

电子科技大学

实验报告

学生姓名：蔡与望	学号：2020010801024
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>埃拉托斯特尼筛法是一种简单有效的素数筛选算法，用于找出指定范围内的所有素数。它的原理是：从 2 开始，依次将每个素数的所有倍数标记为合数，剩下未标记的数就是素数。</p> <p>在 MPI 并行计算的具体语境下，埃拉托斯特尼筛法的具体步骤如下：</p> <ol style="list-style-type: none">1. 将指定的目标范围划分为若干块，每块都分配给一个处理器，让它们并行处理。2. 很显然，埃拉托斯特尼筛法所选的基数不会大于\sqrt{n}，所以处理器需要遍历小于等于\sqrt{n}的素数，并在自己的负责范围内，依次标记它们的所有倍数。3. 当每个处理器都筛选出了自己范围内的素数之后，在所有进程间进行一次通信，就可以得到整个范围内的素数。	
<p>四、实验目的：</p> <ol style="list-style-type: none">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。	
<p>五、实验内容：</p> <ol style="list-style-type: none">1. 安装部署 MPI 实验环境，并调试完成基准代码，并实测在不同进程规模（1，2，4，8，16）加速比，并合理分析原因。2. 完成优化 1：去除偶数优化，并实测在不同进程规模（1，2，4，8，16）加速比，并合理分析原因。3. 完成优化 2：消除广播优化，并实测在不同进程规模（1，2，4，8，16）加速比，并合理分析原因。4. 完成优化 3：cache 优化，并实测在不同进程规模（1，2，4，8，16）加速比，并合理分析原因。5. 性能得分：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。	

六、实验器材（设备、元器件）：

本地开发环境：

1. Arch WSL
2. Visual Studio Code
3. g++ 12.2.1
4. mpich 4.1.5

目标机器配置：

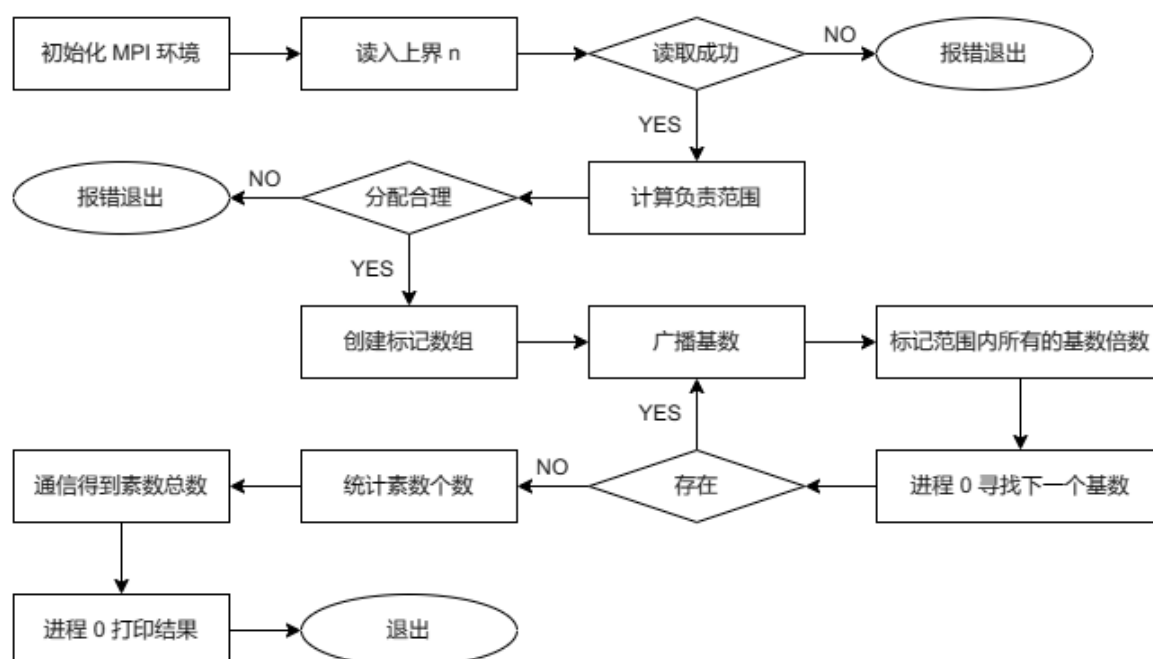
1. 浪潮 5280M4
2. CPU: E5-2660 v4 2 颗
3. 内存: 256G

七、实验步骤及操作：

1. 基准代码

基准代码的核心思路是：让进程 0 管理每次迭代的基数，并通过广播告知其它进程，然后所有进程都根据新基数来标记筛选。

程序框图如下：



SSH 到目标机器，运行命令：

```
mpic++ -o optimizer optimizer.cpp && mpirun -np PROCESS_COUNT ./optimizer 10000000000
```

分别令 PROCESS_COUNT 为 1、2、4、8、16，多次测试测量加速比，并分析原因。

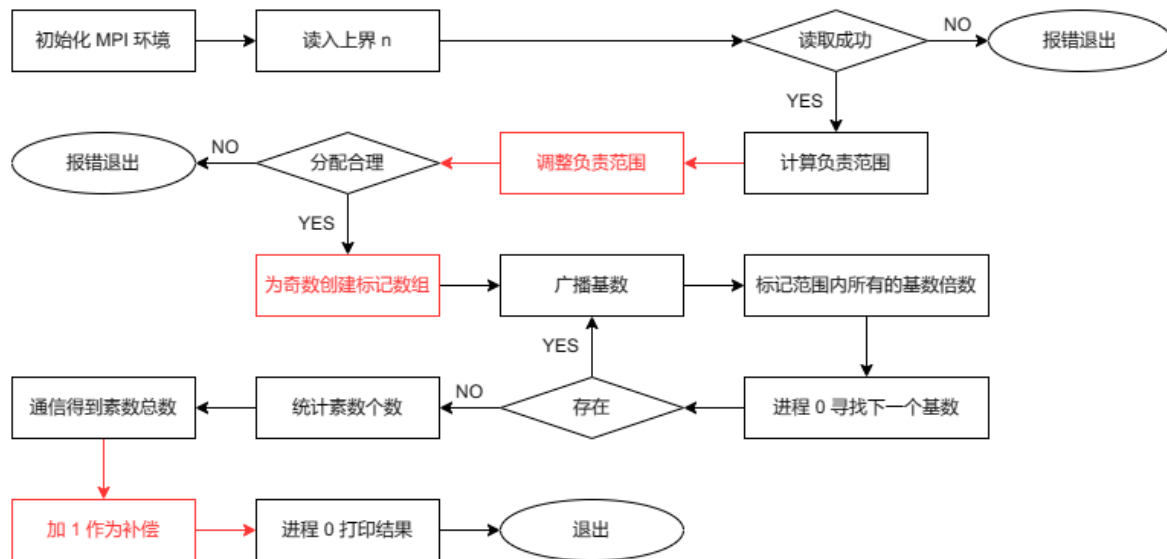
2. 去除偶数

经过分析，我们容易发现，基准代码中的基数是从 2 开始的，这意味着我们第一次循环就需要标记几乎一半的数，而且之后每次基数都需要考虑偶数是否是它的倍数。这是很大的性能损失。

而实际上，我们可以直接忽略所有的偶数。这就相当于要处理的数变成了原来的一半，性能可以得到极大的提升。要实现这一点，我们需要做下面几件事：

- 各进程计算自己负责范围的上下界时，可以预先调整上下界为奇数。
- 标记数组只标记奇数是否为合数。
- 寻找范围内第一个基数倍数的下标时，需要对公式做适当的修改。
- 2 是唯一的偶素数，所以最后要加 1 作为补偿。

程序框图如下（红色部分为新增或更新后的流程）：



SSH 到目标机器，运行命令：

```
mpic++ -o optimizer optimizer.cpp && mpirun -np PROCESS_COUNT ./optimizer 10000000000
```

分别令 PROCESS_COUNT 为 1、2、4、8、16，多次测试测量加速比，并分析原因。

3. 消除广播

在优化 1 后的代码中，我们使用进程 0 向所有进程广播下一个基数。但只要是通信，就会有开销，尤其是在多机环境下。所以我们需要找一种方法消除广播。

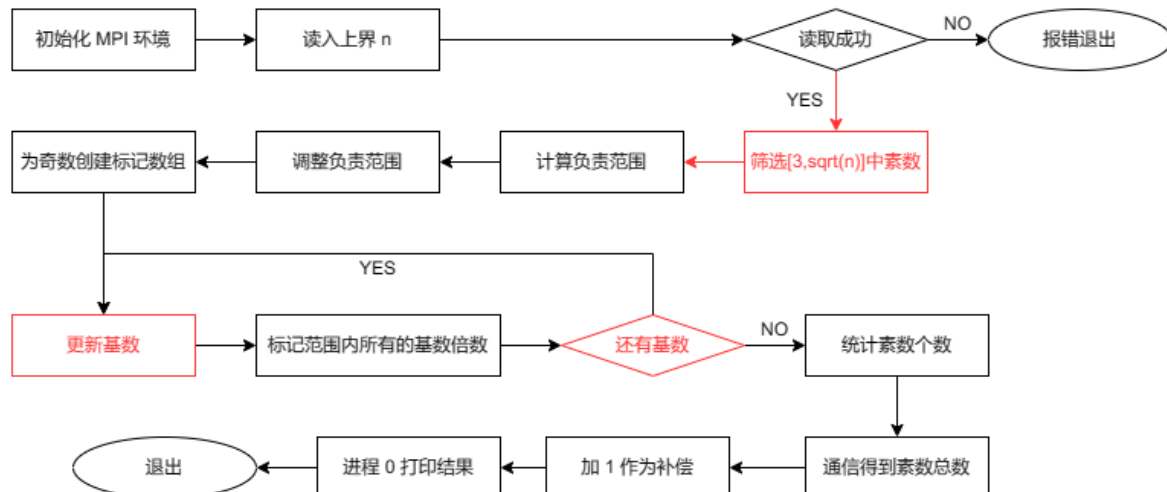
经过分析我们可以发现，进程 0 广播的这些基数其实和进程 0 本身毫无关系。每个进程都可以独自算出各个基数，而完全不需要依赖某一个进程的主导。那么，改进的方法就很显而易见了：让每个进程都各自算出 $[3, \sqrt{n}]$ 的所有素数，然后把它们作为基数，自行对负责范围内的奇数进行筛选。

之所以这个上界是 \sqrt{n} ，是因为：在标记大于 \sqrt{n} 的素数的倍数时，这些倍数都一定已经被小于 \sqrt{n} 的因子标记过了。

同时，我们也可以移除基准代码中“Too many processes”的检查，因为此时各进程已经不再依赖进程 0 来获得基数。

至于如何获取 $[3, \sqrt{n}]$ 的所有素数，我们可以再使用一遍埃拉托斯特尼筛法，并专门进行一些优化。

程序框图如下（红色部分为新增或更新后的流程）：



SSH 到目标机器，运行命令：

```
mpic++ -o optimizer optimizer.cpp && mpirun -np PROCESS_COUNT ./optimizer 10000000000
```

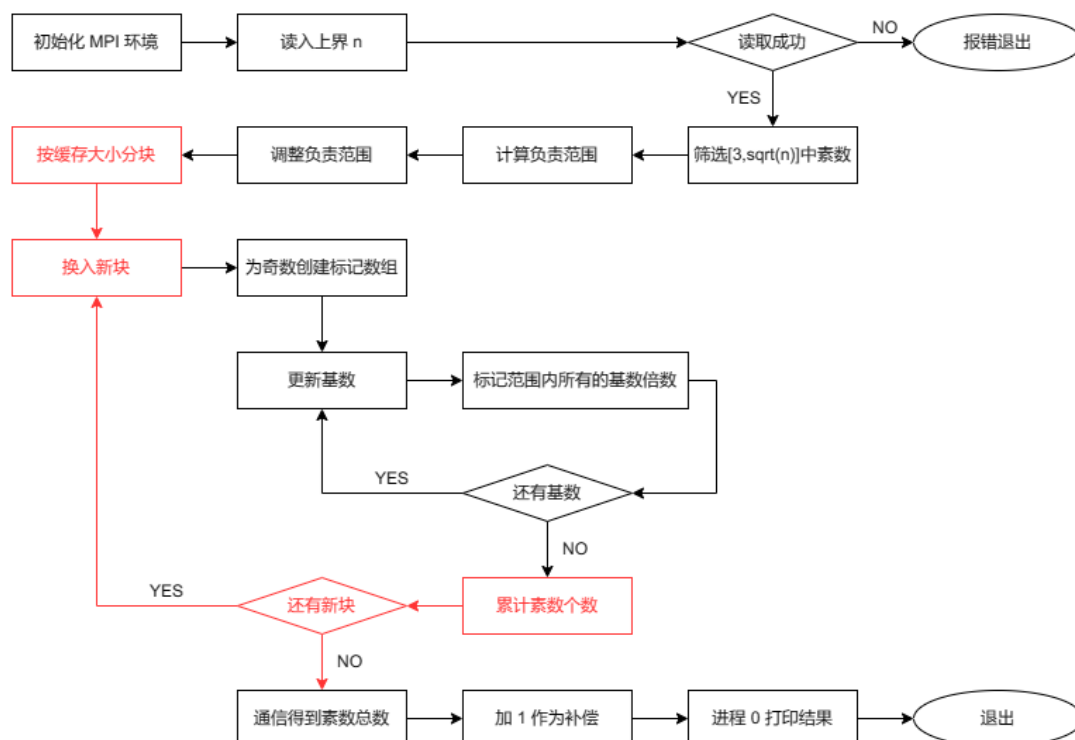
分别令 PROCESS_COUNT 为 1、2、4、8、16，多次测试测量加速比，并分析原因。

4. Cache 优化

在优化 2 后的代码中，我们会发现，进程在处理自己的负责范围时，直接在整个范围上进行迭代。这就会导致：在标记某个基数的倍数时，进程为了标记完一整个范围，进行了多次 cache 的换入换出。比如 cache 一共可以存放 16 个数字，但进程需要处理 64 个数，那么在一次标记过程中，就会发生 4 次 cache 的换入换出。

问题的解决方案也比较明显，就是每个进程再把自己的负责范围分为多个小块，每块都和 cache 保持大小一致。在计数时，也是一块一块依次计数。这样，cache 就不会频繁换入换出，从而大幅提高性能。

程序框图如下（红色部分为新增或更新后的流程）：



SSH 到目标机器，运行命令：

```
mpic++ -o optimizer optimizer.cpp && mpirun -np PROCESS_COUNT ./optimizer 10000000000
```

分别令 PROCESS_COUNT 为 1、2、4、8、16，多次测试测量加速比，并分析原因。

5. 调试

我们可以在初始化 MPI 环境后，让进程 0 等待用户输入，并让各进程都调用 MPI_Barrier。这样，所有进程都会等待进程 0 得到用户输入，从而实现同步。然后，可以使用 gdb 调试 C++ 程序。

八、实验数据及结果分析：

1. 基准代码

代码见附件：base.cpp。

令 n 从 $1e2$ 逐渐增大到 $1e9$ ，记录每次得到的素数个数。

```
2020010801024@mpi-cu08-1:~$ mpirun -np 4 ./base 10
There are 4 primes less than or equal to 10
SIEVE (4) 0.000176
2020010801024@mpi-cu08-1:~$ mpirun -np 4 ./base 100
There are 25 primes less than or equal to 100
SIEVE (4) 0.000420
2020010801024@mpi-cu08-1:~$ mpirun -np 4 ./base 1000
There are 168 primes less than or equal to 1000
SIEVE (4) 0.000255
2020010801024@mpi-cu08-1:~$ mpirun -np 4 ./base 10000
There are 1229 primes less than or equal to 10000
SIEVE (4) 0.000284
2020010801024@mpi-cu08-1:~$ mpirun -np 16 ./base 100000
There are 9592 primes less than or equal to 100000
SIEVE (16) 0.007827
2020010801024@mpi-cu08-1:~$ mpirun -np 16 ./base 1000000
There are 78498 primes less than or equal to 1000000
SIEVE (16) 0.003450
2020010801024@mpi-cu08-1:~$ mpirun -np 16 ./base 10000000
There are 664579 primes less than or equal to 10000000
SIEVE (16) 0.017868
2020010801024@mpi-cu08-1:~$ mpirun -np 16 ./base 100000000
There are 5761455 primes less than or equal to 100000000
SIEVE (16) 0.452796
2020010801024@mpi-cu08-1:~$ mpirun -np 16 ./base 1000000000
There are 50847534 primes less than or equal to 1000000000
SIEVE (16) 8.556192
```

经过对比，结果完全正确。

令 $n=1e8$ ，进程数分别为 1、2、4、8、16，各测量 10 次时间取平均值，并计算加速比。

次数/进程数	1	2	4	8	16
1	7.392537	3.199804	1.607942	0.824565	0.410270
2	6.632556	3.205395	1.613894	0.806270	0.458490
3	6.490392	3.176319	1.657262	0.733116	0.389077
4	6.060084	3.172023	1.607012	0.812276	0.430497
5	6.046200	3.214307	1.621786	0.817257	0.418150
6	6.057211	3.183059	1.609708	0.701926	0.409465
7	6.066682	3.157331	1.625781	0.767014	0.396981
8	6.057636	3.181458	1.615426	0.716809	0.400642
9	6.046111	3.172787	1.618236	0.718302	0.417162
10	6.053371	3.151889	1.607378	0.720156	0.426543
平均	6.290278	3.181437	1.618443	0.761769	0.415728
加速比	1	1.977181	3.886624	8.25746	15.13076

可以看到，基准程序的加速比是亚线性的。这是由于在并程序下，每个进程都只要负责 $1/p$ 的范围，所以在理想情况下，加速比就是 p 。但是，基准程序中每个循环都有一次进程通信，所以带来了一定损耗。而且，进程 0 会做一些额外的工作（例如寻找下一个素数作为基数），所以这是负载不均衡的程序，这也会对加速比造成影响。

2. 去除偶数

代码见附件：optimizer-even.cpp。

令 n 从 $1e2$ 逐渐增大到 $1e9$ ，记录每次得到的素数个数。

```
2020010801024@mpi-cu08-1:~$ mpirun -np 4 ./optimizer-even 10
There are 4 primes less than or equal to 10
SIEVE (4) 0.000078
2020010801024@mpi-cu08-1:~$ mpirun -np 4 ./optimizer-even 100
There are 25 primes less than or equal to 100
SIEVE (4) 0.000270
2020010801024@mpi-cu08-1:~$ mpirun -np 16 ./optimizer-even 1000
There are 168 primes less than or equal to 1000
SIEVE (16) 0.000907
2020010801024@mpi-cu08-1:~$ mpirun -np 16 ./optimizer-even 10000
There are 1229 primes less than or equal to 10000
SIEVE (16) 0.001018
2020010801024@mpi-cu08-1:~$ mpirun -np 16 ./optimizer-even 100000
There are 9592 primes less than or equal to 100000
SIEVE (16) 0.000802
2020010801024@mpi-cu08-1:~$ mpirun -np 16 ./optimizer-even 1000000
There are 78498 primes less than or equal to 1000000
SIEVE (16) 0.001505
2020010801024@mpi-cu08-1:~$ mpirun -np 16 ./optimizer-even 10000000
There are 664579 primes less than or equal to 10000000
SIEVE (16) 0.006400
2020010801024@mpi-cu08-1:~$ mpirun -np 16 ./optimizer-even 100000000
There are 5761455 primes less than or equal to 100000000
SIEVE (16) 0.095905
2020010801024@mpi-cu08-1:~$ mpirun -np 16 ./optimizer-even 1000000000
There are 50847534 primes less than or equal to 1000000000
SIEVE (16) 4.061529
```

经过对比，结果完全正确。

令 $n=1e8$ ，进程数分别为 1、2、4、8、16，各测量 10 次时间取平均值，并计算加速比。

次数/进程数	1	2	4	8	16
1	2.006529	0.659517	0.394605	0.219036	0.135124
2	2.021164	1.042699	0.394794	0.247035	0.136291
3	2.020175	1.032690	0.398584	0.243563	0.113043
4	1.983785	0.654668	0.528927	0.201859	0.121712
5	2.010820	1.043591	0.319938	0.239010	0.122094
6	1.982430	0.726844	0.533407	0.203486	0.133071
7	1.983923	0.669598	0.394072	0.243132	0.117552
8	1.988702	0.637795	0.321870	0.238950	0.138085
9	2.007558	1.037961	0.401108	0.244780	0.113525
10	1.975357	1.030021	0.394810	0.239162	0.098982
平均	1.998044	0.853538	0.408212	0.232001	0.122948
加速比	1	2.340896	4.89463	8.612212	16.25115

可以看到，加速比是超线性的。这是因为：第一，每个进程都对自己的上下界进行了优化，让范围进一步缩小；第二，进程 0 寻找基数的范围也缩小了，负载不均衡的情况有所缓解。

3. 消除广播

代码见附件：optimizer-broadcast.cpp。

令 n 从 $1e2$ 逐渐增大到 $1e9$ ，记录每次得到的素数个数。

```
2020010801024@mpi-cu08-1:~$ mpirun -np 16 ./optimizer-broadcast 10
There are 4 primes less than or equal to 10
SIEVE (16) 0.000102
2020010801024@mpi-cu08-1:~$ mpirun -np 16 ./optimizer-broadcast 100
There are 25 primes less than or equal to 100
SIEVE (16) 0.000059
2020010801024@mpi-cu08-1:~$ mpirun -np 16 ./optimizer-broadcast 1000
There are 168 primes less than or equal to 1000
SIEVE (16) 0.000264
2020010801024@mpi-cu08-1:~$ mpirun -np 16 ./optimizer-broadcast 10000
There are 1229 primes less than or equal to 10000
SIEVE (16) 0.000067
2020010801024@mpi-cu08-1:~$ mpirun -np 16 ./optimizer-broadcast 100000
There are 9592 primes less than or equal to 100000
SIEVE (16) 0.000103
2020010801024@mpi-cu08-1:~$ mpirun -np 16 ./optimizer-broadcast 1000000
There are 78498 primes less than or equal to 1000000
SIEVE (16) 0.000356
2020010801024@mpi-cu08-1:~$ mpirun -np 16 ./optimizer-broadcast 10000000
There are 664579 primes less than or equal to 10000000
SIEVE (16) 0.005224
2020010801024@mpi-cu08-1:~$ mpirun -np 16 ./optimizer-broadcast 100000000
There are 5761455 primes less than or equal to 100000000
SIEVE (16) 0.098864
2020010801024@mpi-cu08-1:~$ mpirun -np 16 ./optimizer-broadcast 1000000000
There are 50847534 primes less than or equal to 1000000000
SIEVE (16) 4.012838
```

经过对比，结果完全正确。

令 $n=1e8$ ，进程数分别为 1、2、4、8、16，各测量 10 次时间取平均值，并计算加速比。

次数/进程数	1	2	4	8	16
1	2.377632	1.090592	0.404905	0.220542	0.114924
2	2.232601	1.181377	0.440029	0.218974	0.099856
3	2.240736	0.617356	0.303900	0.152511	0.102651
4	2.233836	0.618371	0.300458	0.162419	0.107883
5	2.268142	1.198396	0.407503	0.189579	0.104414
6	2.252477	0.605858	0.497668	0.186461	0.105943
7	2.245279	0.619137	0.374014	0.225609	0.109846
8	2.248496	0.603871	0.375162	0.189543	0.105774
9	2.259322	0.662408	0.403101	0.187098	0.096854
10	2.251374	0.637362	0.378815	0.190207	0.094847
平均	2.260990	0.783473	0.388556	0.192294	0.104299
加速比	1	2.885856	5.818962	11.75796	21.67792

可以看到，加速比是超线性的。这是因为广播的消除不仅加速了进程本身，也让进程 0 不再需要做额外的工作，从而使得各进程负载均衡。并且可以发现，在多处理器情况下，加速比有极为显著的提升——这正是消除广播的优势所在。并且可以想象到，这一优势在多机运行时会更加明显。

4. Cache 优化

代码见附件：optimizer-cache.cpp。

使用命令 `getconf LEVEL3_CACHE_SIZE` 得到 L3 缓存（经测试效果最佳）的大小。

```
2020010801024@mpi-cu08-1:~$ getconf LEVEL3_CACHE_SIZE
16777216
```

所以我们可以把每个进程的负责范围按 $16777216 / \text{sizeof}(\text{long long}) / \text{PROCESS_COUNT}$ 再划分块。

令 n 从 $1e2$ 逐渐增大到 $1e9$ ，记录每次得到的素数个数。

```
2020010801024@mpi-cu08-1:~$ mpirun -np 16 ./optimizer 10
There are 4 primes less than or equal to 10
SIEVE (16) 0.000367
2020010801024@mpi-cu08-1:~$ mpirun -np 16 ./optimizer 100
There are 25 primes less than or equal to 100
SIEVE (16) 0.000217
2020010801024@mpi-cu08-1:~$ mpirun -np 16 ./optimizer 1000
There are 168 primes less than or equal to 1000
SIEVE (16) 0.000587
2020010801024@mpi-cu08-1:~$ mpirun -np 16 ./optimizer 10000
There are 1229 primes less than or equal to 10000
SIEVE (16) 0.000661
2020010801024@mpi-cu08-1:~$ mpirun -np 16 ./optimizer 100000
There are 9592 primes less than or equal to 100000
SIEVE (16) 0.001845
2020010801024@mpi-cu08-1:~$ mpirun -np 16 ./optimizer 1000000
There are 78498 primes less than or equal to 1000000
SIEVE (16) 0.000661
2020010801024@mpi-cu08-1:~$ mpirun -np 16 ./optimizer 10000000
There are 664579 primes less than or equal to 10000000
SIEVE (16) 0.005727
2020010801024@mpi-cu08-1:~$ mpirun -np 16 ./optimizer 100000000
There are 5761455 primes less than or equal to 100000000
SIEVE (16) 0.055893
2020010801024@mpi-cu08-1:~$ mpirun -np 16 ./optimizer 1000000000
There are 50847534 primes less than or equal to 1000000000
SIEVE (16) 0.474660
```

经过对比，结果完全正确。

令 $n=1e8$ ，进程数分别为 1、2、4、8、16，各测量 10 次时间取平均值，并计算加速比。

次数/进程数	1	2	4	8	16
1	1.046073	0.520756	0.240131	0.110067	0.060916
2	1.046094	0.526247	0.244702	0.100094	0.058543
3	1.046145	0.521103	0.239511	0.099040	0.059053
4	1.047284	0.522284	0.241331	0.099101	0.056725
5	1.046727	0.519892	0.240888	0.099085	0.058395
6	1.046618	0.519882	0.240633	0.099746	0.057124
7	1.046194	0.520501	0.241246	0.098649	0.059275
8	1.046686	0.516647	0.252185	0.102006	0.055741
9	1.046711	0.523528	0.238133	0.102008	0.057476
10	1.046281	0.529634	0.238154	0.103521	0.054649
平均	1.046481	0.522047	0.241691	0.101332	0.057790
加速比	1	2.004571	4.329824	10.32728	18.10844

可以看到，加速比是超线性的。这是因为程序针对 L3 缓存进行了优化，每次迭代所需要的数字都能够命中 L3 缓存，大大增幅了程序性能。

5. 总结

为了保证程序的正确性，我们把各程序的输出与标准答案对比。

优化项/ $n=10^k$	1	2	3	4	5	6	7	8	9	10
标准答案	4	25	168	1229	9592	78948	664579	5761455	50847534	455052511
基准代码	4	25	168	1229	9592	78948	664579	5761455	50847534	报错
去除偶数	4	25	168	1229	9592	78948	664579	5761455	50847534	455052511
消除广播	4	25	168	1229	9592	78948	664579	5761455	50847534	455052511
Cache 优化	4	25	168	1229	9592	78948	664579	5761455	50847534	455052511

可以看到，三次优化后的代码，都可以得到正确的素数个数。基准代码由于性能原因，无法完成 $1e10$ 规模下的计算。

为了衡量优化的效果，我们在 $n=1e9$ 、 $p=16$ 情况下测试各程序。

次数/优化项	基准	去除偶数	消除广播	Cache 优化
1	8.653525	4.008930	3.915577	0.488920
2	8.701573	3.877280	3.863039	0.520445
3	8.698189	3.901094	3.761315	0.488071
4	8.615983	3.959148	3.945876	0.498802
5	8.697107	4.077742	3.994005	0.487373
6	8.740612	4.053366	4.005857	0.466756
7	8.634685	3.930093	3.961022	0.466249
8	8.462030	3.902477	3.679082	0.483379
9	8.771933	4.153551	3.779053	0.477851
10	8.677119	3.999531	3.738401	0.468095
平均	8.665276	3.986321	3.864323	0.484594
加速比	1	2.173752	2.242379	17.881513

相较于基准代码：

- 去除偶数的性能为 2.17 倍，因为所有的偶数都被忽略了。
- 消除广播的性能为 2.24 倍，因为消除了广播，但每个进程都引入了额外的素数计算。
- Cache 优化的性能为 17.88 倍，因为每次循环的计算都在 L3 缓存内发生。

九、实验结论：

三次优化后的程序都能正确地计算素数个数。从 $1e2$ 到 $1e10$ ，答案分别是：4、25、168、1229、9592、78948、664579、5761455、50847534、455052511。

基准代码的加速比是亚线性的，三次优化的加速比都是超线性的。

在优化的性能提升上，去除偶数的性能是基准代码的 2.17 倍，消除广播是 2.24 倍，均在预期之内。Cache 优化是 17.88 倍，说明最终优化的程序获得了极大的性能提升。

综上，本次实验编写的代码是正确且高性能的。

十、总结及心得体会：

通过本次实验，我对 MPI 库的使用有了亲身的实践和体验，初步了解了怎样写出美观、健壮、高性能的并行代码。

我通过基准代码理解了埃拉托斯特尼筛法的思想，以及它如何被应用到分布式计算的场景下。

通过三次优化，我知道了分布式程序调优的不易，了解了进程通信对性能的影响，掌握了调优的基本方法。尤其是 Cache 优化，我亲身体会到了合理利用缓存对程序的性能会带来多大的提升。

十一、对本实验过程及方法、手段的改进建议：

实验指导书内的基准代码可以更新下，有些变量需要改成 long long，以适应 $1e10$ 的数据规模。

学生签名：  与望

报告评分：

指导教师签字：