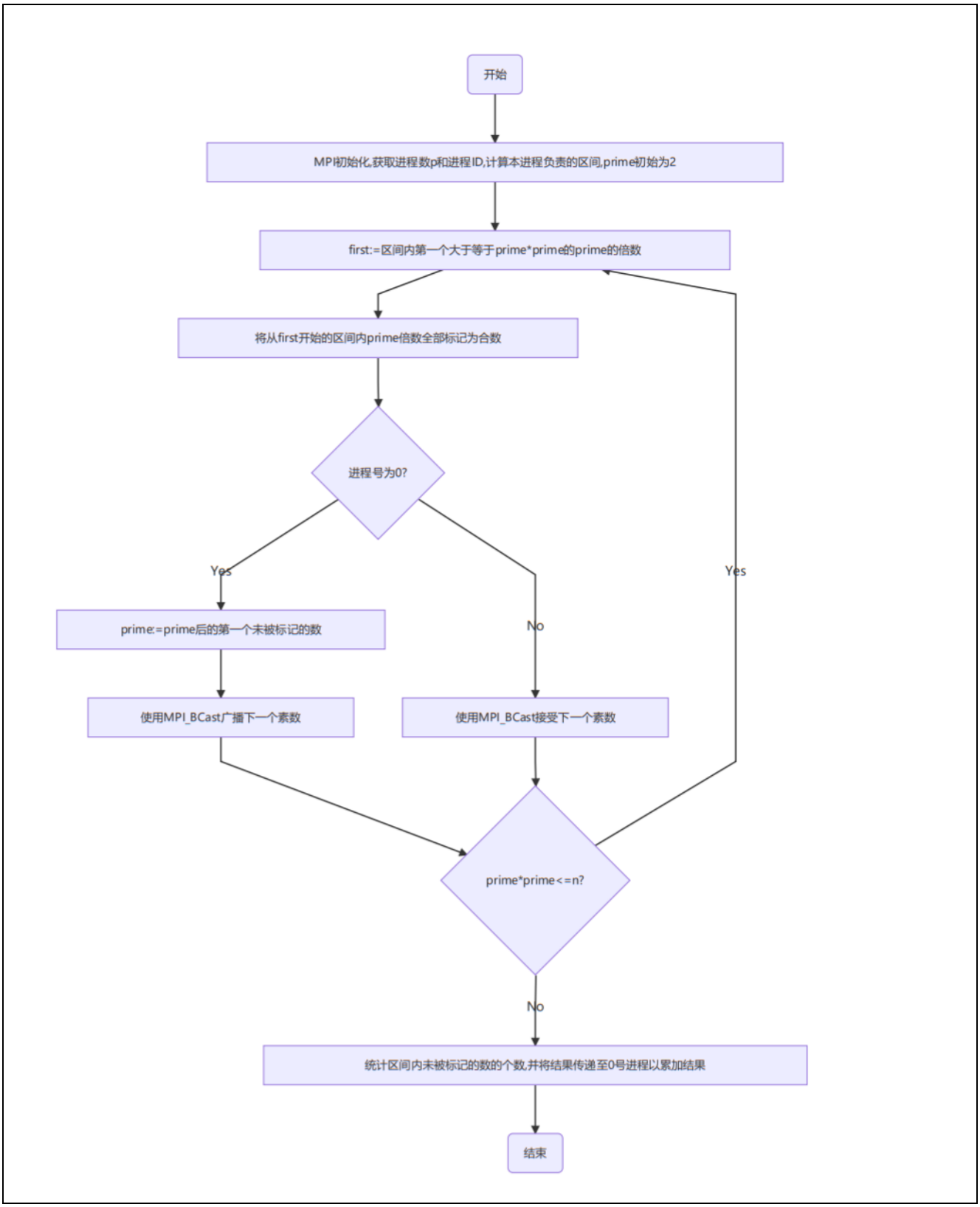
   (1) 去除偶数：在算法执行时直接跳过偶数，最终结果计算时将 2 加上，可以优化一半左右的性能，此外在筛奇数时只需要筛掉奇数的奇数倍数。

   (2) 消除广播：每个区间执行时都将自己所需要的质数单独筛出来，这样可以使 0 号进程不需要每次都将获得的质数广播出去，减少了许多广播过程中的开销

   (3) Cache 优化：在处理某个质数 p 时，将区间全部扫描一遍会造成大量的缓存失效，因此我们对一个一个进程所需要处理的区间分成若干个和 cache 长度相同的区间，每次使所有质数将区间中的数都访问一遍即可，这样 cache 命中率将极大程度提高。

   (4) 线性筛优化：初始化处理每个进程自身需要的小于等于$\sqrt{\mathrm{N}}$的质数时，不难发现许多数被标记了多次，这也导致筛出小于等于$\sqrt{\mathrm{N}}$的质数复杂度为是$O\left(\sqrt{N}\log\log N\right)$，现在有一种理论更好的筛法，仅需要$O\left(\sqrt{N}\right)$的复杂度即可筛出小于等于$\sqrt{\mathrm{N}}$的质数。

5. 程序框图


开始
MPI初始化,获取进程数p和进程ID,计算本进程负责的区间,prime初始为2
first:=区间内第一个大于等于prime*prime的prime的倍数
将从first开始的区间内prime倍数全部标记为合数
进程号为0?
Yes
No
prime:=prime后的第一个未被标记的数
使用MPI_BCast广播下一个素数
使用MPI_BCast接受下一个素数
prime*prime<=n?
Yes
No
统计区间内未被标记的数的个数,并将结果传递至0号进程以累加结果
结束

四、实验目的：

1.熟悉 MPI 用法，使用 MPI 完成埃式筛的并行运算

2.对埃式筛的并行运算进行性能优化。

五、实验内容：

1. 基于 MPI 实现能够分布式并行计算的埃氏筛法.
2. 对其进行优化
3. 性能评估和分析

六、实验器材（设备、元器件）：

Dell 笔记本一台
CPU：Intel® Core™ i7-8750H CPU @ 2.20GHz 2.21 GHz
L1 缓存：384KB

七、实验步骤及操作：

0. 源代码版本清单：
   (1) Version1.cpp:修改了原本代码中的错误
   (2) Version2.cpp:忽略偶数，仅筛选奇数
   (3) Version3.cpp:消除广播，每个进程各自筛选出所需要的质数。
   (4) Version4.cpp:重构循环，充分利用 L1 缓存。
   (5) Version5.cpp:理论上更优的筛法尝试
1. Version1.cpp 更改内容：
   该版本仅针对示例代码中存在问题进行了修改，存在问题主要有 2：
   (1) 区间计算公式错误
   (2) MPI_Barrier 使用错误，在进程多于处理器数量时，0 号进程可能过晚开始导致计时错误

具体更改如下：

```
/*旧代码部分：
MPI_Barrier(MPI_COMM_WORLD);
elapsed_time = -MPI_Wtime();*/

//更改部分开始
MPI_Barrier(MPI_COMM_WORLD);
elapsed_time = -MPI_Wtime();
//进程数大于实际处理器个数时，0号进程可能过晚开始导致计时错误
MPI_Barrier(MPI_COMM_WORLD);
//更改结束

if (argc != 2) {
    if (!id) printf("Command line: %s <m>\n", argv[0]);
    MPI_Finalize();
    exit(1);
}

n = atoi(argv[1]);

//分块存在一定问题
/*low_value = 2 + id * (n - 1) / p;
high_value = 1 + (id + 1)*(n - 1) / p;
size = high_value - low_value + 1;*/

//更改部分开始
int q = (n + 1) / p, r = (n + 1) % p;
if (id < r) {
    low_value = id * (q + 1);
    high_value = low_value + (q + 1) - 1;
}
else {
    low_value = r * (q + 1) + (id - r) * q;
    high_value = low_value + q - 1;
}
if (id == 0) low_value = 2;
size = high_value - low_value + 1;
//更改结束
```

2. Version2.cpp 更改内容
   主要更改内容即是去除偶数影响，具体实现方法是将标记数组的下标 index 对应到 2*index+1 这个质数上，如下图所示。

```
                    //每个k都对应一个2k+1
                    if (id == 0) low_value = 3;
                    size = (high_value)/2 - low_value/2;

    do {
        if (prime * prime > low_value)
            first = prime * prime - low_value;
        else {
            if (!(low_value % prime)) first = 0;
            else first = prime - (low_value % prime);
        }
        //偶数跳过 再加一次
        if ((first + low_value) % 2 == 0)first += prime;
        for (i = first>>1; i < size; i += prime) marked[i] = 1;
        if (!id) {
            while (marked[++index]);
            //下标与prime对应是2k+1
            prime = index * 2 + 3;
        }
        if (p > 1) MPI_Bcast(&prime, 1, MPI_INT, 0, MPI_COMM_WORLD);
    } while (prime * prime <= n);
```

3. Version3.cpp 更改内容

为了消除通信影响，每个进程改为单独筛出自己所需要的质数，即筛出根号 R 以内的质数，这样做的原因是如果一个数为合数，那么他一定有一个因子小于根号 R，具体更改如下图所示。

```
    //更改开始：预先筛出需要用的质数
    int pre_sz = sqrt(high_value) + 1;
    char* pre_mark = (char*)malloc(pre_sz);
    for (int i = 0; i != pre_sz; ++i)
        pre_mark[i] = 0;
    for (int i = 2; i != pre_sz; ++i)
        if (!pre_mark[i])
            for (int j = 2 * i; j < pre_sz; j += i)
                pre_mark[j] = 1;
    if (pre_mark == NULL) {
        printf("Cannot allocate enough memory\n");
        MPI_Finalize();
        exit(1);
    }
    //更改结束

    do {
        if (prime * prime > low_value)
            first = prime * prime - low_value;
        else {
            if (!(low_value % prime)) first = 0;
            else first = prime - (low_value % prime);
        }
        if ((first + low_value) % 2 == 0)
            first += prime;
        for (i = first / 2; i < size; i += prime)
            marked[i] = 1;
        //prime直接从预先的数组寻找
        while (pre_mark[++index]);
        prime = index;
    } while (prime * prime <= high_value);
```

4. Version4.cpp 更改内容

为了提高 cache 命中率，我将每个进程所筛选区间划分为若干个 32*1024 大小（L1 缓存 32KB，一个 bool 型变量 1B）的区间，每次区间将所有质数的倍数全部处理，这样 cache 命中率可以大幅提高，同时我们可以将$\sqrt{R}$内质数在筛选的过程中保存下来，避免遍历标记数组寻找下一个质数，此外我们将每

个区间，每个质数的起始位置都记录下来，避免了每次都要计算位置导致大量的乘除法造成的时间浪费。

```
for (int i = 3, sqri = 9; sqri <= sqrtrb; sqri += ((i + 1) << 2), i += 2)
{
    if (!vis[i])
    {
        pri[++tot] = i;
        pos[tot] = (i * mmax(((low_val - 1) / i + 1) | 1, 1ll * i)) >> 1;
        for (int j = sqri; j <= sqrtrb; j += 2 * i)
            vis[j] = 1;
    }
}

for (int i = pri[tot]; i <= sqrtrb; i += 2) {
    if (!vis[i]) {
        pri[++tot] = i;
        pos[tot] = (i * mmax(((low_val - 1) / i + 1) | 1, 1ll * i)) >> 1;
    }
}



ll count = (high_val >> 1) - (low_val >> 1);
if (!id) count++;
for (ll l = low_val, r; l < high_val; l = r) {
    r = mmin(high_val, l + (cache_sz << 1));
    const ll bl = l >> 1, br = r >> 1;
    memset(vis, 0, br - bl);
    for (int i = 1; i <= tot; ++i) {
        const int p = pri[i];
        ll j = pos[i];
        for (; j < br; j += p)
            vis[j - bl] = 1;
        pos[i] = j;
    }
    for (ll j = bl; j < br; ++j)
        count -= vis[j - bl];
}
```

5. Version5.cpp 更改

这个版本我采用了时间复杂度更优秀的线性筛来进行$\sqrt{R}$以内素数的筛选，但不幸的是线性筛因它需要大量的取模运算因而时间常数略大于埃式筛，因此在 1e9 以下的数据规模表现略微不如埃式筛，但是在 1e12 数据量表现上明显优于埃式筛。

更改如下：

```
for (int i = 3; i <= sqrtrb; i += 2) {
    if (!vis[i]) {
        pri[++tot] = i;
        pos[tot] = (i * mmax(((low_val - 1) / i + 1) | 1, 1ll * i)) >> 1;
    }
    for (int j = 1; j <= tot && i*pri[j] <= sqrtrb; ++j) {
        vis[i*pri[j]] = 1;
        if (i%pri[j] == 0)break;
    }
}
```

八、实验数据及结果分析：

数据量 1e9，时间单位：秒					
版本 进程数	version1	version2	version3	version4	version5
1	9.674397	4.82658	4.84975	0.622653	0.625551
2	7.66421	4.819765	3.781807	0.369918	0.352724
4	7.472055	4.601988	3.745691	0.220461	0.237831
8	7.217596	4.756256	3.735372	0.138038	0.135613
16	7.021352	4.760233	3.658202	0.136695	0.136079

1. version1 到 version2，措施为去除偶数，加速比约为 1.5，与预计 2 加速比有出入，但考虑到程序存在通信等其他问题，该加速比仍算合理。
2. version2 到 version3，措施为消除通信，加速比约为 1.3，说明原先程序通信开销事实上是比较大的，而筛出$\sqrt{R}$内质数开销比不断进行通信要小很多。
3. version3 到 version4，措施为 cache 优化，加速比根据不同进程数在 8-20 之间，现代 CPU cache 命中访存大约只需要 4 个时钟周期，未命中则需要上百个时钟周期，因此这一步的优化十分巨大，进程越多，cache 利用率越高，加速比越高。
4. version4 到 version5，措施为线性筛优化，从数据中可以看出，虽然直接对时间复杂度进行了优化，但是线性筛因其需要大量取模运算因而实际表现并不比埃式筛好多少，加速比约为 1。但是大数据下（1e11）线性筛在复杂度层面的优越性便体现了出来。

埃式筛与欧拉筛在大数据下的比较（时间单位：秒）					
数据量：1e10	version4	version5	数据量：1e11	version4	version5
测试数据	1.696735	1.625805	测试数据	28.868678	21.569756
	1.666166	1.696635		27.75009	23.873437
	1.670561	1.650008		27.204989	24.623426
	1.632052	1.633063		28.791789	25.303157
	1.649393	1.663396		28.752981	25.495827
平均值	1.6629814	1.6537814	平均值	28.2737054	24.1731206

可以看到大数据(1e11)数据量级下，线性筛相比埃式筛有明显的优势，加速比约为 1.17

九、实验结论：

并行化的埃式筛充分展现了其优势，在 0.16 秒左右即可完成 1e9 范围内的质数筛选，充分体现了其价值。5 个优化版本中提升最大的是重构循环以利用 cache，这一步加速比达到了夸张的 8-20，说明我们在设计程序时需要充分考虑到现代计算机的特性。

十、总结及心得体会：

本次实验我提高了代码阅读能力，更对 MPI 多进程编程有了充分的了解。
实验过程中我尝试了许多优化，对筛法的了解也加深了许多，在编程过程中我遇到了许多困难，这证明了我对 C 语言和 MPI 的了解并不充足，需要更加努力学习。此外在算法设计过程中的并行计算设计思想也让我对本课程有了更加深入的认识。

十一、对本实验过程及方法、手段的改进建议：

建议老师提供更加容易修改的代码，在进行 Version1 与 Version2 的修改过程中，因为部分示例代码实在难以理解导致花费了大量时间，重构循环时我也采用全部重写的方式来实现，希望示例代码中能够增加更多的注释以帮助理解。
此外建议提高测试数量级，在 1e9 数据量级下，部分优化由于除法、取模运算多，其表现反而不如朴素的算法，在 1e10 乃至 1e11 数据量级下，程序运行也不会超过 1 分钟时间。

报告评分：

指导教师签字：