电 子 科 技 大 学

实 验 报 告

学生姓名：王正仁	学 号：2019081308021

一、实验室名称：品学楼-A107

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

三、实验原理：

Eratosthenes 素数筛选原理：

Eratosthenes（公元前 276~194）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。

Eratosthenes 筛法的伪代码如下：

1. 创建一个自然数 2,3,4,…,n 的列表，其中所有的自然数都没有被标记。
2. 令 k=2，它是列表中第一个未被标记的数。
3. 重复下面的步骤直到$k^2 > n$ 为止：
   （a） 被k^2和 n 之间的是 k 倍数的数都标记出来。
   （b） 找出比 k 大的未被标记的数中最小的那个，令 k 等于这个数。
4. 列表中未被标记的数就是素数。

Eratosthenes 筛法的示例如下（N=120）：

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。

四、实验目的：

1. 掌握 MPI 环境搭建和 MPI 程序编译执行方法。
2. 使用 MPI 编程实现埃拉托斯特尼筛法。
3. 掌握并行程序性能分析以及优化的方法。

五、实验内容：

操作系统：Windows 10 x64

编程环境：WSL Ubuntu 18.04 bionic, MPICH 3.3a2

CPU 信息：

内核：8
逻辑处理器：16
L1 缓存：256 KB
L2 缓存：1.0 MB
L3 缓存：8.0 MB

1、根据实验指导书，完成 MPI 编译运行环境的配置。

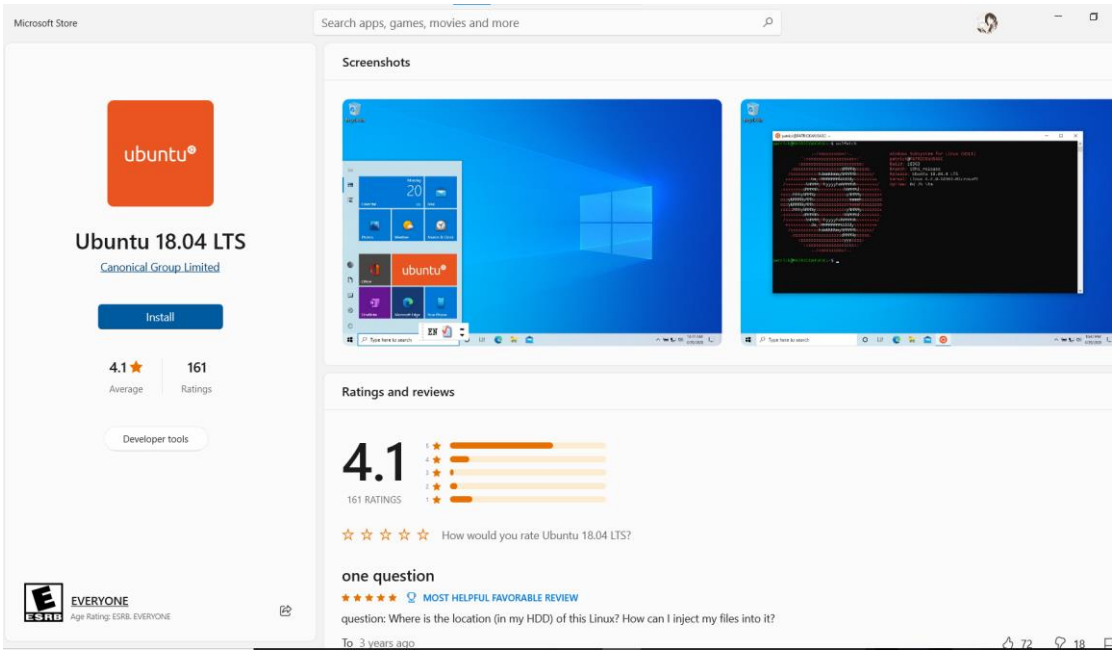
2、根据实验指导书给出的 MPI 基准代码，实测加速比并绘制曲线。

3、根据实验指导书给出的优化思路实现程序的并行优化。

4、在实验指导书的基础上，利用课内外知识，全面优化代码性能。

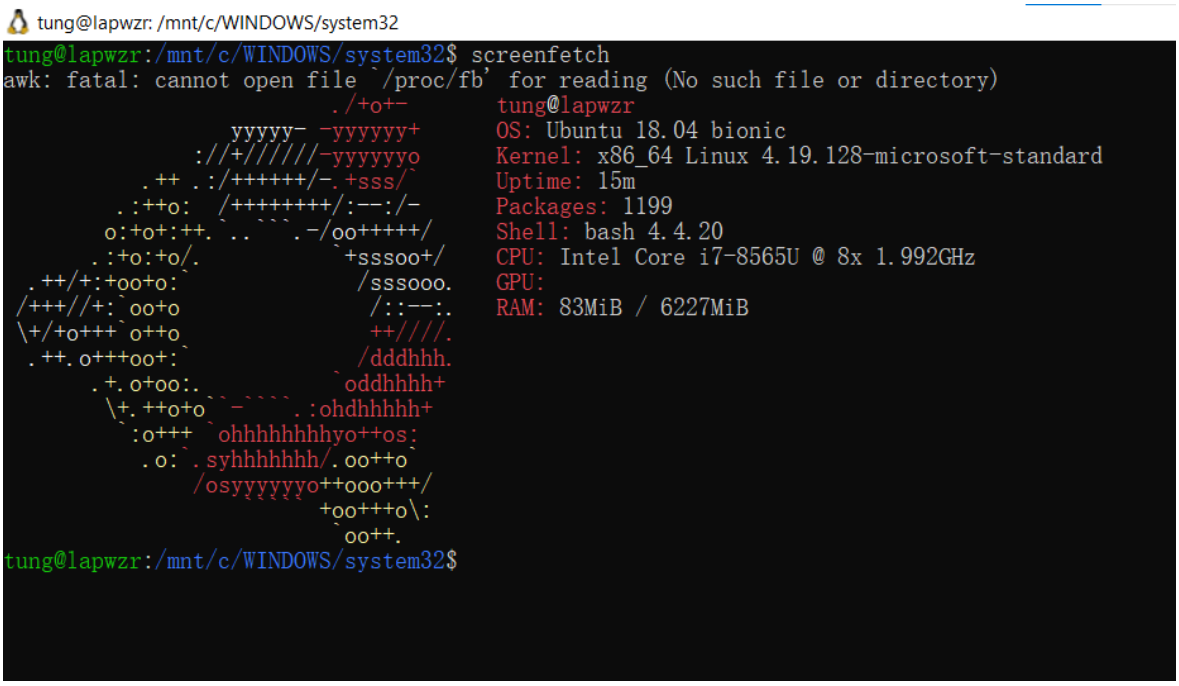
六、实验器材（环境配置）：

1. WSL 环境配置：



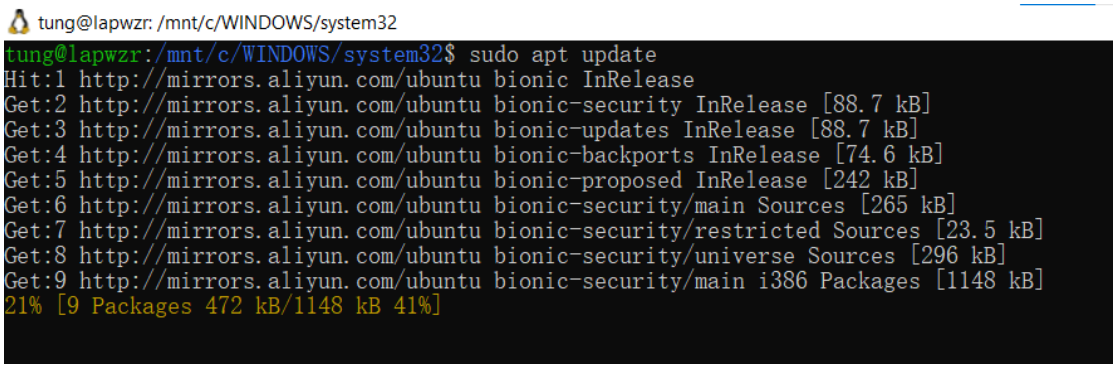
在微软应用商店中搜索 Ubuntu 18.04 LTS，点击安装。

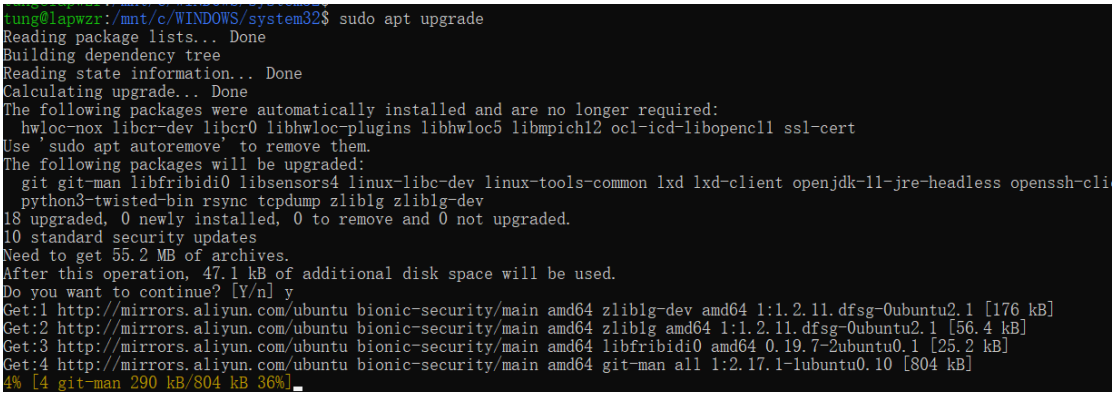
安装结束后重启系统，即可正常使用 WSL。



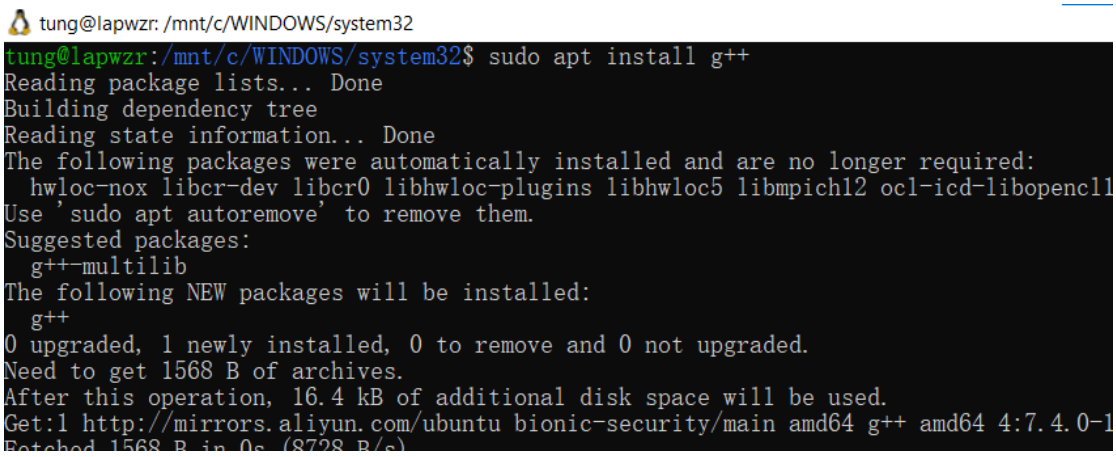
2. C++编译器环境配置：

首先分别执行“sudo apt update”和“sudo apt upgrade”更新软件源信息并且升级低版本系统软件。

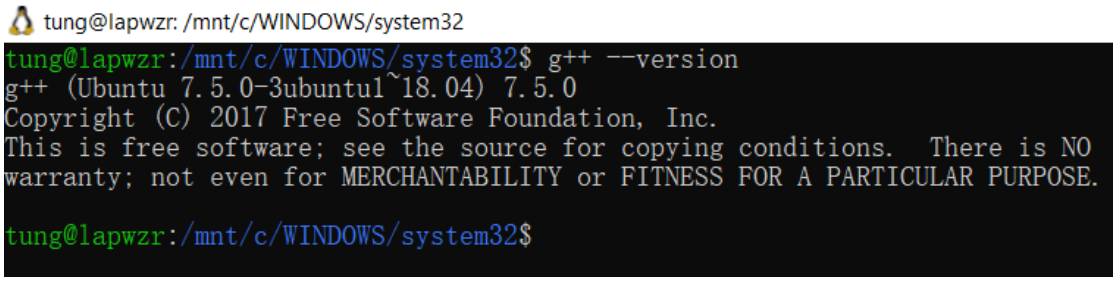




然后执行“sudo apt install g++”安装最新版本的 g++。

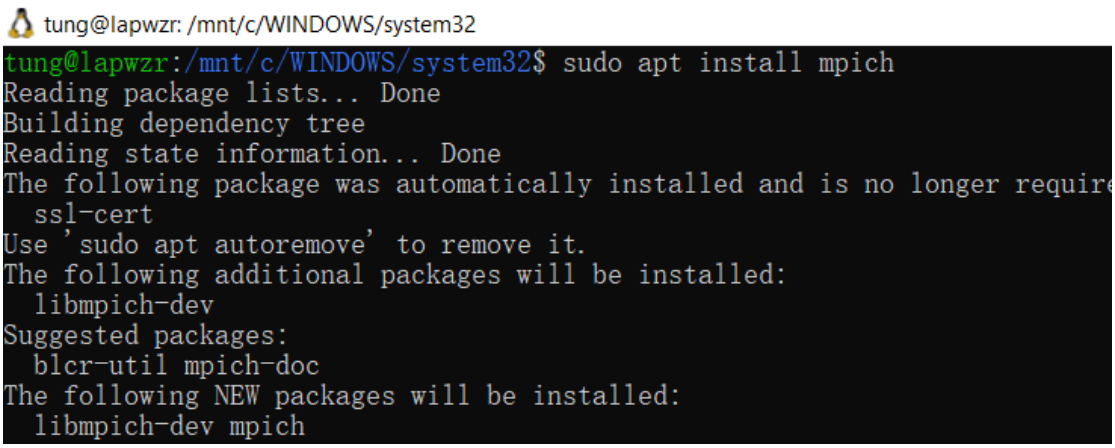


最后使用“g++ --version”验证安装成功。

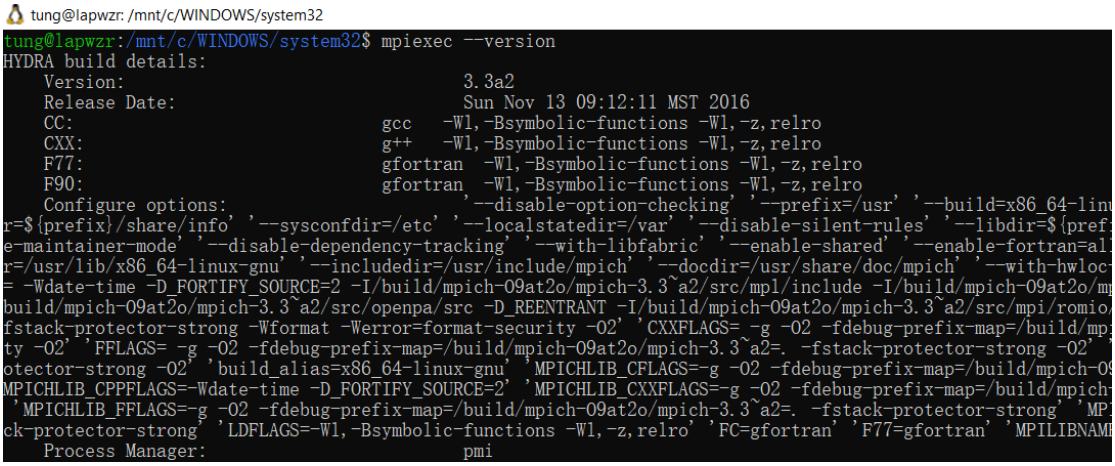


2.MPI 环境配置：

首先执行执行“sudo apt install mpich”安装最新版本的 MPICH。



然后使用“mpiexec --version”验证安装成功。



七、实验步骤及操作：

1. 首先，将实验源代码保存并命名为“base.cpp”，并根据优化任务创建四个文件“optimizer1.cpp”，“optimizer2.cpp”，“optimizer3.cpp”和“optimizer4.cpp”。同时为简化项目管理，编写 Makefile 如下：

Makefile

```
.PHONY: all

all: base optimizer1 optimizer2 optimizer3 optimizer4

base: base.cpp
    mpic++ -std=c++11 ./base.cpp -o base

optimizer1: optimizer1.cpp
    mpic++ -std=c++11 ./optimizer1.cpp -o optimizer1

optimizer2: optimizer2.cpp
    mpic++ -std=c++11 ./optimizer2.cpp -o optimizer2

optimizer3: optimizer3.cpp
    mpic++ -std=c++11 ./optimizer3.cpp -o optimizer3

optimizer4: optimizer4.cpp
    mpic++ -std=c++11 ./optimizer4.cpp -o optimizer4

clean:
    rm base optimizer1 optimizer2 optimizer3 optimizer4
```

2. 使用“make”命令编译，并使用“mpiexec -np n ./base limit”命令指定启动 n 个进程筛选不大于 limit 的素数。通过指定参数运行程序，得到结果如下图所示。

```
tung@lapwzr: /mnt/c/Users/tungsten/Desktop/2019081308021_王正仁
tung@lapwzr:/mnt/c/Users/tungsten/Desktop/2019081308021_王正仁$ make
mpic++ -std=c++11 ./base.cpp -o base
tung@lapwzr:/mnt/c/Users/tungsten/Desktop/2019081308021_王正仁$ mpiexec -np 1 ./base 10
There are -610386824 primes less than or equal to 10
SIEVE (1)   0.000001
tung@lapwzr:/mnt/c/Users/tungsten/Desktop/2019081308021_王正仁$ mpiexec -np 5 ./base 10
There are 5 primes less than or equal to 10
SIEVE (5)   0.000027
tung@lapwzr:/mnt/c/Users/tungsten/Desktop/2019081308021_王正仁$
```

显然可以发现，虽然编译成功完成，但程序运行中出现了不正确的结果，因此，需要对程序进行修改。

3. 排除基准代码中存在的 BUGs：

1）解决进程数为 1 时的异常现象：

查看并分析基准代码 base.cpp，注意到基准代码只在“p>1”的情况下进行了“Reduce”操作，并未考虑“p==0”的情况，如下图所示：

```
90      if (p > 1) MPI_Reduce (&count, &global_count, 1, MPI_INT, MPI_SUM,
91        0, MPI_COMM_WORLD);
```

因此补充“p==0”的情况，修改后的程序如下图所示：

```
90    if (p > 1) MPI_Reduce (&count, &global_count, 1, MPI_INT, MPI_SUM,
91       0, MPI_COMM_WORLD);
92    else global_count=count;
```

2）解决进程数过多时，统计结果偏大的异常现象：

查看并分析基准代码 base.cpp，发现该现象产生的原因是判断特殊情况的表达式边界处理条件有误--基准代码依赖于 sqrt(N)内所有的素数都在 0 号节点的区间内，因此要求 0 号节点处理的最大值不小于 sqrt(N)。原判断表达式如下：

```
54    if ((2 + proc0_size) < (int) sqrt((double) n)) {
55       if (!id) printf ("Too many processes\n");
56       MPI_Finalize();
57       exit (1);
58    }
```

应当修改“<”为“<=”：

```
54    if ((2 + proc0_size) <= (int) sqrt((double) n)) {
55       if (!id) printf ("Too many processes\n");
56       MPI_Finalize();
57       exit (1);
58    }
```

3）解决求解范围较大时，统计结果不正确的异常现象：

查看并分析基准代码 base.cpp，并结合对变量“low_value”和“high_value”的输出调试，发现该现象是 id*n 的范围超出整型从而发生了溢出导致的。原始代码如下：

```
45       low_value = 2 + id*(n-1)/p;
46       high_value = 1 + (id+1)*(n-1)/p;
47       size = high_value - low_value + 1;
```

仅需提升整型宽度为 long long 即可，修改后代码如下：

```
45       low_value = 2 + 1LL*id*(n-1)/p;
46       high_value = 1 + 1LL*(id+1)*(n-1)/p;
47       size = high_value - low_value + 1;
```

4. 评估修正后基准代码的性能：

对基准代码在不同进程数目和不同数据规模下的运行时间和加速比进行测试，并以此作为性能度量。选取的测试进程数为 1、2、4、8 和 16，数据规模为：100000、1000000、10000000、100000000 和 1000000000。

5. 去掉偶数（优化 1）：

基础的数学知识告诉我们：除 2 以外的所有偶数都不是素数。显然，我们可以提前筛去所有的偶数，从而将待筛元素减半，从而提高筛选效率。

基于以上思想，修改基准代码 base.cpp 为 optimizer1.cpp（代码于附录给出），并进行相同的测试。

6. 消除广播（优化 2）：

基准代码/优化 1 通过进程 0 广播当前筛去其倍数的素数，进程之间需要通过 MPI_Bcast 函数进行通信。通信会带来相互等待和阻塞等时间开销，因此消除通信有利于我们提升执行效率。

基础的数学知识告诉我们：任何一个合数 N 总有一个不大于 sqrt(N)的素因子。因此我们可以让每个进程都各自找出前 sqrt(N)个数中的素数，再通过这些本地的素数进行进一步筛选，从而消除广播通信。

基于以上思想，进一步修改 optimizer1.cpp 为 optimizer2.cpp（代码于附录给出），并进行相同的测试。

7. Cache 优化（优化 3）：

优化 2 算法的核心是一个嵌套循环，外层循环在$\sqrt{n}$内的素数间迭代，内层循环在待筛整数间迭代。内层循环其实是在一个巨大的数组上随机访问，容易造成严重的缓冲不命中现象，从而严重影响性能。但如果我们将数组分块，每次仅处理一块数组，先筛除其中所有的合数，然后再读入下一块数组，就可以显著提升缓存命中率。

在 Windows 系统中，可以通过任务管理器的“性能”选项查看机器的 Cache 缓存信息，在实验所用机器中，缓存信息如下：

L1 cache:	256 KB
L2 cache:	1.0 MB
L3 cache:	8.0 MB

考虑 L1、L2、L3 cache 的访问速度依然有一定差距，并以 L1 cache 访问速度最快；同时考虑现代 CPU 中通常 L1 cache 会被进一步划分为 dcache 和 icache；同时考虑到超线程技术下多个进程对同一 cache 的占用；同时考虑实验测试效果。我们将有效的 cache 大小取做 L1 cache 大小的一半，即 256*1024/2= 131072 字节。

基于以上思想，进一步修改 optimizer2.cpp 为 optimizer3.cpp（代码于附录给出），并

进行相同的测试。

7. 综合优化（优化 4）：

1）充分利用编译器工作者们的智慧结晶，开启 O3 优化。即在首行插入“#pragma GCC optimize(3,"Ofast","inline")”

2）重叠化各个分块，即不同分块共用相同的空间。

在优化 3 中，我们对待筛整数区间进行了分块，并且分块间顺序处理。注意到处理结束的分块中的素数可以被立即统计，在后续的处理中前面分块不需要再占据其空间。换而言之，我们可以让后面的分块重复利用当前分块的空间，从而进一步提高空间局部性。

3）去掉 3 和 5 的倍数

在优化 1 中，我们去掉了所有的偶数，即 2 的倍数，从而缩小了筛选空间，提高了整体性能。相似地，我们可以提前筛去 3 和 5 的倍数。

考虑 2、3、5 的最小公倍数 30。模 30 的同余系为{0，1，2，3，4，5，6，7，8，9，10，11，12，13，14，15，16，17，18，19，20，21，22，23，24，25，26，27，28，29}。

任一整数 i 可表示为 i=k*30+j（0<=j<30），因此整数 i 不是 2、3、5 的倍数等价于 j 不是 2、3、5 的倍数。故不是 2、3、5 倍数的整数一定可表示为 i=k*30+j（j={1，7，11，13，17，19，23，29}）。

根据带筛选整数模 30 的余数，我们可以将其分为 8 类，依次对应余数{1，7，11，13，17，19，23，29}。分类如下：

① {1，31，61，91，…}

② {7，37，67，97，…}

③ {11，41，71，101，…}

④ {13，43，73，103，…}

⑤ {17，47，77，107，…}

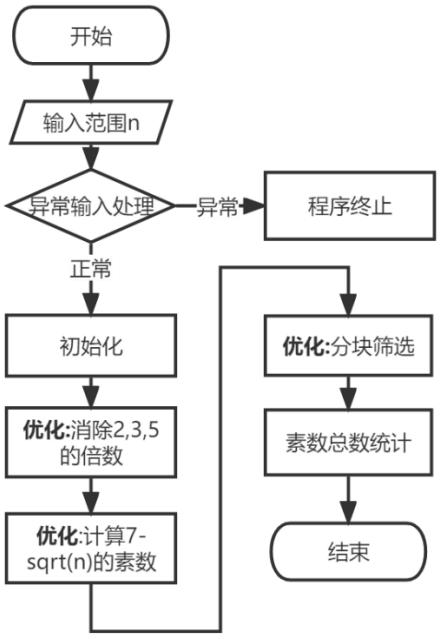
⑥ {19，49，79，109，…}

⑦ {23，53，83，113，…}

⑧ {29，59，89，119，…}

由此我们有 8 个待筛选集合，每个集合的大小为最初的 1/30（等间隔 30），总计相当于基准版本 8/30（26.7%）的筛选空间。

综合优化版本的流程图如下：



基于以上思想，进一步修改 optimizer3.cpp 为 optimizer4.cpp（代码于附录给出），并进行相同的测试。

八、实验数据及结果分析：

1. 基准代码

表 8-1：base 运行时间测量数据

规模 / 进程数	100000	1000000	10000000	100000000	1000000000
1	0.000821	0.008482	0.100809	1.749532	19.76493
2	0.000541	0.004728	0.06391	1.343613	15.77966
4	0.000498	0.003696	0.03855	1.223782	14.014738
8	0.000299	0.002221	0.031537	1.183006	13.93035
16	0.119963	0.189991	0.369967	2.179973	15.299984

表 8-2：base 加速比测量数据

规模 / 进程数	100000	1000000	10000000	100000000	1000000000
1	1	1	1	1	1
2	1.517560074	1.793993232	1.577358786	1.302110057	1.252557406
4	1.648594378	2.29491342	2.615019455	1.429610829	1.410296075
8	2.745819398	3.81900045	3.196531059	1.478886836	1.418839441
16	0.006843777	0.04464422	0.272481059	0.802547554	1.291826841

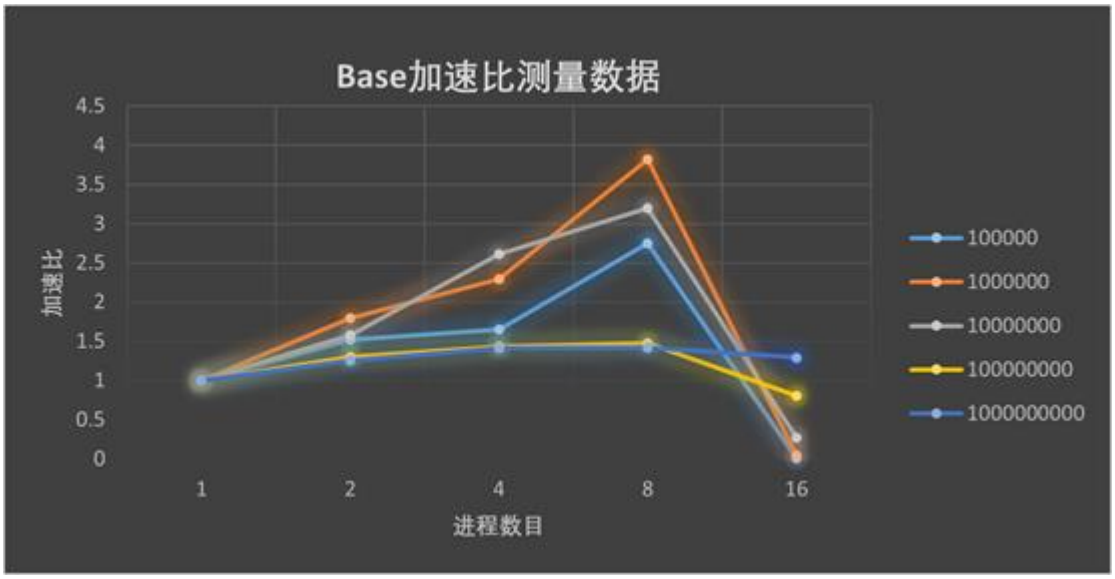


图 8-1：base 加速比测量数据

2. 去掉偶数（优化 1）：

表 8-3：optimizer1 运行时间测量数据

规模 / 进程数	100000	1000000	10000000	100000000	1000000000
1	0.00036	0.003938	0.041609	0.853435	9.887594
2	0.000263	0.002031	0.022682	0.645135	7.787484
4	0.000221	0.001587	0.01405	0.566372	6.986532
8	0.000225	0.001133	0.009545	0.556263	6.934211
16	0.119988	0.159992	0.319976	1.089974	7.799997

表 8-4：optimizer1 加速比测量数据

规模 / 进程数	100000	1000000	10000000	100000000	1000000000
1	1	1	1	1	1
2	1.368821293	1.938946332	1.834450225	1.322878157	1.269677601
4	1.628959276	2.481411468	2.961494662	1.506845324	1.415236343
8	1.6	3.475728155	4.359245678	1.534229312	1.425914787
16	0.0030003	0.024613731	0.130037878	0.782986567	1.267640744

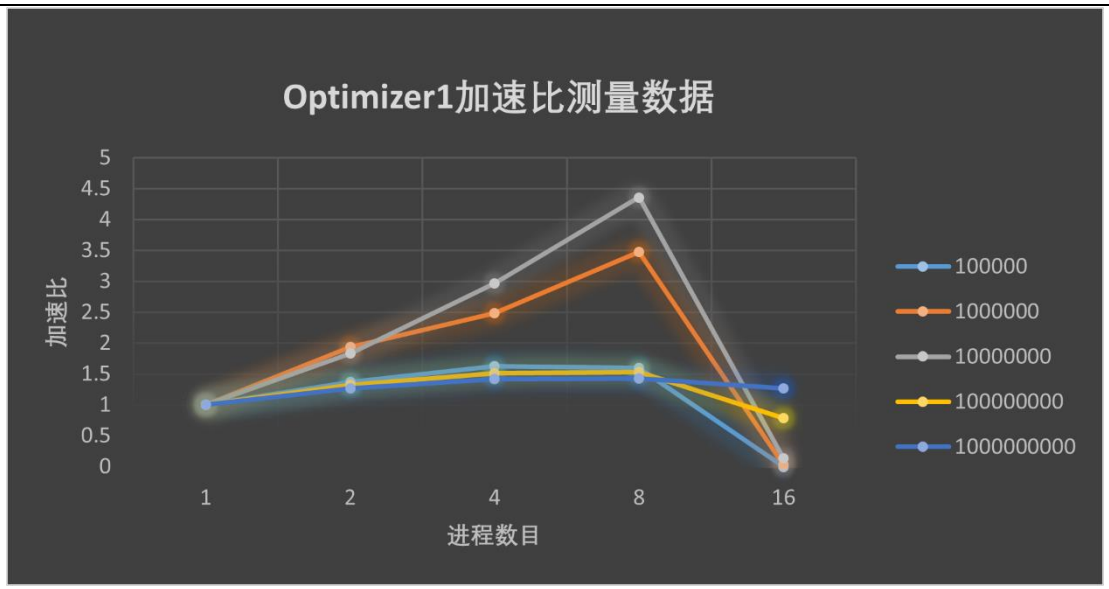


图 8-2：optimizer1 加速比测量数据

3. 消除广播（优化 2）：

表 8-5：optimizer2 运行时间测量数据

规模 / 进程数	100000	1000000	10000000	100000000	1000000000
1	0.000341	0.003721	0.039695	0.846831	9.86921
2	0.000193	0.001957	0.022296	0.652324	7.777509
4	0.000151	0.00146	0.015463	0.554592	6.974796
8	0.000103	0.000868	0.008291	0.554706	6.933726
16	0.060022	0.079996	0.079965	0.319994	6.922714

表 8-6：optimizer2 加速比测量数据

规模 / 进程数	100000	1000000	10000000	100000000	1000000000
1	1	1	1	1	1
2	1.766839378	1.901379663	1.780364191	1.298175447	1.26894228
4	2.258278146	2.548630137	2.567095648	1.526944132	1.414981886
8	3.310679612	4.286866359	4.787721626	1.526630323	1.423363138
16	0.00568125	0.046514826	0.496404677	2.646396495	1.42562729

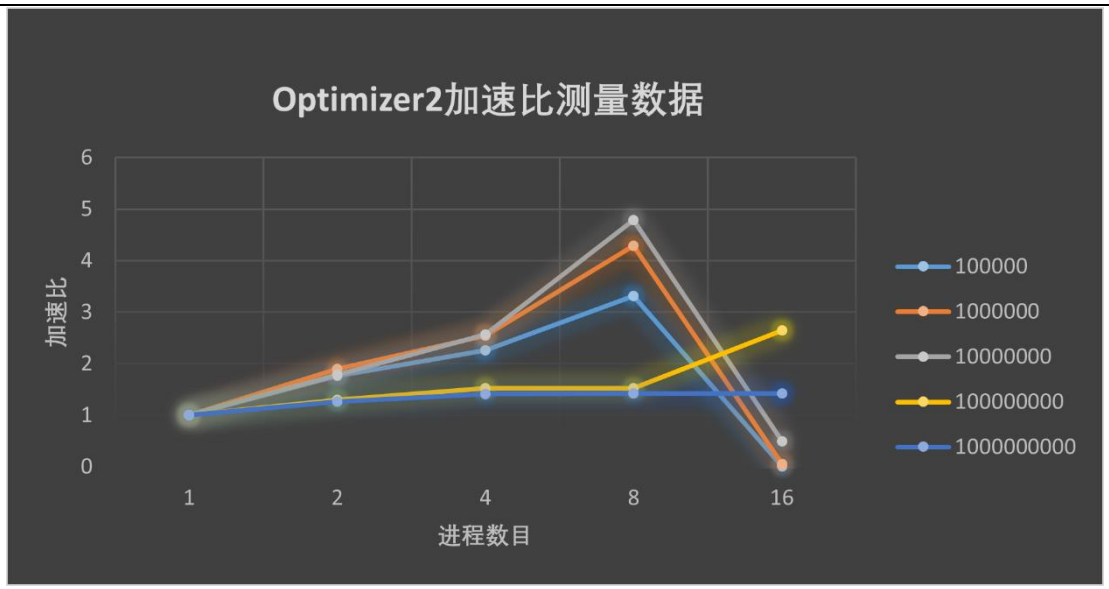


图 8-3：optimizer2 加速比测量数据

4. Cache 优化（优化 3）：

表 8-7：optimizer3 运行时间测量数据

规模 / 进程数	100000	1000000	10000000	100000000	1000000000
1	0.000375	0.004025	0.043421	0.429383	4.496545
2	0.000275	0.002121	0.023162	0.229668	2.370196
4	0.000159	0.001594	0.012853	0.128893	1.357996
8	0.00012	0.000937	0.008661	0.078063	0.841558
16	0.039982	0.059989	0.059992	0.144598	0.959954

表 8-8：optimizer3 加速比测量数据

规模 / 进程数	100000	1000000	10000000	100000000	1000000000
1	1	1	1	1	1
2	1.363636364	1.897689769	1.8746654	1.869581309	1.897119479
4	2.358490566	2.525094103	3.378277445	3.33131357	3.311162183
8	3.125	4.295624333	5.013393373	5.500467571	5.343119547
16	0.009379221	0.067095634	0.723779837	2.969494737	4.684125489

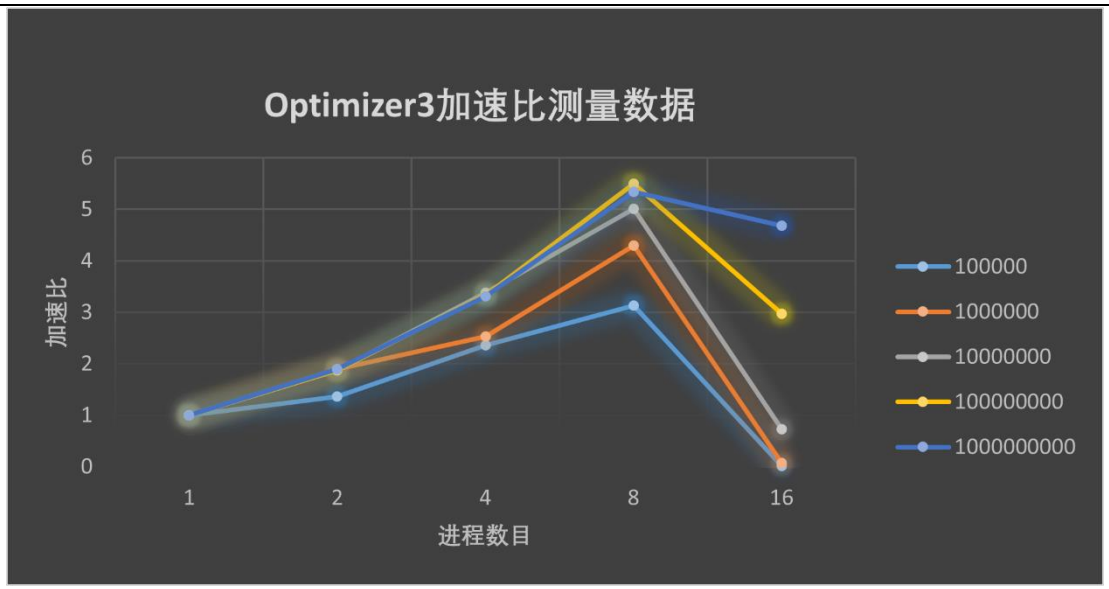


图 8-4：optimizer3 加速比测量数据

5. 综合优化（优化 4）：

表 8-9：optimizer4 运行时间测量数据

规模 / 进程数	100000	1000000	10000000	100000000	1000000000
1	0.000028	0.000214	0.00342	0.038223	0.427246
2	0.000029	0.00014	0.0018	0.02192	0.255225
4	0.000033	0.000134	0.001534	0.013579	0.164544
8	0.000041	0.000127	0.000986	0.01122	0.112971
16	0.049968	0.088348	0.068928	0.069994	0.189805

表 8-10：optimizer4 加速比测量数据

规模 / 进程数	100000	1000000	10000000	100000000	1000000000
1	1	1	1	1	1
2	0.965517241	1.528571429	1.9	1.74375	1.673997453
4	0.848484848	1.597014925	2.22946545	2.814861183	2.596545605
8	0.682926829	1.68503937	3.468559838	3.406684492	3.781908631
16	0.000560359	0.002422239	0.049616992	0.546089665	2.250973367

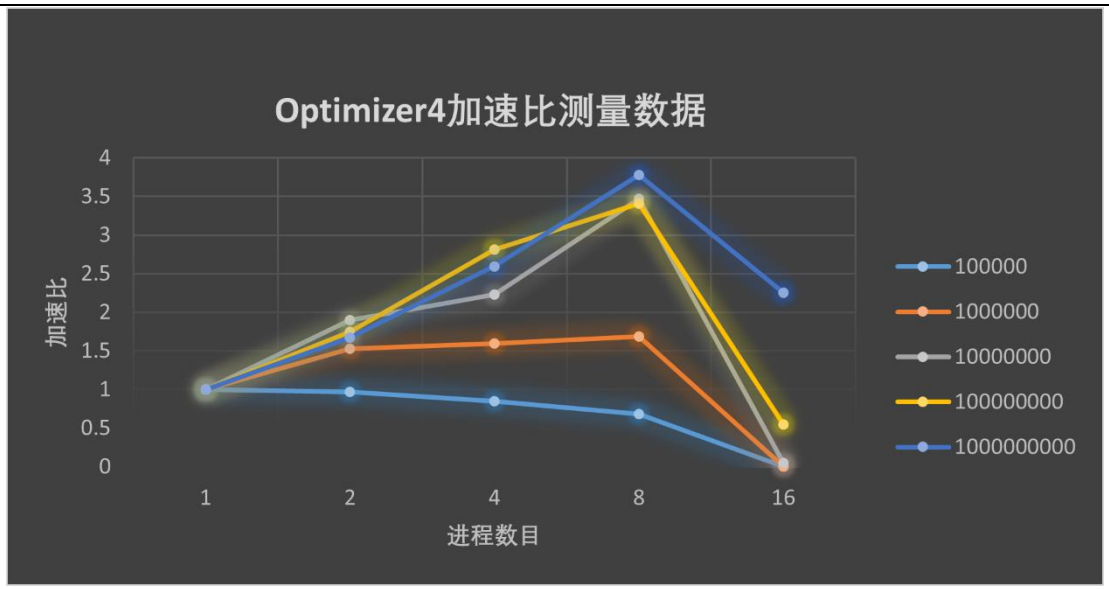


图 8-5：optimizer4 加速比测量数据

6. 版本对比：

表 8-11：各版本运行时间测量数据（N=1000000000）

版本 / 进程数	Base	Optimizer1	Optimizer2	Optimizer3	Optimizer4
1	19.76493	9.887594	9.86921	4.496545	0.427246
2	15.77966	7.787484	7.777509	2.370196	0.255225
4	14.014738	6.986532	6.974796	1.357996	0.164544
8	13.93035	6.934211	6.933726	0.841558	0.112971
16	15.299984	7.799997	6.922714	0.959954	0.189805

表 8-12：各版本加速比测量数据（N=1000000000）

版本 / 进程数	Base	Optimizer1	Optimizer2	Optimizer3	Optimizer4
1	1	1.998962538	2.002686132	4.395581496	46.2612406
2	1.252557406	2.538037959	2.541293106	8.338943277	77.44119894
4	1.410296075	2.829004433	2.8337646	14.55448322	120.1194209
8	1.418839441	2.850350242	2.850549618	23.48611742	174.9557851
16	1.291826841	2.533966359	2.855084003	20.58945533	104.1328205

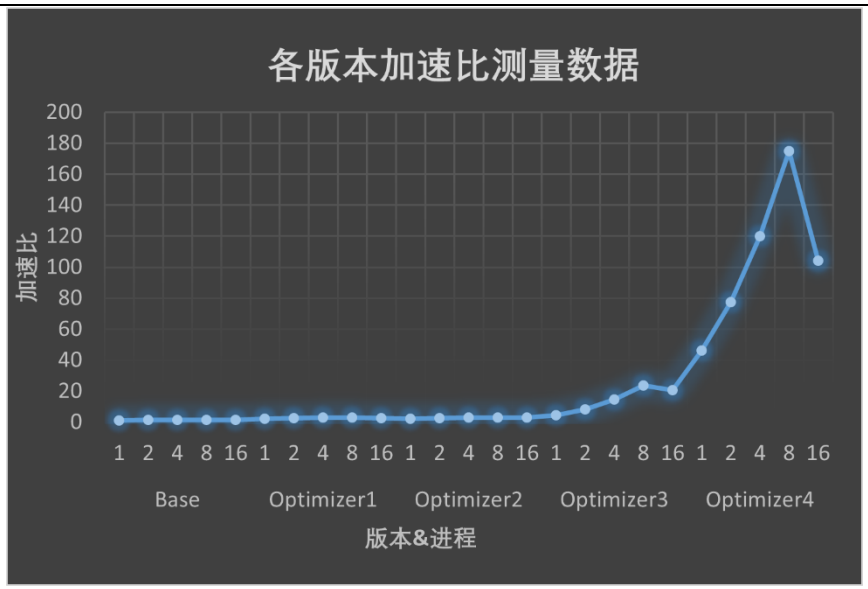


图 8-6：各版本加速比测量数据（N=1000000000）

初步分析以上图表，对于不同数据规模和进程数量，每一次优化后的程序总体性能都明显优于之前的程序，即：base<optimizer1<optimizer2<optimizer3<optimizer4。程序运行时间的减少符合预期结果，证明我们的优化切实有效，实验基本取得成功。

进一步分析以上图表：

1）不难看出，当数据规模较小或者进程数目较多时，程序运行时间有时随着进程数目增多反而增加。即，增加进程数目未必能提升总体效率。出现这种情况有多方面的原因：一是系统创建、管理及调度进程具有一定的开销，当任务量较小时，这些额外开销的代价可能会超过并行化的收益，反而不利于提高整体性能；二是大量进程的相互等待（通信）会造成进程的阻塞，严重影响有效执行时间；三是过多的进程会竞争处理机、内存和缓存资源，资源频繁被剥夺不利用性能的提高。但数据规模较大时，并行计算和算法优化带来的性能提升就较为明显。

2）比较基准版本 base 与去掉偶数版本 optimizer1，可以发现 optimizer1 的运行时间几乎是 base 的一半。原因在于：去除偶数减少了一半的数据空间，可近似看作减少了一半的计算量和内存占用，在数据规模较大可以充分发挥程序性能时，带来的提升就非常可观。

3）比较去掉偶数版本 optimizer1 与消除广播版本 optimizer2，可以发现二者之间无明显的性能差距，在数据规模不大或进程数不多时性能接近。原因在于：一方面各进程负载较为均衡，步调较为一致，因此通信时阻塞等待现象不明显，通信开销相比计算开销并不显著，可优化空间有限；另一方面，由于硬件配置的限制，无法测试更高的进程数目，在当前配置（最多 16 个进程）下通信开销尚未成为瓶颈。

4）比较消除广播版本 optimizer2 与 Cache 优化版本 optimizer3，可以发现数据规模较大时，改善程序局部性将带来明显的性能提升。原因在于：计算机系统内部的内存层次结构，不同层的容量和访问速度往往有数量级的差别，程序的主要部分是在一个巨大的表格上进行筛选，通过优化表项访问时间可以有效地提高性能。

5）比较 Cache 优化版本 optimizer3 和综合优化版本 optimizer4，可以发现综合优化版本相对于 Cache 优化版本进一步得到了明显提升。原因在于：一方面开启了 O3 优化，充分利用了科研工作者在编译优化上的智慧结晶；另一方面同时提前筛去了 3 和 5 的倍数，分析可知待筛选空间大约是基准版本空间的 1/4，进一步减少了计算量和内存占用；最后一方面我们将让不同的分块共用一块空间，使得内存占用量恰好为一个缓存块规模，进一步改善了缓存命中率。

总之，通过算法优化可以大幅度提升程序性能，同时针对不同任务，应具体问题具体分析，设计高效的算法。同时，程序的性能也受到其它方面的制约，应该综合考虑系统内各个因素之间的关系，这需要不断的实践、反馈和改进。

九、实验结论：

1.针对不同的计算任务，应结合理论并实事求是，具体问题具体分析,设计高效的算法。

2.并行算法设计类似于串行算法设计，设计第一步应该考虑减少总工作量。

3.并行算法设计不同于串行算法设计，设计时要考虑不同进程之间的并行度（通信等）。

4.无论是并行算法还是串行算法设计，设计时要考虑程序的局部性以提高缓存命中率。

5.实践中越多的并行进程不意味着更多的加速比，对于规模较小的任务，并行的额外开销和进程竞争现象可能会影响效率。应该综合考虑数据规模、软硬件、系统状态来决定应该如何应用并行程序。

十、总结及心得体会：

分布式并行计算充分利用了多核和集群等技术的资源优势，使得分布式并行系统能求解大规模问题，具有非常广泛的应用背景和价值。理解分布式并行系统的工作原理，并且编写基于该平台的高效代码和算法具有很高的研究价值。

本次实验中，我结合课内外所学知识（特别是 MPI 和 Cache 相关知识），立足于“埃拉托斯特尼素数筛选算法并行及性能优化”这一问题，学习和动手实战了分布式并行程序的设计、实现、调试、评估和 4 次优化，拓宽了知识面的同时，锻炼了编程能力、动手能力、分析能力、总结能力，获益匪浅。

王正仁

十一、对本实验过程及方法、手段的改进建议：

1. 服务器限制学生进程的执行时间，避免失败的编程导致对服务器资源的消耗和独占。

2. 在教学经费允许的条件下，考虑搭建集群测试环境，从而能真正分布式的环境中进行实验，而不只是多核编程环境；同时在分布式环境下能结合 MPI 和 OpenMP 技术。

报告评分：

指导教师签字：

十二、附录：

一、base.cpp（Bugs Fixed）

```
#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    low_value;    /* Lowest value on this proc */
   int    high_value;   /* Highest value on this proc */
   int    i;
   int    id;           /* Process ID number */
   int    index;        /* Index of current prime */
   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' */

   /* Start the timer */
   MPI_Init (&argc, &argv);
   MPI_Comm_rank (MPI_COMM_WORLD, &id);
   MPI_Comm_size (MPI_COMM_WORLD, &p);
   MPI_Barrier(MPI_COMM_WORLD);
   elapsed_time = -MPI_Wtime();

   /* Handle Illegal input*/
   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
```

```
41.       well as the integers represented by the first and
42.       last array elements */
43.    low_value = 2 + 1LL*id*(n-1)/p;
44.    high_value = 1 + 1LL*(id+1)*(n-1)/p;
45.    size = high_value - low_value + 1;
46.
47.    /* Bail out if all the primes used for sieving are
48.       not all held by process 0 */
49.    proc0_size = (n-1)/p;
50.    if ((2 + proc0_size) <= (int) sqrt((double) n)) {
51.       if (!id) printf ("Too many processes\n");
52.       MPI_Finalize();
53.       exit (1);
54.    }
55.
56.    /* Allocate this process's share of the array. */
57.    marked = (char *) malloc (size);
58.
59.    /* Handle insufficent memory*/
60.    if (marked == NULL) {
61.       printf ("Cannot allocate enough memory\n");
62.       MPI_Finalize();
63.       exit (1);
64.    }
65.    for (i = 0; i < size; i++) marked[i] = 0;
66.
67.    if (!id) index = 0;
68.    prime = 2;
69.    do {
70.       /* Get the index of first multiple*/
71.       if (prime * prime > low_value)
72.          first = prime * prime - low_value;
73.       else {
74.          if (!(low_value % prime)) first = 0;
75.          else first = prime - (low_value % prime);
76.       }
77.       /* Sieve the multiples*/
78.       for (i = first; i < size; i += prime) marked[i] = 1;
79.       /* Get the next prime*/
80.       if (!id) {
81.          while (marked[++index]);
82.          prime = index + 2;
83.       }
84.       /* Broadcast the next prime*/
```

```
85.      if (p > 1) MPI_Bcast (&prime,  1, MPI_INT, 0, MPI_COMM_WORL
   D);
86.    } while (prime * prime <= n);
87.
88.    /* Count the number of primes*/
89.    count = 0;
90.    for (i = 0; i < size; i++) if (!marked[i]) count++;
91.
92.    /* Reduce local counts*/
93.    if (p > 1) MPI_Reduce (&count, &global_count, 1, MPI_INT, MPI_
   SUM,0, MPI_COMM_WORLD);
94.    else global_count=count;
95.
96.    /* Stop the timer */
97.    elapsed_time += MPI_Wtime();
98.
99.    /* Print the results */
100.    if (!id) {
101.       printf ("There are %d primes less than or equal to %d\n",g
   lobal_count, n);
102.       printf ("SIEVE (%d) %10.6f\n", p, elapsed_time);
103.    }
104.    MPI_Finalize ();
105.    return 0;
106. }
```

二、optimizer1.cpp

```
1. #include "mpi.h"
2. #include <cmath>
3. #include <cstdio>
4. #include <cstdlib>
5. #define BLOCK_LOW(id,p,n)  ((1LL*id)*(n)/(p))
6. #define BLOCK_HIGH(id,p,n) (BLOCK_LOW((id)+1,p,n)-1)
7.
8. int main(int argc, char *argv[])
9. {
10.    double elapsed_time; /* Parallel execution time */
11.    int    count;        /* Local prime count */
12.    int    global_count; /* Global prime count */
13.    int    high_value;   /* Highest value on this proc */
14.    int    low_value;    /* Lowest value on this proc */
15.    int    first;        /* Index of first multiple */
16.    int    id;           /* Process ID number */
17.    int    index;        /* Index of current prime */
```

```
18.   bool  *marked;       /* Portion of 2,...,'n' */
19.   int    n;            /* Sieving from 2, ..., 'n' */
20.   int    p;            /* Number of processes */
21.   int    prime;        /* Current prime */
22.   int    size;         /* Elements in 'marked' */
23.   int    no;           /* Number of odds in 3, ..., 'n' */
24.
25.   /* Start the timer */
26.   MPI_Init (&argc, &argv);
27.   MPI_Comm_rank (MPI_COMM_WORLD, &id);
28.   MPI_Comm_size (MPI_COMM_WORLD, &p);
29.   MPI_Barrier(MPI_COMM_WORLD);
30.   elapsed_time = -MPI_Wtime();
31.
32.   n = atoi(argv[1]);
33.   no = (n-1)/2; /* Number of odds in 3, ..., 'n' */
34.
35. /* Figure out this process's share of the array, as
36.      well as the integers represented by the first and
37.      last array elements */
38.   low_value = 2 * BLOCK_LOW(id, p, no) + 3;
39.   high_value = 2 * BLOCK_HIGH(id, p, no) + 3;
40.   size = (high_value-low_value)/2+1;
41.
42. /* Allocate this process' share of the array */
43. marked = (bool *) malloc(size);
44. for (int i=0; i<size; i++) marked[i] = 0;
45.
46. index = 0;
47. prime = 3;
48. do {
49.      /* Get the index of first multiple*/
50.  first = (prime*prime-low_value)/2;
51.  if(first<0) first=(first%prime+prime)%prime;
52.      /* Sieve the multiples*/
53.  for (int i=first; i <size; i+=prime) marked[i] = 1;
54.      /* Get the next prime*/
55.      if(!id){
56.         while (marked[++index]);
57.     prime = (index<<1)+3;
58.      }
59.      /* Broadcast the next prime*/
60.      MPI_Bcast (&prime,  1, MPI_INT, 0, MPI_COMM_WORLD);
61. } while (prime*prime<=n);
```

```
62. /* Count the number of primes*/
63. count = 0;
64. for (int i=0; i<size; i++)if (!marked[i]) count++;
65.   /* Reduce local counts*/
66. MPI_Reduce(&count, &global_count, 1, MPI_INT, MPI_SUM, 0, MPI_CO
    MM_WORLD);
67. /* Stop the timer */
68. elapsed_time += MPI_Wtime();
69.
70. /* Print the results */
71.   if (!id) {
72.      printf ("There are %d primes less than or equal to %d\n",gl
    obal_count+1, n);
73.      printf ("SIEVE (%d) %10.6f\n", p, elapsed_time);
74.   }
75.   free(marked);
76.   MPI_Finalize ();
77.   return 0;
78.}
```

三、optimizer2.cpp

```
1. #include "mpi.h"
2. #include <cmath>
3. #include <cstdio>
4. #include <cstdlib>
5. #define BLOCK_LOW(id,p,n)  ((1LL*id)*(n)/(p))
6. #define BLOCK_HIGH(id,p,n) (BLOCK_LOW((id)+1,p,n)-1)
7.
8. int main(int argc, char *argv[])
9. {
10.   double elapsed_time; /* Parallel execution time */
11.   int    count;        /* Local prime count */
12.   int    global_count; /* Global prime count */
13.   int    high_value;   /* Highest value on this proc */
14.   int    low_value;    /* Lowest value on this proc */
15.   int    first;        /* Index of first multiple */
16.   int    id;           /* Process ID number */
17.   int    index;        /* Index of current prime */
18.   bool  *marked;       /* Portion of 2,...,'n' */
19.   int    n;            /* Sieving from 2, ..., 'n' */
20.   int    p;            /* Number of processes */
21.   int    prime;        /* Current prime */
22.   int    size;         /* Elements in 'marked' */
23.   int    no;           /* Number of odds in 3, ..., 'n' */
```

```
24.
25.    /* Start the timer */
26.    MPI_Init (&argc, &argv);
27.    MPI_Comm_rank (MPI_COMM_WORLD, &id);
28.    MPI_Comm_size (MPI_COMM_WORLD, &p);
29.    MPI_Barrier(MPI_COMM_WORLD);
30.    elapsed_time = -MPI_Wtime();
31.
32.    n = atoi(argv[1]);
33.    no = (n-1)/2; /* Number of odds in 3, ..., 'n' */
34.
35. /* Figure out this process's share of the array, as
36.       well as the integers represented by the first and
37.       last array elements */
38.    low_value = 2 * BLOCK_LOW(id, p, no) + 3;
39.    high_value = 2 * BLOCK_HIGH(id, p, no) + 3;
40.    size = (high_value-low_value)/2+1;
41.
42. int sqr=sqrt(n),sqrr=sqrt(sqr),primes_size = (sqr-
    1)/2; /* Number of odds in 3, ..., 'sqr' */
43. /* Allocate this process' share of the array */
44. marked = (bool *) malloc(size);
45. for (int i=0; i<size; i++) marked[i] = 0;
46. bool* primes = (bool *) malloc(primes_size);
47. for (int i=0; i<primes_size; i++) primes[i] = 0;
48.
49. index = 0;
50. prime = 3;
51. do {
52.       /* Sieve the multiples*/
53.  for (int i = (prime*prime-
    3)/2; i < primes_size; i += prime) primes[i] = 1;
54.  /* Get the next prime*/
55.       while (primes[++index]);
56.  prime = (index<<1)+3;
57. } while (prime <= sqrr);
58.
59. index = 0;
60. prime = 3;
61. do {
62.       /* Get the index of first multiple*/
63.  first = (prime*prime-low_value)/2;
64.  if(first<0) first=(first%prime+prime)%prime;
65.       /* Sieve the multiples*/
```

```
66.  for (int i=first; i <size; i+=prime) marked[i] = 1;
67.      /* Get the next prime*/
68.  while (primes[++index]);
69.  prime = (index<<1)+3;
70. } while (index<primes_size);
71.   /* Count the number of primes*/
72. count = 0;
73. for (int i=0; i<size; i++)if (!marked[i]) count++;
74.   /* Reduce local counts*/
75. MPI_Reduce(&count, &global_count, 1, MPI_INT, MPI_SUM, 0, MPI_CO
    MM_WORLD);
76. /* Stop the timer */
77. elapsed_time += MPI_Wtime();
78.
79. /* Print the results */
80.   if (!id) {
81.      printf ("There are %d primes less than or equal to %d\n",gl
    obal_count+1, n);
82.      printf ("SIEVE (%d) %10.6f\n", p, elapsed_time);
83.   }
84.   free(marked);
85.   free(primes);
86.   MPI_Finalize ();
87.   return 0;
88.}
```

四、optimizer3.cpp

```
1. #include "mpi.h"
2. #include <cmath>
3. #include <cstdio>
4. #include <cstdlib>
5. #define BLOCK_LOW(id,p,n)  ((1LL*id)*(n)/(p))
6. #define BLOCK_HIGH(id,p,n) (BLOCK_LOW((id)+1,p,n)-1)
7.
8. int main(int argc, char *argv[])
9. {
10.   double elapsed_time; /* Parallel execution time */
11.   int    count;        /* Local prime count */
12.   int    global_count; /* Global prime count */
13.   int    high_value;   /* Highest value on this proc */
14.   int    low_value;    /* Lowest value on this proc */
15.   int    first;        /* Index of first multiple */
16.   int    id;           /* Process ID number */
17.   int    index;        /* Index of current prime */
```

```
18.    bool  *marked;       /* Portion of 2,...,'n' */
19.    int    n;            /* Sieving from 2, ..., 'n' */
20.    int    p;            /* Number of processes */
21.    int    prime;        /* Current prime */
22.    int    size;         /* Elements in 'marked' */
23.    int    no;           /* Number of odds in 3, ..., 'n' */
24.    int    chunk=128*1024;/* Batch size (half of L1-cache)*/
25.
26.    /* Start the timer */
27.    MPI_Init (&argc, &argv);
28.    MPI_Comm_rank (MPI_COMM_WORLD, &id);
29.    MPI_Comm_size (MPI_COMM_WORLD, &p);
30.    MPI_Barrier(MPI_COMM_WORLD);
31.    elapsed_time = -MPI_Wtime();
32.
33.    n = atoi(argv[1]); if(argc==3)chunk=atoi(argv[2]);
34.    no = (n-1)/2; /* Number of odds in 3, ..., 'n' */
35.
36. /* Figure out this process's share of the array, as
37.       well as the integers represented by the first and
38.       last array elements */
39.    low_value = 2 * BLOCK_LOW(id, p, no) + 3;
40.    high_value = 2 * BLOCK_HIGH(id, p, no) + 3;
41.    size = (high_value-low_value)/2+1;
42.
43.  int sqr=sqrt(n),sqrr=sqrt(sqr),primes_size=(sqr-
    1)/2; /* Number of odds in 3, ..., 'sqr' */
44.  /* Allocate this process' share of the array */
45.  marked = (bool *) malloc(size);
46.  for (int i=0; i<size; i++) marked[i] = 0;
47.  bool* primes = (bool *) malloc(primes_size);
48.  for (int i=0; i<primes_size; i++) primes[i] = 0;
49.
50.  index = 0;
51.  prime = 3;
52.  do {
53.       /* Sieve the multiples*/
54.   for (int i = (prime*prime-
    3)/2; i < primes_size; i += prime) primes[i] = 1;
55.   /* Get the next prime*/
56.       while (primes[++index]);
57.   prime = (index<<1)+3;
58.  } while (prime <= sqrr);
59.
```

```
   /* Sieve by batch for better cache-hit rate*/
 for (int sec = 0; sec < size; sec += chunk) {
  index = 0;
  prime = 3;
  int lv = 2*((low_value-3)/2+sec)+3;
  do {
         /* Get the index of first multiple*/
   first = (prime*prime-lv)/2;
   if(first<0) first=(first%prime+prime)%prime;
         /* Sieve the multiples*/
   for (int i = first+sec; i < first+sec+chunk && i < size; i += prime)
    marked[i] = 1;
   /* Get the next prime*/
         while (primes[++index]);
   prime = 2*index + 3;
  } while (prime <= sqr);
 }
 /* Count the number of primes*/
 count = 0;
 for (int i=0; i<size; i++)if (!marked[i]) count++;
   /* Reduce local counts*/
 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);
   }
   free(marked);
   free(primes);
   MPI_Finalize ();
   return 0;
}
```

五、optimizer4.cpp

```
#pragma GCC optimize(3,"Ofast","inline") /* O3 optimization */
#include "mpi.h"
#include <cmath>
#include <cstdio>
```

```
#include <cstdlib>
#include <cstring>
#include <vector>
#define BLOCK_LOW(id,p,n)  ((1LL*id)*(n)/(p))
#define BLOCK_HIGH(id,p,n) (BLOCK_LOW((id)+1,p,n)-1)
#define MIN(a,b) ((a)<(b)?(a):(b))
/* Auxiliary array to handle low 'n' */
int aux[8]={0,0,1,2,2,3,3,4};
/* Auxiliary arrays to compute first multiple's index */
int begArr[8]={31,7,11,13,17,19,23,29};
int posArr[30]={0,0,0,0,0,0,0,1,0,0,0,2,0,3,0,0,0,4,0,5,0,0,0,6,0
,0,0,0,0,7};
int offsetArr[8][8]={
{0,6,0,24,6,0,24,0},
{6,24,6,6,24,24,6,24},
{10,16,20,4,26,10,14,20},
{12,12,12,18,12,18,18,18},
{16,4,26,16,14,4,26,14},
{18,0,18,0,0,12,0,12},
{22,22,2,28,2,28,8,8},
{28,10,8,10,20,22,20,2}
};
int main(int argc, char *argv[])
{
   double elapsed_time; /* Parallel execution time */
   int    count;        /* Local prime count */
   int    global_count; /* Global prime count */
   int    high_block;   /* Highest block index on this proc */
   int    low_block;    /* Lowest block index on this proc */
   int    first;        /* Index of first multiple */
   int    id;           /* Process ID number */
   int    index;        /* Index of current prime */
   bool  *marked;       /* Portion of 2,...,'n' */
   int    n;            /* Sieving from 2, ..., 'n' */
   int    p;            /* Number of processes */
   int    prime;        /* Current prime */
   int    num_block;    /* Elements in 'marked' */
   int    nblk;         /* Number of blocks in 7, ..., 'n' */
   int    chunk=128*1024;/* Batch size (half of L1-cache)*/

   /* Start the timer */
   MPI_Init (&argc, &argv);
   MPI_Comm_rank (MPI_COMM_WORLD, &id);
   MPI_Comm_size (MPI_COMM_WORLD, &p);
```

```
48.    MPI_Barrier(MPI_COMM_WORLD);
49.    elapsed_time = -MPI_Wtime();
50.
51.    n = atoi(argv[1]); if(argc==3)chunk=atoi(argv[2]);
52.
53.    /* Handle low 'n' */
54.    if(n<=7){
55.       if(!id){
56.          elapsed_time += MPI_Wtime();
57.          printf ("There are %d primes less than or equal to %d\n"
   ,aux[n], n);
58.          printf ("SIEVE (%d) %10.6f\n", p, elapsed_time);
59.       }
60.       MPI_Finalize ();
61.       return 0;
62.    }
63.
64.  int sqr=sqrt(n),sqrr=sqrt(sqr),primes_size=(sqr-
   1)/2; /* Number of odds in 3, ..., 'sqr' */
65.  /* Allocate this process' share of the array */
66.    bool* primes = (bool *) malloc(primes_size);
67.  memset(primes,0,primes_size);
68.
69.  index = 0; prime = 3;
70.  do {
71.       /* Sieve the multiples*/
72.   for (int i = (prime*prime-
   3)/2; i < primes_size; i += prime) primes[i] = 1;
73.   /* Get the next prime*/
74.       while (primes[++index]);
75.   prime = (index<<1)+3;
76.  } while (prime <= sqrr);
77.    /* Compact primes*/
78.    std::vector<int> primeArr; primeArr.reserve(primes_size);
79.    index = 2; prime = 7;
80.    do{
81.       primeArr.push_back(prime);
82.       while (primes[++index]);
83.       prime = (index<<1)+3;
84.    }while(prime <= sqr);
85.
86.    nblk = (n+23)/30; /* Number of blocks in 7, ..., 'n' */
87.    low_block = BLOCK_LOW(id, p, nblk);
88.    high_block = BLOCK_HIGH(id, p, nblk);
```

```
89.    num_block = high_block-low_block+1;
90.  marked = (bool *) malloc(MIN(num_block,chunk));
91.
92.    count=0;
93.    /* Sieve by batch for better cache-hit rate*/
94.    for (int sec = 0; sec < num_block; sec += chunk) {
95.       /* Sieve each Congruence Class*/
96.       for(int k=0;k<8;++k){
97.          int block=MIN(chunk,num_block-sec);
98.          if(n<begArr[k]+(low_block+sec+block-1)*30)block--;
99.          memset(marked,0,block);
100.          for(int i=0;i<primeArr.size();++i){
101.             int prime=primeArr[i];
102.             /* Get the index of first multiple*/
103.             int pos=posArr[prime%30], offset=offsetArr[k][pos];
104.             first = (prime*(prime+offset)-7)/30-low_block-sec;
105.             if(first<0) first=(first%prime+prime)%prime;
106.             /* Sieve the multiples*/
107.             for (int i = first; i < block; i += prime)marked[i]
    = 1;
108.          }
109.          /* Count the number of primes*/
110.          for (int i=0; i<block; i++)if (!marked[i]) count++;
111.       }
112.    }
113. /* Reduce local counts*/
114. MPI_Reduce(&count, &global_count, 1, MPI_INT, MPI_SUM, 0, MPI_C
    OMM_WORLD);
115. /* Stop the timer */
116. elapsed_time += MPI_Wtime();
117.
118. /* Print the results */
119.    if (!id) {
120.       printf ("There are %d primes less than or equal to %d\n",g
    lobal_count+3, n);
121.       printf ("SIEVE (%d) %10.6f\n", p, elapsed_time);
122.    }
123.    free(marked);
124.    free(primes);
125.    MPI_Finalize ();
126.    return 0;
127. }
```