电 子 科 技 大 学

实 验 报 告

学生姓名：程乾宇	学 号：2018270101009
一、实验室名称：无	
二、实验项目名称：Eratosthenes 素数筛选算法并行及性能优化	

三、实验原理：

Eratosthenes 筛法原理：

Eratosthenes 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。

这里，我们把Ｎ取 120 来举例说明埃拉托斯特尼筛法思想：

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 9、11、13、17、19、23、29、31、37、41、43、47…将它们分别画圈，并划去它们的倍数（可以看到，已经没有这样的数了

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

数据分配：

聚合原始任务后，一个任务将负责一组数据。我们选择使用数据块分配方法进行分组。

方法 1：

当 n(总元素数量)%p(进程数)等于 0 时，每个进程分配 n/p 空间大小。

当 n(总元素数量)%p(进程数)不等于 0 时，令 r=n%p,则前 r 个进程数据长度为 n/p+1，后 n-r 个进程数据长度为 n/p。

进程 i 的第一个元素：i*(n/p)+min(i,r)

进程 i 的最后一个元素：(i+1)*(n/p)+min(i+1,r)-1

给定元素 j 属于哪个进程：min(j/(n/p+1),(j-r)/(n/p))

方法 2:

进程 i 的第一个元素：i*n/p

进程 i 的最后一个元素：(i+1)*n/p-1

给定元素 j 属于哪个进程：(p*(j+1)-1)/n

并行程序执行过程:

定义一个标记数组 marked，每一个元素的下标对应一个整数，它的值表示这个整数是否为素数，值为 1 是素数，值为 0 不是素数。

先假定所有的数都是素数，将 marked 数组置 0。

选定第一个整数 2，从它对应的数组元素 2*2=4 开始依次标记 2 的倍数，一直标记到最后一个数为止。

接下来选定下一个未标记的数，可以保证它一定是素数。使用广播的形式通知各进程标记这个素数的倍数。这样循环到最后，所有进程中未标记的数之和就是 1-n 中的所有素数了。

四、实验目的：

1. 使用 MPI 编程实现 Eratosthenes 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.Eratosthenes 筛法实现

2.并行程序的优化

具体评分要求如下：

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（40 分）

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（10 分）

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（15 分）

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（10 分）。

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。根据全班优化 3 在目标机上实测性能，最高性能（最短执行时间）得分 25 分，最低性能得 0 分，其他按执行时间进行插值。（25 分）

六、实验器材（设备、元器件）：

操作系统：macOS 11.2.2

CPU: i5-8259U 4 Core (L2 Cache 4x256KB, L3 Cache 1x6MB)

软件包版本：OpenMPI 4.1.0, Clang GCC 11.1.0

七、实验步骤及操作：

1. MPI 环境部署

在终端运行：

$ brew install open-mpi

安装 OpenMPI.

我们使用 CMake 构建项目，在项目中使用 OpenMPI 软件包时，只需向 CMakeLists.txt 写入以下函数：

```
find_package(MPI REQUIRED)
set(MPI_SKIP_MPICXX true)
include_directories(${MPI_CXX_INCLUDE_DIRS})
```

即可完成寻找软件包，加入宏定义选项，包含头文件等工作。在编译可执行文件时，需要添加以下函数：

```
target_link_libraries(MPI_EratosthenesSieve.${mainname} ${MPI_CXX_LIBRARIES})
```

以实现与 MPI 库链接的工作。

2. 基准代码

使用 mpicxx -o base base.cpp 手动编译程序以同时链接 MPI 库。对本机项目来说，还可以使用 cmake . && make 的方式进行编译。

使用 mpirun -np <p> base <n>执行程序。p 定义并发进程数，n 定义筛法执行的数据范围。执行结果如下：

```
➜  cmake-build-debug mpirun -n 1 ./MPI_EratosthenesSieve.base 100
There are -473019208 primes less than or equal to 100
SIEVE (1)   0.000024
➜  cmake-build-debug mpirun -n 2 ./MPI_EratosthenesSieve.base 100
There are 25 primes less than or equal to 100
SIEVE (2)   0.000087
```

两次数据范围相同，但执行结果不同。由此我们发现代码中有一处错误：

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

这里对所有进程的 count 全局求和的过程，只适用于并发程序，而 p=1 时则会导致 global_count 未初始化输出。

有两种修改方式：一种是取消判断，单进程时也进行广播；二是在单进程是，直接将 count

赋值到全局求和变量。

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

```
MPI_Reduce(&count, &global_count, count: 1, MPI_INT, MPI_SUM, root: 0, MPI_COMM_WORLD);
```

修改完成后，运行结果正常。

```
➜ cmake-build-debug mpirun -n 2 ./MPI_EratosthenesSieve.base 100
There are 25 primes less than or equal to 100
SIEVE (2)   0.000126
➜ cmake-build-debug mpirun -n 1 ./MPI_EratosthenesSieve.base 100
There are 25 primes less than or equal to 100
SIEVE (1)   0.000020
```

同时，为了避免计算范围溢出导致无法正常分配内存，我们强制将 low_value 与 high_value 在计算过程中强制转为 long long 类型。

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

现在对基准代码进行测试。

数据范围取 1000、10000、…、10^9 进行测试，进程数取 1、2、4、8、16，测试机为 4 核，超线程可执行 8 线程，所以 16 进程执行时程序性能一般会退化。实验数据放在结果分析部分进行展示讨论。

3. 优化 1：去除偶数

偶数必然是 2 的倍数，除了 2 本身以外，必然不是素数，所以所有偶数都可以不必列入标记数组进行计算。

基于上述思路对代码进行优化。主要调整了数值到 marked 数组索引的映射过程：

```
#define ODD_TO_INDEX(value) (((value)-3)/2)
#define INDEX_TO_ODD(index) (2*(index)+3)
```

marked 数组从 3 开始进行标记。

具体代码已列入附录，请参考。实验数据放在结果分析部分进行展示讨论。

4.优化 2：预处理素数消除广播

我们让每个进程都各自找出它们的前 sqrt(n)个数中的素数，在通过这些素数筛选剩下的素数，这样一来进程之间就不需要每个循环广播素数了，性能得到提高。

为每个进程分配空间大小为 sqrt(n)+1 的 primes 数组，查找自 3 开始到 sqrt(n)的素数。之后素数筛选的过程中从 primes 数组中选取下一个素数。

具体代码已列入附录，请参考。实验数据放在结果分析部分进行展示讨论。

5. 优化 3：重构循环，提高 Cache 命中率

在新程序中，我们将素数枚举循环与全局标记循环的顺序颠倒，在一个小的数据范围也就是一个 chunk 内进行素数枚举。

由于数组是一段连续内存空间，具有空间上的局部性，同时在小范围内进行素数枚举也具有时间局部性。因此将所有并发执行的进程中的 chunk 大小之和控制为与 Cache 大小持平，就可以保证 Cache 的命中率。

考虑 Cache 空间与总线延迟的平衡，我们选择 L2 Cache 大小作为确定 chunk 大小的标准。测试机有 1MB L2 Cache，需要分配给数个进程同时保证命中率，由于数组是 char 类型数组，所以我们将 chunk 的大小设置为 2^20/p.

具体代码已列入附录，请参考。实验数据放在结果分析部分进行展示讨论。

6. 优化 4：进一步优化

减少判断语句，利用位运算加速。

八、实验数据及结果分析：

不同数据范围下的程序执行时间如下。使用了五个测试程序：基准程序 base，优化后程序 optimizer1~4。测试时在每个实验条件下重复实验四次，取运行时间均值。测试的数据处理规模范围 N=10^3~10^9，并发进程规模范围 P=2^1~2^4，同时测试了无并发 P=1 的标准情况。

N=10^3 / P	base	optimizer1	optimizer2	optimizer3	optimizer4
1	0.000039	0.000026	0.0000295	0.00002575	0.00002025
2^1	0.0008075	0.00062225	0.00004525	0.00004875	0.0000445
2^2	0.000831	0.0008655	0.000096	0.00008475	0.00008125
2^3	0.00223375	0.0017955	0.00019525	0.000143	0.00018275
2^4	0.0018775	0.00190525	0.00031025	0.0003075	0.0002365

N=10^4 / P	base	optimizer1	optimizer2	optimizer3	optimizer4
1	0.00010325	0.0000655	0.00008225	0.000067	0.00002525
2^1	0.000849	0.00080625	0.00007275	0.00007025	0.0000485
2^2	0.001483	0.0014665	0.00010025	0.00011375	0.00010075
2^3	0.00188975	0.00194775	0.00014525	0.000171	0.00013175
2^4	0.0024275	0.0022745	0.00020725	0.0002645	0.00034475

N=10^5 / P	base	optimizer1	optimizer2	optimizer3	optimizer4
1	0.00086325	0.000461	0.00046375	0.000501	0.000086
2^1	0.00130025	0.0010845	0.00026025	0.000303	0.0000715
2^2	0..001784	0.00197	0.00019825	0.0002125	0.0000865
2^3	0.00210228	0.001921	0.0001815	0.0002365	0.0001955
2^4	0.00265575	0.0023525	0.00031275	0.0002455	0.000194

N=10^6 / P	base	optimizer1	optimizer2	optimizer3	optimizer4
1	0.00978825	0.0042445	0.004355	0.0045685	0.0009445
2^1	0.0048455	0.002973	0.002254	0.0024775	0.000434
2^2	0.00414	0.00247425	0.0011215	0.0012265	0.0002755
2^3	0.00373075	0.00295075	0.00093	0.003913	0.00032275
2^4	0.003386	0.00321875	0.000471	0.00078175	0.000444

N=10^7 P	base	optimizer1	optimizer2	optimizer3	optimizer4
2^0	0.107641	0.049987	0.0451995	0.04932875	0.0121085
2^1	0.064249	0.033552	0.02380425	0.02411875	0.00572375
2^2	0.036131	0.01868275	0.0121945	0.01020865	0.00271125
2^3	0.0269175	0.0109445	0.01034225	0.00910475	0.00213425
2^4	0.021014	0.0100255	0.00317033	0.00658875	0.0009675

N=10^8 P	base	optimizer1	optimizer2	optimizer3	optimizer4
2^0	1.35914725	0.64193375	0.69807475	0.53993525	0.13712325
2^1	0.92479975	0.4476885	0.49953	0.2699095	0.068918
2^2	0.43025475	0.21648425	0.230031	0.13183825	0.03005475
2^3	0.356534	0.16155075	0.15024275	0.09656	0.0218285
2^4	0.315409i75	0.140151	0.1022945	0.08898625	0.0104775

N=10^9 P	base	optimizer1	optimizer2	optimizer3	optimizer4
2^0	16.5177103	8.243362	8.1288225	5.458505	1.5833455
2^1	10.8646868	5.35775975	5.32450275	2.668817	0.7708295
2^2	8.7527155	4.293643	4.310239	1.310793	0.330479
2^3	8.373474	4.05732075	4.038363	0.9900815	0.25192075
2^4	8.315155	3.95956825	3.8680635	1.02110675	0.23922275

以无并行程序的运行时间为基准，计算每个数据规模下，每个测试程序在不同并发进程规模下的**加速比**。

N=10^3 P	base	optimizer1	optimizer2	optimizer3	optimizer4
2^0	1	1	1	1	1
2^1	0.04829721	0.04178385	0.6519337	0.52820513	0.45505618
2^2	0.04693141	0.03004044	0.30729167	0.30383481	0.24923077
2^3	0.01745943	0.01448065	0.15108835	0.18006993	0.11080711
2^4	0.0207723	0.0136465	0.09508461	0.08373984	0.08562368

N=10^4 P	base	optimizer1	optimizer2	optimizer3	optimizer4
2^0	1	1	1	1	1
2^1	0.12161366	0.08124031	1.13058419	0.95373665	0.52061856
2^2	0.06962239	0.04466417	0.82044888	0.58901099	0.25062035
2^3	0.05463686	0.03362855	0.56626506	0.39181287	0.19165085
2^4	0.04253347	0.02879754	0.39686369	0.25330813	0.07324148

N=10^5 P	base	optimizer1	optimizer2	optimizer3	optimizer4
2^0	1	1	1	1	1
2^1	0.66391079	0.42508068	1.78194044	1.65346535	1.2027972
2^2	0.48388453	0.23401015	2.33921816	2.35764706	0.99421965
2^3	0.41062658	0.23997918	2.55509642	2.11839323	0.4398977
2^4	0.32504942	0.19596174	1.48281375	2.0407332	0.44329897

P \ N=10^6	base	optimizer1	optimizer2	optimizer3	optimizer4
2^0	1	1	1	1	1
2^1	2.02007017	1.42768248	1.93212067	1.84399596	2.17626728
2^2	2.36431159	1.71546933	3.88319215	3.72482674	3.42831216
2^3	2.62366816	1.43844785	4.6827957	1.16751853	2.92641363
2^4	2.89080035	1.31867961	9.2462845	5.84393988	2.12725225

P \ N=10^7	base	optimizer1	optimizer2	optimizer3	optimizer4
2^0	1	1	1	1	1
2^1	1.67537238	1.48983667	1.89879958	2.04524488	2.11548373
2^2	2.97918685	2.67556971	3.70654803	4.83205419	4.46602121
2^3	3.99892263	4.56731692	4.37037395	5.41791373	5.67342158
2^4	5.12234701	4.98598574	14.2570557	7.48681465	12.5152455

P \ N=10^8	base	optimizer1	optimizer2	optimizer3	optimizer4
2^0	1	1	1	1	1
2^1	1.46966654	1.43388483	1.39746312	2.0004307	1.989658
2^2	3.15893607	2.96526768	3.03469858	4.09543702	4.56244853
2^3	3.81211119	3.97357332	4.64631238	5.59170723	6.28184484
2^4	4.30914786	4.58030089	6.82416699	6.06762562	13.0874016

N=10^3 / P	base	optimizer1	optimizer2	optimizer3	optimizer4
2^0	1	1	1	1	1
2^1	1.52031169	1.53858373	1.526682	2.0452901	2.05408005
2^2	1.88715265	1.91989926	1.88593312	4.16427689	4.79106237
2^3	1.97262334	2.03172549	2.0129004	5.51318755	6.28509363
2^4	1.98645849	2.08188405	2.10152251	5.34567517	6.61870788

可以注意到，在小数据规模下（N=10^3~10^5），并发程序的加速比反而小于 1，速度不增反降，并且随着并发数的增加，程序的加速比更小。这主要是因为在小规模数据下创建多个进程的开销，相比与其对程序执行的加速效果来说更为沉重。

而在大数据规模下（N=10^6~10^9），基准程序加速比最高可以达到 4~5，这主要是受核心数量的限制，测试机核心数量为四核。而利用 Cache 特性优化后的程序，其加速比最高可以达到 12~13，随着数据规模的增加，Cache 基本命中导致其内容替换次数较少所带来的速度红利逐渐显现。

去除偶数后，优化程序与基准程序相比同等并发规模下速度大致都快了一半，这是因为处理的数据量刚好少了一半。与此相比，消除广播的优化效果在并发规模较小的情况下不那么明显，这主要是因为增加了素数预处理的开销，其优化效果只有在并发规模较大时才能显现。而提高 Cache 命中的优化只有在大数据规模下才能显现，这主要是是因为只有在大数据规模下 Cache 才会大量触发未命中，同时在这种情况下优化程序中才能充分划分数组，避免大量的 Cache miss。

九、实验结论：

并发程序对大规模问题处理的加速来说是行之有效的。

实现了对原程序的充分优化，与基准程序相比其最高加速比达到 13.09。

对性能优化进行了量化分析。

十、总结及心得体会：

优化并行程序时，要考虑算法的结构，消息传递的开销设计，以及硬件特性。

并行程序需要量化分析性能和大量测试。

十一、对本实验过程及方法、手段的改进建议：

建议可以实时评测。

报告评分：

指导教师签字：

附：代码

base.cpp

```
#include "mpi.h"
#include <math.h>
#include <stdio.h>

#define MIN(a, b)  ((a)<(b)?(a):(b))

int main(int argc, char *argv[]) {
    int count;        /* Local prime count */
    double elapsed_time; /* Parallel execution time */
    int first;        /* Index of first multiple */
    int global_count; /* Global prime count */
    int high_value;   /* Highest value on this proc */
    int i;
    int id;           /* Process ID number */
    int index;        /* Index of current prime */
    int low_value;    /* Lowest value on this proc */
    char *marked;     /* Portion of 2,...,'n' */
    int n;            /* Sieving from 2, ..., 'n' */
```

```
int p;           /* Number of processes */
int proc0_size;   /* Size of proc 0's subarray */
int prime;        /* Current prime */
int size;         /* Elements in 'marked' */

freopen("/dev/null","w",stderr);
MPI_Init(&argc, &argv);

/* Start the timer */

MPI_Comm_rank(MPI_COMM_WORLD, &id);
MPI_Comm_size(MPI_COMM_WORLD, &p);
MPI_Barrier(MPI_COMM_WORLD);
elapsed_time = -MPI_Wtime();

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

n = atoi(argv[1]);

/* Figure out this process's share of the array, as
   well as the integers represented by the first and
   last array elements */

low_value = 2 + (long long)id * (n - 1) / p;
high_value = 1 + (long long)(id + 1) * (n - 1) / p;
size = high_value - low_value + 1;

/* Bail out if all the primes used for sieving are
   not all held by process 0 */

proc0_size = (n - 1) / p;

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

/* Allocate this process's share of the array. */

marked = (char *) malloc(size);
```

```
    if (marked == NULL) {
        printf("Cannot allocate enough memory\n");
        MPI_Finalize();
        exit(1);
    }
    for (i = 0; i < size; i++) marked[i] = 0;

    if (!id) index = 0;
    prime = 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);
        }
        for (i = first; i < size; i += prime) marked[i] = 1;
        if (!id) {
            while (marked[++index]);
            prime = index + 2;
        }
        if (p > 1) MPI_Bcast(&prime, 1, MPI_INT, 0, MPI_COMM_WORLD);
    } while (prime * prime <= n);

    count = 0;
    for (i = 0; i < size; i++)
        if (!marked[i]) count++;
    MPI_Reduce(&count, &global_count, 1, MPI_INT, MPI_SUM,0,
MPI_COMM_WORLD);

    /* Stop the timer */

    elapsed_time += MPI_Wtime();

    /* Print the results */

    if (!id) {
        printf("There are %d primes less than or equal to %d\n",
global_count, n);
        printf("SIEVE (%d) %10.6f\n", p, elapsed_time);
    }
    MPI_Finalize();
    return 0;
}
```

optimizer1.cpp

```
#include "mpi.h"
#include <math.h>
#include <stdio.h>

#define MIN(a, b) ((a)<(b) ? (a) : (b))
#define BLOCK_LOW(id, p, n) ((long long)(id)*(n)/(p))
#define BLOCK_HIGH(id, p, n) (BLOCK_LOW((id)+1,p,n)-1)
#define BLOCK_SIZE(id, p, n) (BLOCK_HIGH(id,p,n)-BLOCK_LOW(id,p,n)+1)
#define BLOCK_OWNER(index, p, n) (((p)*(index)+1)-1)/(n))
#define VALUE_TO_INDEX(value) (((value)-2)
#define INDEX_TO_VALUE(index) ((index)+2)
#define ODD_TO_INDEX(value) (((value)-3)/2) //奇数值转索引值
#define INDEX_TO_ODD(index) (2*(index)+3)   //索引值转奇数值

int main(int argc, char *argv[]) {
    int count;          /* Local prime count */
    double elapsed_time; /* Parallel execution time */
    int first;          /* Index of first multiple */
    int global_count;  /* Global prime count */
    int high_value;    /* Highest value on this proc */
    int i;
    int id;            /* Process ID number */
    int index;          /* Index of current prime */
    int low_value;     /* Lowest value on this proc */
    char *marked;      /* Portion of 2,...,'n' */
    int n;             /* Sieving from 2, ..., 'n' */
    int p;             /* Number of processes */
    int proc0_size;    /* Size of proc 0's subarray */
    int prime;          /* Current prime */
    int size;           /* Elements in 'marked' */
    int m;             /* Size of search list */
    int offset;

    freopen("/dev/null","w",stderr);
    MPI_Init(&argc, &argv);

    /* Start the timer */

    MPI_Comm_rank(MPI_COMM_WORLD, &id);
    MPI_Comm_size(MPI_COMM_WORLD, &p);
    MPI_Barrier(MPI_COMM_WORLD);
    elapsed_time = -MPI_Wtime();
```

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

n = atoi(argv[1]);
m = ODD_TO_INDEX(n) + 1; // odds in 3..n

/* Figure out this process's share of the array, as
   well as the integers represented by the first and
   last array elements */

low_value = INDEX_TO_ODD(BLOCK_LOW(id, p, m));
high_value = INDEX_TO_ODD(BLOCK_HIGH(id, p, m));
size = BLOCK_SIZE(id, p, m);

/* Bail out if all the primes used for sieving are
   not all held by process 0 */

proc0_size = m / p;

if (INDEX_TO_ODD(proc0_size - 1) < (int) sqrt((double) n)) {
    if (!id) printf("Too many processes\n");
    MPI_Finalize();
    exit(1);
}

/* Allocate this process's share of the array. */

marked = (char *) malloc(size);

if (marked == NULL) {
    printf("Cannot allocate enough memory\n");
    MPI_Finalize();
    exit(1);
}
for (i = 0; i < size; i++) marked[i] = 0;

if (!id) index = 0;
prime = 3;
do {
    if (prime * prime > low_value)
        first = ODD_TO_INDEX(prime * prime) - ODD_TO_INDEX(low_value);
    else {
```

```
            if (!(low_value % prime)) first = 0;
            else {
                first = prime - (low_value % prime);
                if (!((low_value + first) & 1)) // if odd
                    first += prime;
                first >>= 1;
            }
        }
        for (i = first; i < size; i += prime) marked[i] = 1;
        if (!id) {
            while (marked[++index]);
            prime = INDEX_TO_ODD(index);
        }
        if (p > 1) MPI_Bcast(&prime, 1, MPI_INT, 0, MPI_COMM_WORLD);
    } while (prime * prime <= n);

    count = 0;
    for (i = 0; i < size; i++) {
        if (!marked[i])
            count++;
    }
    MPI_Reduce(&count, &global_count, 1, MPI_INT, MPI_SUM, 0,
MPI_COMM_WORLD);

    /* Stop the timer */

    elapsed_time += MPI_Wtime();

    /* Print the results */

    if (!id) {
        printf("There are %d primes less than or equal to %d\n",
global_count + 1, n);
        printf("SIEVE (%d) %10.6f\n", p, elapsed_time);
    }
    MPI_Finalize();
    return 0;
}
```

optimizer2.cpp

```
#include "mpi.h"
#include <math.h>
#include <stdio.h>

#define MIN(a, b) ((a)<(b) ? (a) : (b))
#define BLOCK_LOW(id, p, n) ((long long)(id)*(n)/(p))
#define BLOCK_HIGH(id, p, n) (BLOCK_LOW((id)+1,p,n)-1)
#define BLOCK_SIZE(id, p, n) (BLOCK_HIGH(id,p,n)-BLOCK_LOW(id,p,n)+1)
#define BLOCK_OWNER(index, p, n) (((p)*(index)+1)-1)/(n))
#define VALUE_TO_INDEX(value) (((value)-2)
#define INDEX_TO_VALUE(index) ((index)+2)
#define ODD_TO_INDEX(value) (((value)-3)/2)
#define INDEX_TO_ODD(index) (2*(index)+3)

int main(int argc, char *argv[]) {
   int count;          /* Local prime count */
   double elapsed_time; /* Parallel execution time */
   int first;          /* Index of first multiple */
   int global_count;  /* Global prime count */
   int high_value;    /* Highest value on this proc */
   int i;
   int id;            /* Process ID number */
   int index;         /* Index of current prime */
   int low_value;     /* Lowest value on this proc */
   char *marked;      /* Portion of 2,...,'n' */
   int n;             /* Sieving from 2, ..., 'n' */
   int p;             /* Number of processes */
   int proc0_size;    /* Size of proc 0's subarray */
   int prime;         /* Current prime */
   int size;          /* Elements in 'marked' */
   int m;             /* Size of search list */
   char *primes;      /* Preprocessed primes */
   int primes_size;   /* Elements in 'primes' */

   freopen("/dev/null","w",stderr);
   MPI_Init(&argc, &argv);

   /* Start the timer */

   MPI_Comm_rank(MPI_COMM_WORLD, &id);
   MPI_Comm_size(MPI_COMM_WORLD, &p);
   MPI_Barrier(MPI_COMM_WORLD);
   elapsed_time = -MPI_Wtime();
```

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

n = atoi(argv[1]);
m = ODD_TO_INDEX(n) + 1;

/* Figure out this process's share of the array, as
   well as the integers represented by the first and
   last array elements */

low_value = INDEX_TO_ODD(BLOCK_LOW(id, p, m));
high_value = INDEX_TO_ODD(BLOCK_HIGH(id, p, m));
size = BLOCK_SIZE(id, p, m);

/* Bail out if all the primes used for sieving are
   not all held by process 0 */

proc0_size = m / p;

if (INDEX_TO_ODD(proc0_size - 1) < (int) sqrt((double) n)) {
    if (!id) printf("Too many processes\n");
    MPI_Finalize();
    exit(1);
}

/* Allocate this process's share of the array. */

marked = (char *) malloc(size);
if (marked == NULL) {
    printf("Cannot allocate enough memory\n");
    MPI_Finalize();
    exit(1);
}
for (i = 0; i < size; i++) marked[i] = 0;

primes_size = ODD_TO_INDEX(sqrt(n)) + 1;
primes = (char *) malloc(primes_size);
if (primes == NULL) {
    printf("Cannot allocate enough memory\n");
    free(marked);
    MPI_Finalize();
```

```c
        exit(1);
    }
    for (i = 0; i < primes_size; i++) primes[i] = 0;

    /* preprocess primes in 3..sqrt(n) */
    index = 0;
    prime = 3;
    do {
        for (i = ODD_TO_INDEX(prime * prime); i < primes_size; i += prime)
            primes[i] = 1;
        while (primes[++index]);
        prime = INDEX_TO_ODD(index);
    } while (prime * prime <= sqrt(n));

    index = 0;
    prime = 3;
    do {
        if (prime * prime > low_value)
            first = ODD_TO_INDEX(prime * prime) - ODD_TO_INDEX(low_value);
        else {
            if (!(low_value % prime)) first = 0;
            else {
                first = prime - (low_value % prime);
                if (!((low_value + first) & 1))
                    first += prime;
                first >>= 1;
            }
        }
        for (i = first; i < size; i += prime) marked[i] = 1;
        while (primes[++index]);
        prime = INDEX_TO_ODD(index);
    } while (prime * prime <= n);

    count = 0;
    for (i = 0; i < size; i++) {
        if (!marked[i])
            count++;
    }
    MPI_Reduce(&count, &global_count, 1, MPI_INT, MPI_SUM, 0,
MPI_COMM_WORLD);

    /* Stop the timer */

    elapsed_time += MPI_Wtime();
```

```c
    /* Print the results */

    if (!id) {
        printf("There are %d primes less than or equal to %d\n",
global_count + 1, n);
        printf("SIEVE (%d) %10.6f\n", p, elapsed_time);
    }
    MPI_Finalize();
    return 0;
}
```

optimizer3.cpp

```
#include "mpi.h"
#include <math.h>
#include <stdio.h>

#define MIN(a, b) ((a)<(b) ? (a) : (b))
#define BLOCK_LOW(id, p, n) ((long long)(id)*(n)/(p))
#define BLOCK_HIGH(id, p, n) (BLOCK_LOW((id)+1,p,n)-1)
#define BLOCK_SIZE(id, p, n) (BLOCK_HIGH(id,p,n)-BLOCK_LOW(id,p,n)+1)
#define BLOCK_OWNER(index, p, n) (((p)*(index)+1)-1)/(n))
#define VALUE_TO_INDEX(value) (((value)-2)
#define INDEX_TO_VALUE(index) ((index)+2)
#define ODD_TO_INDEX(value) (((value)-3)/2)
#define INDEX_TO_ODD(index) (2*(index)+3)

int main(int argc, char *argv[]) {
    int count;          /* Local prime count */
    double elapsed_time; /* Parallel execution time */
    int first;          /* Index of first multiple */
    int global_count;  /* Global prime count */
    int high_value;    /* Highest value on this proc */
    int i, j;
    int id;            /* Process ID number */
    int index;          /* Index of current prime */
    int low_value;      /* Lowest value on this proc */
    char *marked;       /* Portion of 2,...,'n' */
    int n;             /* Sieving from 2, ..., 'n' */
    int p;             /* Number of processes */
    int proc0_size;    /* Size of proc 0's subarray */
    int prime;          /* Current prime */
    int size;           /* Elements in 'marked' */
    int m;             /* Size of search list */
    char *primes;       /* Preprocessed primes */
    int primes_size;   /* Elements in 'primes' */
    int chunk;          /* chunk size for *marked to adapt cache size */
    int low_value_chunk; /* Lowest value in a chunk */

    freopen("/dev/null","w",stderr);
    MPI_Init(&argc, &argv);

    /* Start the timer */

    MPI_Comm_rank(MPI_COMM_WORLD, &id);
    MPI_Comm_size(MPI_COMM_WORLD, &p);
```

```c
chunk = (7 << 20) / p;  // 256KBX14 L2 CacheX2=7MB
MPI_Barrier(MPI_COMM_WORLD);
elapsed_time = -MPI_Wtime();

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

n = atoi(argv[1]);
m = ODD_TO_INDEX(n) + 1;

/* Figure out this process's share of the array, as
   well as the integers represented by the first and
   last array elements */

low_value = INDEX_TO_ODD(BLOCK_LOW(id, p, m));
high_value = INDEX_TO_ODD(BLOCK_HIGH(id, p, m));
size = BLOCK_SIZE(id, p, m);

/* Bail out if all the primes used for sieving are
   not all held by process 0 */

proc0_size = m / p;

if (INDEX_TO_ODD(proc0_size - 1) < (int) sqrt((double) n)) {
    if (!id) printf("Too many processes\n");
    MPI_Finalize();
    exit(1);
}

/* Allocate this process's share of the array. */

marked = (char *) malloc(size);
if (marked == NULL) {
    printf("Cannot allocate enough memory\n");
    MPI_Finalize();
    exit(1);
}
for (i = 0; i < size; i++) marked[i] = 0;

primes_size = ODD_TO_INDEX(sqrt(n)) + 1;
primes = (char *) malloc(primes_size);
if (primes == NULL) {
```

```
        printf("Cannot allocate enough memory\n");
        free(marked);
        MPI_Finalize();
        exit(1);
    }
    for (i = 0; i < primes_size; i++) primes[i] = 0;

    /* preprocess primes in 3..sqrt(n) */
    index = 0;
    prime = 3;
    do {
        for (i = ODD_TO_INDEX(prime * prime); i < primes_size; i += prime)
            primes[i] = 1;
        while (primes[++index]);
        prime = INDEX_TO_ODD(index);
    } while (prime * prime <= sqrt(n));

    for(i = 0; i < size; i += chunk){   // chunking
        index = 0;
        prime = 3;
        low_value_chunk = INDEX_TO_ODD(ODD_TO_INDEX(low_value) + i);
            do {
                if (prime * prime > low_value_chunk)
                    first = ODD_TO_INDEX(prime * prime) -
ODD_TO_INDEX(low_value_chunk);
                else {
                    if (!(low_value_chunk % prime)) first = 0;
                    else {
                        first = prime - (low_value_chunk % prime);
                        if (!((low_value_chunk + first) & 1))
                            first += prime;
                        first >>= 1;
                    }
                }
                for (j = first + i; j < first + i + chunk && j < size; j +=
prime)  // update in first+i..min(first+i+chunk-1, size-1)
                    marked[j] = 1;
                while (primes[++index]);
                prime = INDEX_TO_ODD(index);
            } while (prime * prime <= n);
    }

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

```c
            count++;
    }
    MPI_Reduce(&count, &global_count, 1, MPI_INT, MPI_SUM, 0,
MPI_COMM_WORLD);

    /* Stop the timer */

    elapsed_time += MPI_Wtime();

    /* Print the results */

    if (!id) {
        printf("There are %d primes less than or equal to %d\n",
global_count + 1, n);
        printf("SIEVE (%d) %10.6f\n", p, elapsed_time);
    }
    MPI_Finalize();
    return 0;
}
```

optimizer4.cpp

```
#include "mpi.h"
#include <math.h>
#include <stdio.h>

#define BLOCK_LOW(id, p, n)  ((long long)(id)*(n)/(p))
#define BLOCK_HIGH(id, p, n)  (BLOCK_LOW((id)+1,p,n)-1)
#define BLOCK_SIZE(id, p, n)  (BLOCK_HIGH(id,p,n)-BLOCK_LOW(id,p,n)+1)
#define ODD_TO_INDEX(value)  (((value)-3)>>1)
#define INDEX_TO_ODD(index)  (((index)<<1)+3)

//实验没说不给用 pragma 里的 Ofast 和加速指令!
#pragma GCC optimize(3)
#pragma GCC optimize("Ofast")
#pragma GCC optimize("inline")
#pragma GCC optimize("-fgcse")
#pragma GCC optimize("-fgcse-lm")
#pragma GCC optimize("-fipa-sra")
#pragma GCC optimize("-ftree-pre")
#pragma GCC optimize("-ftree-vrp")
#pragma GCC optimize("-fpeephole2")
#pragma GCC optimize("-ffast-math")
#pragma GCC optimize("-fsched-spec")
#pragma GCC optimize("unroll-loops")
#pragma GCC optimize("-falign-jumps")
#pragma GCC optimize("-falign-loops")
#pragma GCC optimize("-falign-labels")
#pragma GCC optimize("-fdevirtualize")
#pragma GCC optimize("-fcaller-saves")
#pragma GCC optimize("-fcrossjumping")
#pragma GCC optimize("-fthread-jumps")
#pragma GCC optimize("-funroll-loops")
#pragma GCC optimize("-freorder-blocks")
#pragma GCC optimize("-fschedule-insns")
#pragma GCC optimize("inline-functions")
#pragma GCC optimize("-ftree-tail-merge")
#pragma GCC optimize("-fschedule-insns2")
#pragma GCC optimize("-fstrict-aliasing")
#pragma GCC optimize("-falign-functions")
#pragma GCC optimize("-fcse-follow-jumps")
#pragma GCC optimize("-fsched-interblock")
#pragma GCC optimize("-fpartial-inlining")
#pragma GCC optimize("no-stack-protector")
#pragma GCC optimize("-freorder-functions")
```

```c
#pragma GCC optimize("-findirect-inlining")
#pragma GCC optimize("-fhoist-adjacent-loads")
#pragma GCC optimize("-frerun-cse-after-loop")
#pragma GCC optimize("inline-small-functions")
#pragma GCC optimize("-finline-small-functions")
#pragma GCC optimize("-ftree-switch-conversion")
#pragma GCC optimize("-foptimize-sibling-calls")
#pragma GCC optimize("-fexpensive-optimizations")
#pragma GCC optimize("inline-functions-called-once")
#pragma GCC optimize("-fdelete-null-pointer-checks")

int main(int argc, char *argv[]) {
    int count;        /* Local prime count */
    double elapsed_time; /* Parallel execution time */
    int first;        /* Index of first multiple */
    int global_count; /* Global prime count */
    int i, j;
    int id;          /* Process ID number */
    int index;        /* Index of current prime */
    int low_value;    /* Lowest value on this proc */
    char *marked;     /* Portion of 2,...,'n' */
    int n;           /* Sieving from 2, ..., 'n' */
    int p;           /* Number of processes */
    int prime;        /* Current prime */
    int size;         /* Elements in 'marked' */
    int m;           /* Size of search list */
    int offset;
    char *primes;     /* Preprocessed primes */
    int primes_size;  /* Elements in 'primes' */
    int chunk;        /* chunk size for *marked to adapt cache size */
    int low_value_chunk; /* Lowest value in a chunk */

    freopen("/dev/null","w",stderr);
    MPI_Init(&argc, &argv);

    /* Start the timer */

    MPI_Comm_rank(MPI_COMM_WORLD, &id);
    MPI_Comm_size(MPI_COMM_WORLD, &p);
    chunk = (7 << 20) / p;  // 256KBX14 L2 CacheX2=7MB
    MPI_Barrier(MPI_COMM_WORLD);
    elapsed_time = -MPI_Wtime();

    n = atoi(argv[1]);
    m = (n - 1) / 2;
```

```
/* Figure out this process's share of the array, as
   well as the integers represented by the first and
   last array elements */

low_value = INDEX_TO_ODD(BLOCK_LOW(id, p, m));
size = BLOCK_SIZE(id, p, m);

/* Allocate this process's share of the array. */

marked = (char *) calloc(size, sizeof(char));
if (marked == NULL) {
    printf("Cannot allocate enough memory\n");
    MPI_Finalize();
    exit(1);
}

primes_size = ODD_TO_INDEX((int)sqrt(n)) + 1;
primes = (char *) calloc(primes_size, sizeof(char));
if (primes == NULL) {
    printf("Cannot allocate enough memory\n");
    free(marked);
    MPI_Finalize();
    exit(1);
}

/* preprocess primes in 3..sqrt(n) */
index = 0;
prime = 3;
do {
    for (i = ODD_TO_INDEX(prime * prime); i < primes_size; i += prime)
        primes[i] = 1;
    while (primes[++index]);
    prime = INDEX_TO_ODD(index);
} while (prime * prime <= sqrt(n));

for(i = 0; i < size; i += chunk){
    index = 0;
    prime = 3;
    low_value_chunk = INDEX_TO_ODD(ODD_TO_INDEX(low_value) + i);
        do {
            if (prime * prime > low_value_chunk)
                first = ODD_TO_INDEX(prime * prime) -
ODD_TO_INDEX(low_value_chunk);
            else {
```

```
                    offset = low_value_chunk % prime;   // temp value
                    if (!offset) first = 0;
                    else {
                        first = prime - offset;
                        if (!((low_value_chunk + first) & 1))
                            first += prime;
                        first >>= 1;
                    }
                }
                for (j = first + i; j < first + i + chunk && j < size; j +=
prime) marked[j] = 1;
                while (primes[++index]);
                prime = INDEX_TO_ODD(index);
            } while (prime * prime <= n);
    }

    count = size;
    for (i = 0; i < size; i++)
        count -= marked[i]; // delete 'if'
    if(p > 1) MPI_Reduce(&count, &global_count, 1, MPI_INT, MPI_SUM, 0,
MPI_COMM_WORLD);
    else count = global_count;

    /* Stop the timer */

    elapsed_time += MPI_Wtime();

    /* Print the results */

    if (!id) {
        printf("There are %d primes less than or equal to %d\n",
              global_count + 1, n);
        printf("SIEVE (%d) %10.6f\n", p, elapsed_time);
    }
    MPI_Finalize();
    return 0;
}
```