电 子 科 技 大 学

实 验 报 告

学生姓名：关文聪	学 号：2016060601008

一、实验室名称：无

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：

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 MPI 实现：

(1)数据块分配方法

先分配每个进程所需要计算的元素，需要解决两个问题：每个进程的最小元素和最大元素、给定元素属于哪个进程。

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

初始版并行代码：

定义一个数组 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，它一定是素数，在使用广播的形式通知各进程筛选出这个素数的倍数。

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. 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：Microsoft MPI（MSMPI）、MinGW-W64 gcc version 8.1.0

1、根据附录 1 指示，完成 MPI 编译运行环境的配置。

2、根据附录 3 给出的基础 MPI 版本埃拉托斯特尼筛法 sieve1，实测加速比并绘制曲线。

3、根据附录 4 给出的优化思路实现程序的并行优化。

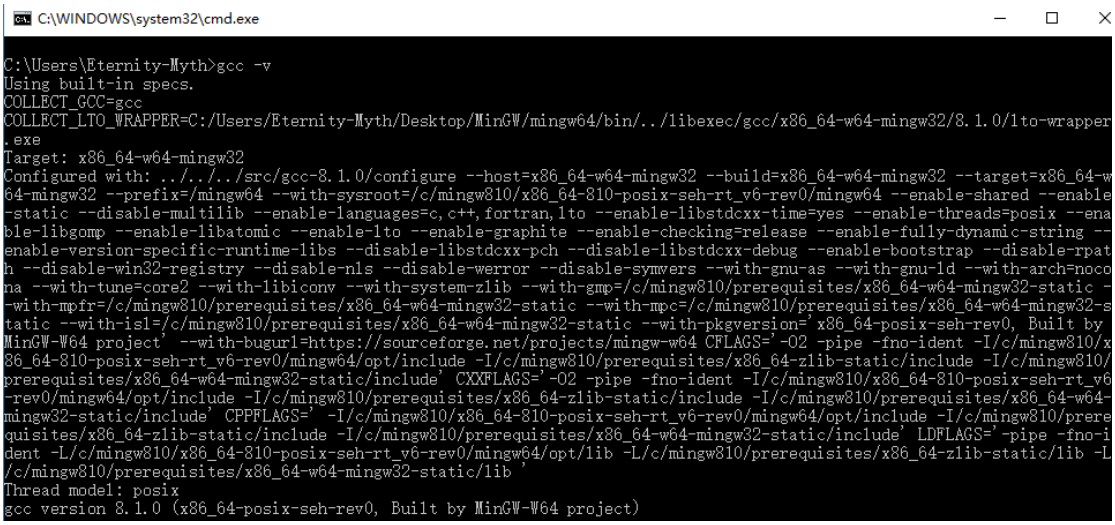
六、实验器材（环境配置）：

1. C/C++编译器环境配置：

首先，到 MinGW 官方网站：http://www.mingw-w64.org 下载并安装 MinGW-W64（详细过程略）。安装完成后，进行环境变量配置：将 mingw64 目录下的 bin 文件夹添加到系统环境变量的 Path 中；新建环境变量“INCLUDE”，将 mingw64 目录下的 include 文件夹添加到该环境变量“INCLUDE”中；新建环境变量“LIB”，将 mingw64 目录下的 lib 文件夹添加到该环境变量“LIB”中，保存退出。

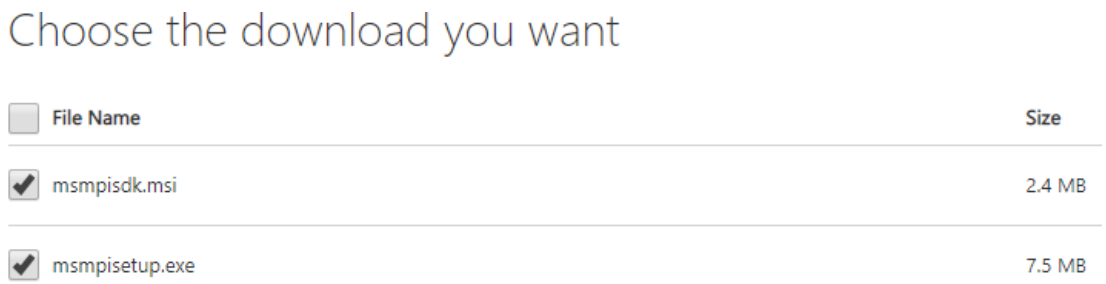
运行 cmd.exe 打开命令行窗口，输入“gcc -v”，能成功输出 gcc 版本信息，即为配置成功，

如图所示：



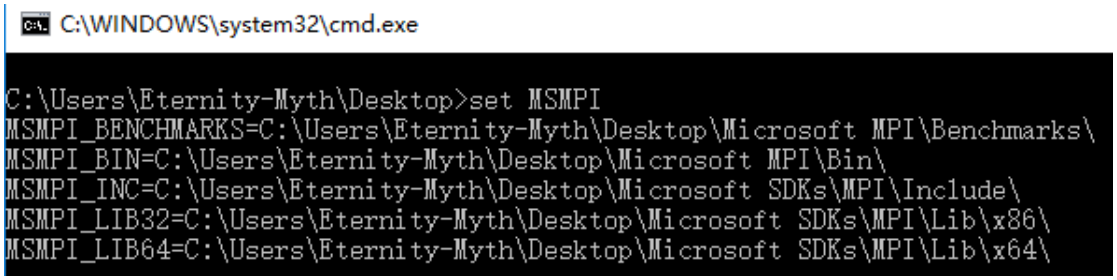
2. MS-MPI 环境配置：

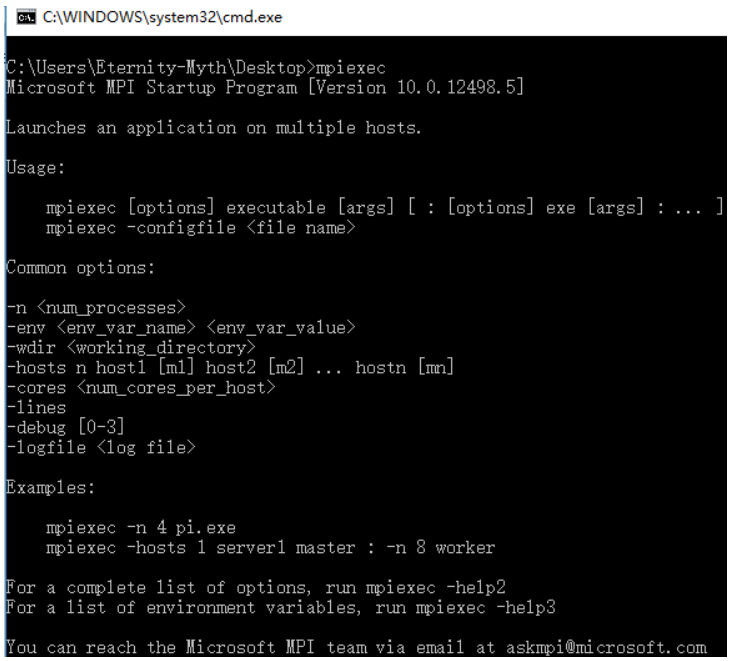
首先，到 Microsoft 官方网站：https://www.microsoft.com/en-us/download/details.aspx?id=57467 下载 MS-MPI，注意“msmpisdk.msi”和“msmpisetup.exe”两个文件都要下载，如图所示：



然后，分别进行安装。

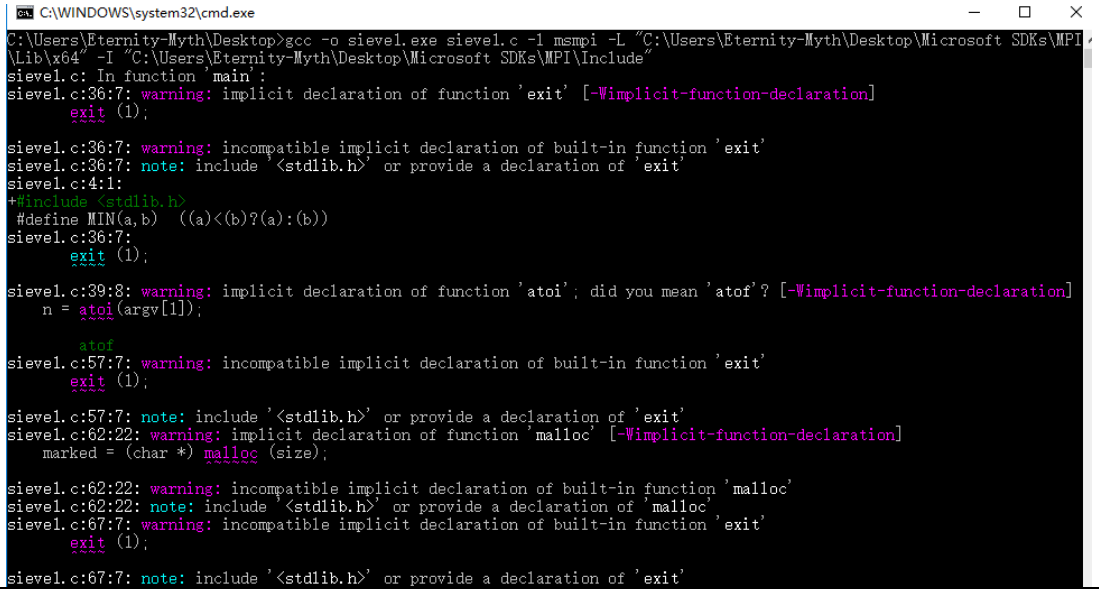
二者均安装完成后，运行 cmd.exe 打开命令行窗口，输入“set MSMPI”可以看到 MSMPI 环境变量已经自动配置完成。输入“mpiexec”可以正常输出相关信息，则 MSMPI 环境配置成功，如图所示：



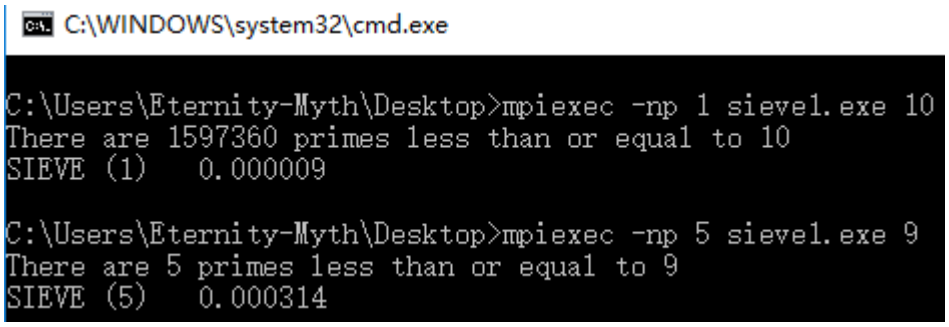


七、实验步骤及操作：

首先，将实验源代码保存并命名为“sieve1.c”，使用 gcc+命令行参数编译运行，将编译生成的可执行文件命名为“sieve1.exe”。具体命令：$ gcc -o sieve1.exe sieve1.c -l msmpi -L "C:\Users\Eternity-Myth\Desktop\Microsoft SDKs\MPI\Lib\x64" -I "C:\Users\Eternity-Myth\Desktop\Microsoft SDKs\MPI\Include" （说明：在使用 gcc 编译时，-o 参数用于自定义编译生成的文件名；-l 参数用来指定程序要链接的库，此处编译时需要链接 msmpi.lib 库；-L 参数用于指定程序编译时寻找库的路径；-I 参数用于指定引入的头文件所在目录。库文件与头文件所在目录可通过命令“set MSMPI”查看系统环境变量得到，注意在 64 位系统的机器上运行需要使用 x64 的库）



如图所示，编译出现了若干警告和提示信息，但无错误，程序编译成功。生成可执行文件“sieve1.exe”。接下来只要直接调用 mpiexec.exe 并指定 process 个数与筛选范围运行程序即可：命令“mpiexec -np n sieve1.exe number”可以指定启动 n 个 process 并在小于等于 number 范围的正整数范围筛选素数。通过指定参数运行程序，得到结果如下图所示



显然，可以发现，程序在这两次运行中都出现了不正确的结果，因此，需要对程序进行修改。

首先解决 process 数为 1 时的异常情况：

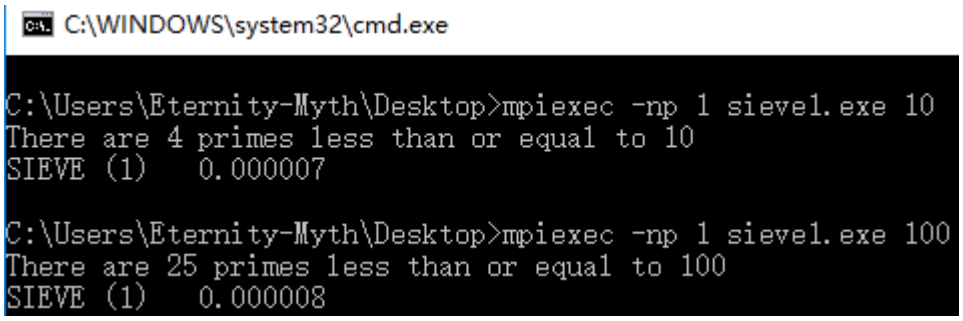
查看并分析源代码 sieve1.c，注意到在源代码的 90、91 行处只判断和处理了“p>1”的情况，如下图所示：

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

对此，在该语句后添加处理即可，修改后的程序语句如下图所示：

```
90      if (p > 1) MPI_Reduce (&count, &global_count, 1, MPI_INT, MPI_SUM,
91        0, MPI_COMM_WORLD);
92      else {
93       global_count = count;
94      }
```

修改后保存退出，按照之前所述命令重新编译并运行程序，指定参数-np 1 时，程序能够输出正确的结果，如下图所示：



接下来解决素数个数统计错误的问题：

查看并分析源代码 sieve1.c，注意到在统计素数的个数时，是通过判断数组 marked[]中每个元素的值是否为 0 来统计的。由经验猜测，对于这种统计个数相差 1 个的类似情况，原因往往是数组的初始化或者数组的下标问题，在某种特定的条件或边界值下会出错。进一步观察可以发现，marked[]数组分配内存空间位于源代码第 62 行，随后，在第 70 行，对数组中的元素赋予了初始值 0，如下图所示：

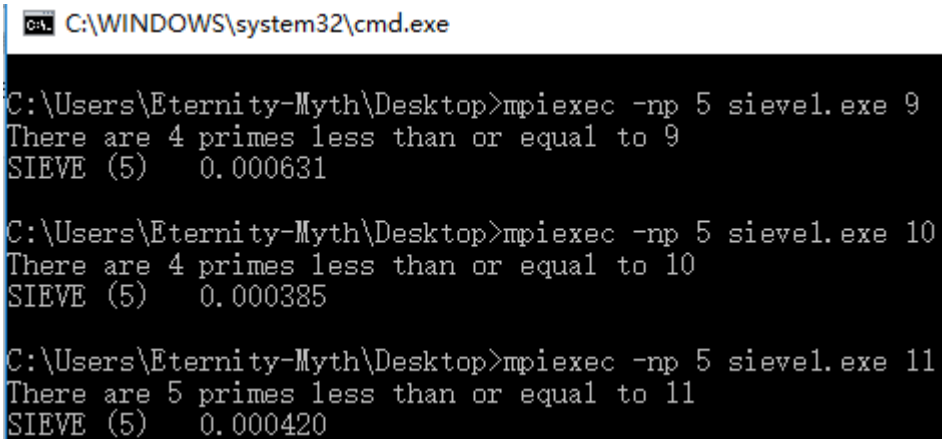
```
62      marked = (char *) malloc (size);
```

```
70    for (i = 0; i < size; i++) marked[i] = 0;
```

进一步调试后修改程序：注释掉或者直接删除第 70 行的赋值操作，并修改第 62 行代码如下：

```
62      marked = (char *) calloc(size, sizeof(char));
```

修改后保存退出，按照之前所述命令重新编译并运行程序，程序运行结果正确，如图所示：



至此，程序已经能够成功运行并且得到正确的结果，但我们可以进一步对程序进行拓展，使程序可以处理更大范围的数据：将源代码中的“low_value”与“high_value”变量修改为“long int”类型变量，并且，对源代码中第 45 行-47 行的代码（如下图所示）作如下修改：

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

（原始）

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

（修改后）

接下来对 sieve1 程序在不同的数据规模和不同进程配置下的性能进行测试。在本次测试中，测试进程数分别为 1、2、4、8 和 16，选取的数据规模有：1000、10000、100000、1000000、10000000 和 100000000。也就是说，一共需要测试 5*6=30 组数据。为避免偶然性，对于每组数据，均在相同的测试环境下测试 10 次，并取平均值作为最终结果。测试结果列表如下：

组号	进程数（np）	数据范围	10 次测试平均时间（秒）
1	1	1000	0.000015
2	1	10000	0.0000827
3	1	100000	0.0008434
4	1	1000000	0.0117579
5	1	10000000	0.1666059
6	1	100000000	1.9808614
7	2	1000	0.0000741
8	2	10000	0.0001282
9	2	100000	0.0006611
10	2	1000000	0.0065801
11	2	10000000	0.1108784
12	2	100000000	1.4257667
13	4	1000	0.0004401
14	4	10000	0.0005051
15	4	100000	0.0009826
16	4	1000000	0.0042348
17	4	10000000	0.0885877
18	4	100000000	1.2055877
19	8	1000	0.0012258
20	8	10000	0.0011222
21	8	100000	0.0028717
22	8	1000000	0.0051616

23	8	10000000	0.0653903
24	8	100000000	1.1574096
25	16	1000	0.0037298
26	16	10000	0.0042704
27	16	100000	0.0046095
28	16	1000000	0.0080269
29	16	10000000	0.0623539
30	16	100000000	1.1127653

表一：sieve1 程序测试结果表

根据表一结果，可以绘制 sieve1 程序在不同进程数与不同数据规模下的并行性能曲线图（篇幅所限，图与结果详细分析请参见：八、实验数据及结果分析 部分）

接下来，在 sieve1 程序的基础上，对算法进行性能优化。

优化思路 1 去掉待筛选偶数：

显然，利用已知除 2 以外的所有偶数都不是素数的常识，可以将待筛选数字总量减半，从而提高筛选效率。

基于以上思想，修改源代码 sieve1.c（注，由于代码改动处较多，此处不一一列举出，优化后的完整代码将在实验报告后的附录中给出），记经过第一次优化去掉待筛选偶数后的代码为 modification1.c。

接下来对 modification1 程序在不同的数据规模和不同进程配置下的性能进行测试。在本次测试中，测试进程数分别为 1、2、4、8 和 16，选取的数据规模有：1000、10000、100000、1000000、10000000 和 100000000。也就是说，一共需要测试 5*6=30 组数据。为避免偶然性，对于每组数据，均在相同的测试环境下测试 10 次，并取平均值作为最终结果。测试结果列表如下：

组号	进程数（np）	数据范围	10 次测试平均时间（秒）
1	1	1000	0.0000411
2	1	10000	0.0000791
3	1	100000	0.0004915

4	1	1000000	0.0056681
5	1	10000000	0.075265
6	1	100000000	0.9681718
7	2	1000	0.0000594
8	2	10000	0.0001057
9	2	100000	0.0003701
10	2	1000000	0.0032979
11	2	10000000	0.0469701
12	2	100000000	0.6943679
13	4	1000	0.0003699
14	4	10000	0.0003012
15	4	100000	0.0005021
16	4	1000000	0.0026812
17	4	10000000	0.0289978
18	4	100000000	0.5885879
19	8	1000	0.0011759
20	8	10000	0.0010012
21	8	100000	0.0015086
22	8	1000000	0.0031416
23	8	10000000	0.0265533
24	8	100000000	0.5455628
25	16	1000	0.0031812
26	16	10000	0.0039769
27	16	100000	0.0044567
28	16	1000000	0.0069956
29	16	10000000	0.0245085
30	16	100000000	0.4910188

表二：modification1 程序测试结果表

根据表二结果，可以绘制 modification1 程序在不同进程数与不同数据规模下的并行性

能曲线图（篇幅所限，图与结果详细分析请参见：八、实验数据及结果分析 部分）

优化思路 2 去掉广播通信：

初始的代码是通过进程 0 广播下一个筛选倍数的素数。进程之间需要通过 MPI_Bcast 函数进行通信。通信就一定会有开销，因此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。

假设每个任务除了分配到 n/p 个整数外,还拥有一个单独的数组，用于包含整数 $3,5,7,\cdots,\lfloor n \rfloor$。 在寻找 3~n 的素数之前， 每个任务可以先用串行算法计算出 3~$\sqrt{n}$的素数。一旦这一步骤完成， 每个任务就拥有了所有的数组，其中包含了所有 3~$\sqrt{n}$的素数。在这些任务可以在消除了广播步骤的情况下，对整个数组进行筛选。

基于以上思想，修改第一次优化后的代码 modification1.c（注，由于代码改动处较多，此处不一一列举出，优化后的完整代码将在实验报告后的附录中给出），记经过第二次优化，进一步去除广播通信后的代码为 modification2.c。

接下来对 modification2 程序在不同的数据规模和不同进程配置下的性能进行测试。在本次测试中，测试进程数分别为 1、2、4、8 和 16，选取的数据规模有：1000、10000、100000、1000000、10000000 和 100000000。也就是说，一共需要测试 5*6=30 组数据。为避免偶然性，对于每组数据，均在相同的测试环境下测试 10 次，并取平均值作为最终结果。测试结果列表如下：

组号	进程数（np）	数据范围	10 次测试平均时间（秒）
1	1	1000	0.0000315
2	1	10000	0.0000665
3	1	100000	0.0004293
4	1	1000000	0.0055843
5	1	10000000	0.0744138
6	1	100000000	0.9665156
7	2	1000	0.0000416

8	2	10000	0.0000659
9	2	100000	0.0002724
10	2	1000000	0.0026030
11	2	10000000	0.0447923
12	2	100000000	0.6876903
13	4	1000	0.0002676
14	4	10000	0.0002753
15	4	100000	0.0003567
16	4	1000000	0.0028537
17	4	10000000	0.0275986
18	4	100000000	0.5763973
19	8	1000	0.0009297
20	8	10000	0.0007689
21	8	100000	0.0010852
22	8	1000000	0.0028321
23	8	10000000	0.0198515
24	8	100000000	0.5180015
25	16	1000	0.0030115
26	16	10000	0.0034173
27	16	100000	0.0034758
28	16	1000000	0.0045161
29	16	10000000	0.0180183
30	16	100000000	0.4042959

表三：modification2 程序测试结果表

根据表三结果，可以绘制 modification2 程序在不同进程数与不同数据规模下的并行性能曲线图（篇幅所限，图与结果详细分析请参见：八、实验数据及结果分析 部分）

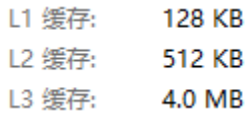
优化思路 3 分块筛选，提高 cache 命中率：

每个进程根据机器 Cache Block 的大小，将待筛选数据进一步分块，在每个块内使用 3-

sqrt(n)中的素数进行标记筛选，从而提高 cache 命中率。

并行筛法算法的大部分执行时间都用在对一个非常巨大的数组的分散的元素进行标记上了，造成了很差的 Cache 命中率。我们把算法的核心内容看成 2 个循环，外层循环为 3~$\sqrt{n}$的素数之间迭代， 内层循环在属于该进程的 3~n 的整数之间迭代。如果我们把内外层循环交换一下， 就可以改进程序的 Cache 命中率。我们可以将数组的一部分放入 Cache，标记其中所有小于$\sqrt{n}$的素数的倍数，然后再读入数组的下一个部分。

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：



由于测试中可能会涉及到比较大的数据范围，为保证统一，也为了方便进行比较，统一选取 L3 级缓存。由于在系统中，一个 int 类型数据占 4B，因此，L3 缓存共可以容纳 4*1024*1024/4=1048576 个 int 类型数据，因此，可以将 1048576 作为划分块的大小。

基于以上思想，再次修改第二次优化后的代码 modification2.c（注，由于代码改动处较多，此处不一一列举出，优化后的完整代码将在实验报告后的附录中给出），记经过第三次优化，重新组织循环并提升 Cache 命中率后的代码为 modification3.c。

接下来对 modification3 程序在不同的数据规模和不同进程配置下的性能进行测试。在本次测试中，测试进程数分别为 1、2、4、8 和 16，选取的数据规模有：1000、10000、100000、1000000、10000000 和 100000000。也就是说，一共需要测试 5*6=30 组数据。为避免偶然性，对于每组数据，均在相同的测试环境下测试 10 次，并取平均值作为最终结果。测试结果列表如下：

组号	进程数（np）	数据范围	10 次测试平均时间（秒）
1	1	1000	0.0000299
2	1	10000	0.0000713
3	1	100000	0.0004567
4	1	1000000	0.0056906
5	1	10000000	0.0726635

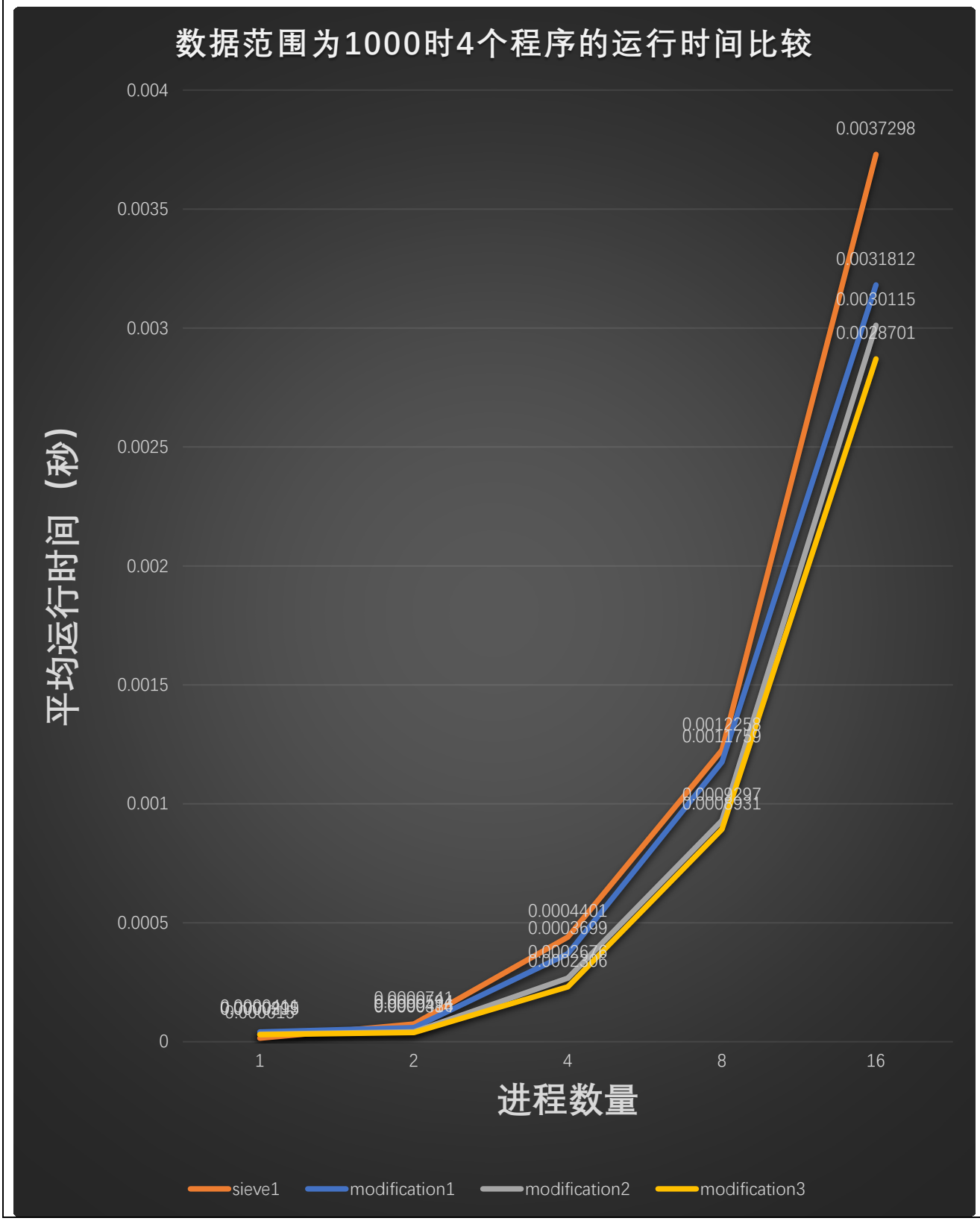
6	1	100000000	0.8029341
7	2	1000	0.0000384
8	2	10000	0.0000649
9	2	100000	0.0003198
10	2	1000000	0.0027058
11	2	10000000	0.0398478
12	2	100000000	0.4318809
13	4	1000	0.0002306
14	4	10000	0.0002322
15	4	100000	0.0004712
16	4	1000000	0.0022883
17	4	10000000	0.0264559
18	4	100000000	0.2665422
19	8	1000	0.0008931
20	8	10000	0.0006660
21	8	100000	0.0012072
22	8	1000000	0.0021580
23	8	10000000	0.0181239
24	8	100000000	0.2044096
25	16	1000	0.0028701
26	16	10000	0.0025690
27	16	100000	0.0026986
28	16	1000000	0.0037125
29	16	10000000	0.0160945
30	16	100000000	0.1731379

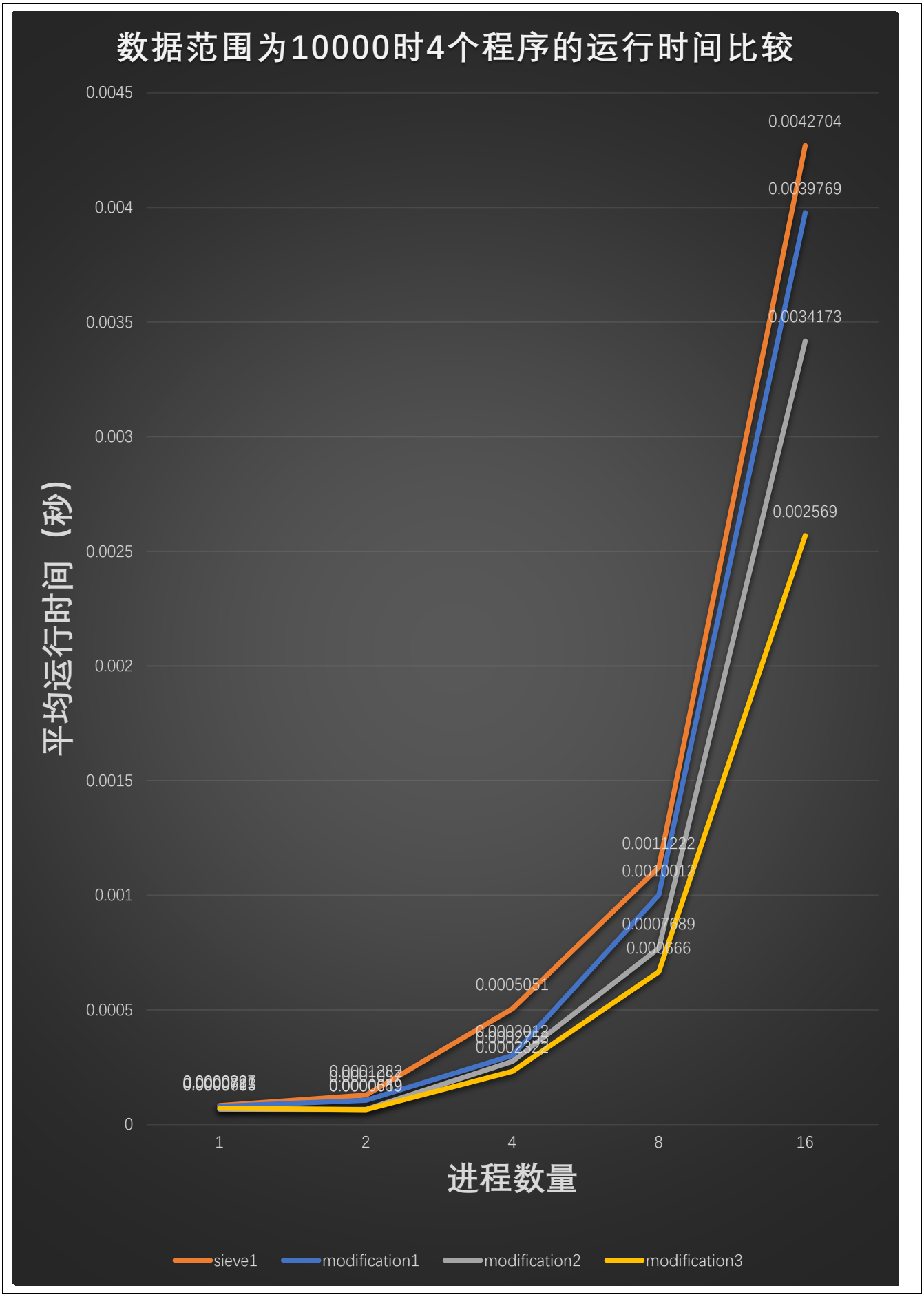
表四：modification3 程序测试结果表

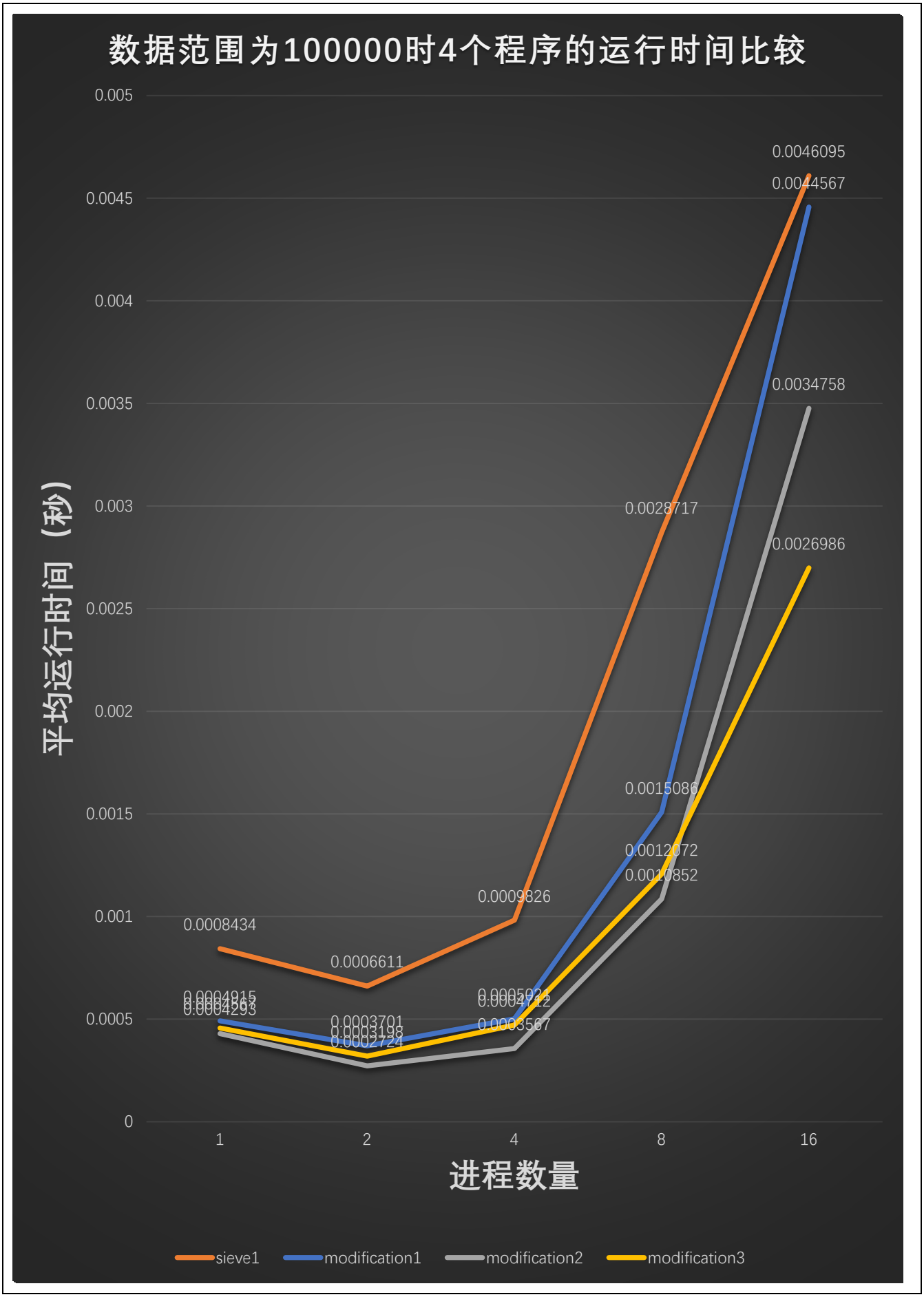
根据表四结果，可以绘制 modification3 程序在不同进程数与不同数据规模下的并行性能曲线图（篇幅所限，图与结果详细分析请参见：八、实验数据及结果分析 部分）

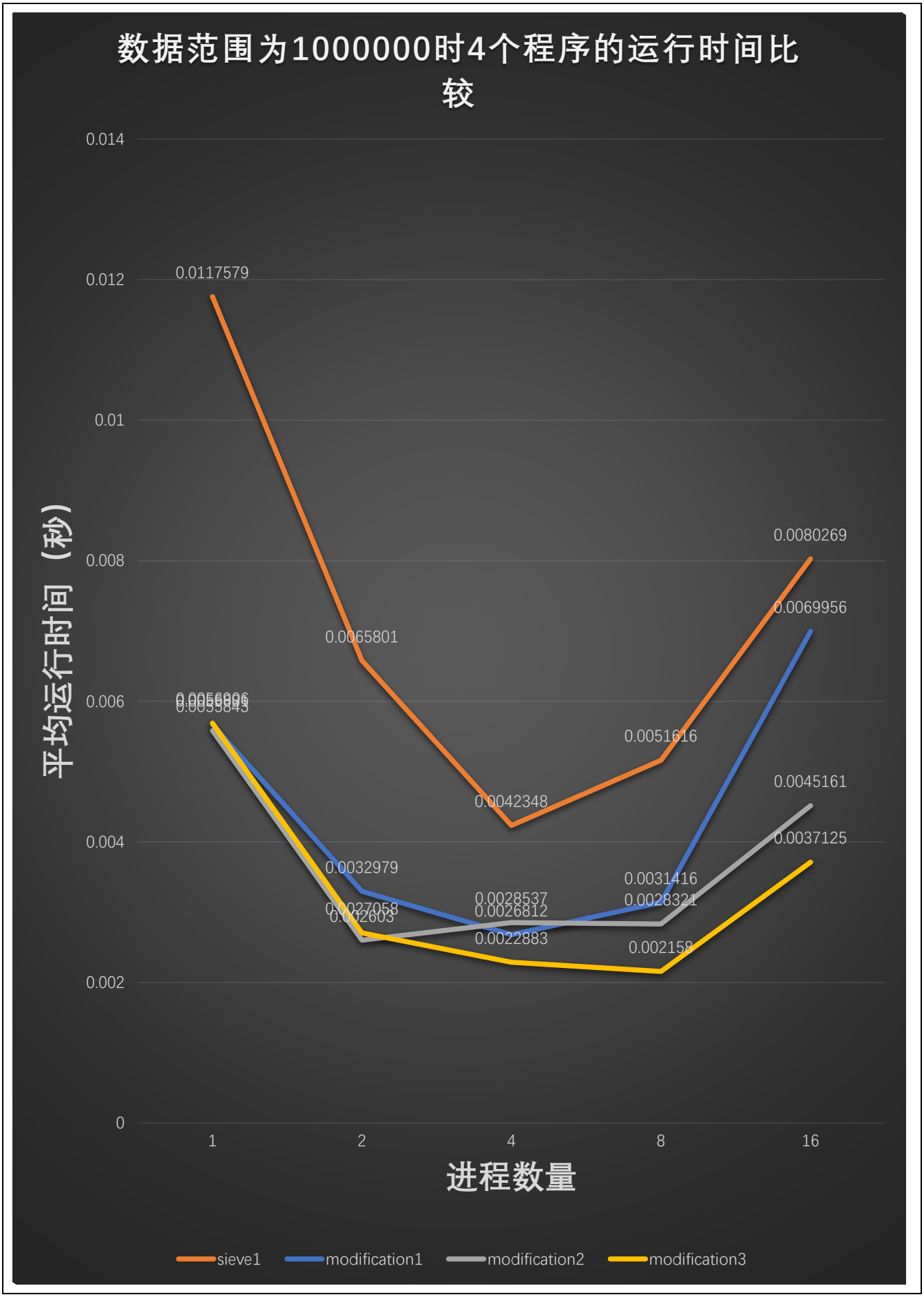
八、实验数据及结果分析：

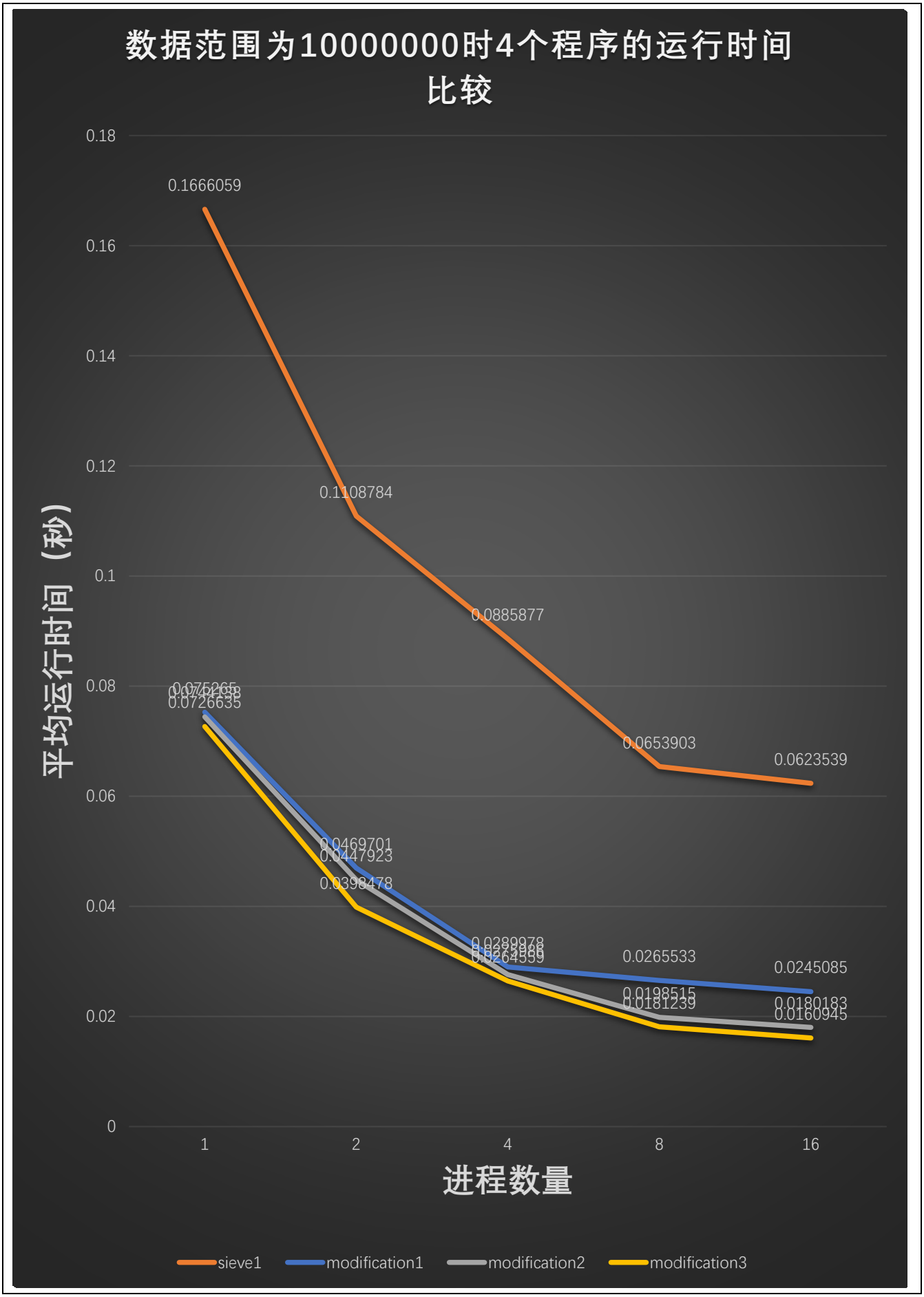
根据以上四个表格的实验数据结果，可以绘制 4 个程序在不同数据规模以及不同进程数量下的运行性能曲线。分别固定数据规模，在同一张图中作出 4 个程序的折线图如下：

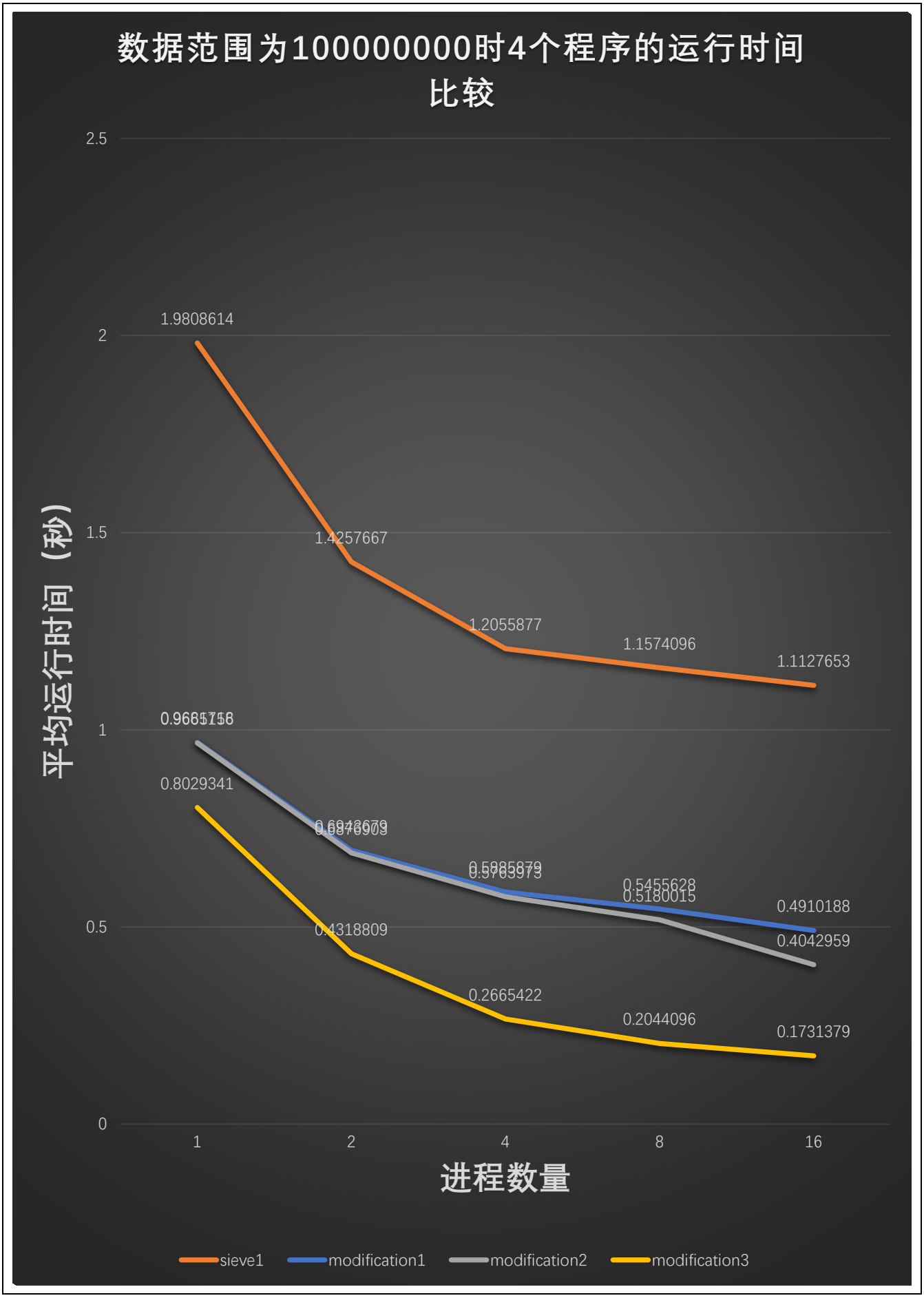



数据范围为10000时4个程序的运行时间比较
0.0045
0.004
0.0035
0.003
0.0025
0.002
0.0015
0.001
0.0005
0
平均运行时间（秒）
0.0042704
0.0039769
0.0034173
0.002569
0.0011222
0.0010012
0.0007689
0.000566
0.0005051
0.0003013
0.0002329
0.0001282
0.0000669
1
2
4
8
16
进程数量
sieve1
modification1
modification2
modification3


数据范围为100000时4个程序的运行时间比较
平均运行时间（秒）
0.005
0.0045
0.004
0.0035
0.003
0.0025
0.002
0.0015
0.001
0.0005
0
1
2
4
8
16
进程数量
0.0046095
0.0044567
0.0034758
0.0028717
0.0026986
0.0015086
0.0012072
0.0010852
0.0009826
0.0008434
0.0006611
0.0005021
0.0004915
0.0004567
0.0004293
0.0004142
0.0003701
0.0003567
0.0003198
0.0002724
sieve1
modification1
modification2
modification3


数据范围为1000000时4个程序的运行时间比较
0.014
0.012
0.01
0.008
0.006
0.004
0.002
0
平均运行时间（秒）
0.0117579
0.0065801
0.0042348
0.0051616
0.0080269
0.0069956
0.0045161
0.0037125
0.0032979
0.0031416
0.0028537
0.0028321
0.0026812
0.0022883
0.002158
1
2
4
8
16
进程数量
sieve1
modification1
modification2
modification3


数据范围为10000000时4个程序的运行时间比较
平均运行时间（秒）
0.18
0.16
0.14
0.12
0.1
0.08
0.06
0.04
0.02
0
0.1666059
0.1108784
0.0885877
0.0653903
0.0623539
0.0726635
0.0469701
0.0447923
0.0398478
0.0265533
0.0245085
0.0198515
0.0181239
0.0180183
0.0160945
1
2
4
8
16
进程数量
sieve1
modification1
modification2
modification3


数据范围为100000000时4个程序的运行时间比较
2.5
2
1.5
1
0.5
0
平均运行时间（秒）
1.9808614
1.4257667
1.2055877
1.1574096
1.1127653
0.8029341
0.4318809
0.2665422
0.2044096
0.1731379
0.4910188
0.4042959
1
2
4
8
16
进程数量
sieve1
modification1
modification2
modification3

通过分析以上图表数据，可以得出，在相同的数据规模和进程数量条件下，从总体上看，每一次优化后的程序性能都要优于之前的程序，也就是说从总体的性能上看：sieve1<modification1<modification2<modification3。这是符合预期的实验结果，该结果也证明了以上所进行的每一次优化是行之有效的，其作用也直接体现在了程序平均运行时间的减少上。因此，至此实验基本取得成功。

进一步分析以上图表数据，还可以得出更多的结论与信息。可以注意到当数据的规模不太大时（如 1000000 以内），无论是原程序还是优化后的任一程序，其平均运行时间都随着进程数增多而增加。也就是说，在数据规模较小且一定时，增加进程数未必能提升性能加速计算，相反，反而还可能使程序运行的时间延长。出现这种情况可能是因为系统创建进程、管理进程以及调度进程具有一定的开销（这种开销体现在系统资源上和时间消耗上），在任务量比较小时，多创建进程会使得系统主要的开销都花在这些进程上，反而是不利于计算的。也就是说，此时，多进程并行计算带来的收益微乎其微，远远不及系统资源的开销，造成了系统资源的浪费。所以，在计算数据规模比较小时，并不适宜开启太多进程计算。

在计算数据规模足够大时，优化算法以及多进程并行计算带来的性能提升就显得非常明显。比较最初始版本 sieve1 程序与进行了第一次优化去掉偶数后的 modification1 程序，可以发现筛除偶数后的程序运行时间几乎比原来减少了一半。因为去除偶数实际上直接让数据规模减少了一半，也就是程序的计算量直接减少了一半，在数据规模足够大可以使程序性能充分发挥时，带来的性能提升就非常可观。

进一步比较第一次优化后的程序 modification1 与第二次优化去除广播通信的程序 modification2，这组比较相对于前面一组的比较来说，性能的提升不是那么明显，二者的性能差距并不是很大，在计算数据规模不是很大时或是进程数不多时性能很接近。从原理上来理解，进程间的广播通信时间消耗相比于计算时间消耗并不大，本身可优化和性能提升的空间就比较有限，另外，由于在测试时指定的进程数都比较少，因此不能完全发挥其作用。可能当数据规模足够大并且进程数足够多时，这种优化才能体现较为明显的性能提升。

最后比较以上程序与最终优化的程序 modification3，可以看出，通过提升 Cache 命中率来对程序进行优化同样需要计算数据规模足够大才能发挥明显的作用。从原理上来理解，由于机器的 Cache 块本身能容纳一定量的数，如果数据量明显小于 Cache 块容纳量，那么 Cache 块就可以放得下所有的数，也就不存在所谓的 Cache miss 情况，这时也就相当于没有提升。只有数据量足够大时，才需要考虑数据在 Cache 块的存放，此时才需要对数据进行分块，从

而通过减少 Cache miss，提高 Cache 命中率来提升性能。

总结来说，优化算法对程序性能的提升是行之有效的，因此，针对不同任务，应该设计和采用有效的算法提升性能。但是，程序的性能也受到其它方面的制约，应该综合考虑数据规模以及机器和系统本身的性能和资源量来决定，这往往需要经过尝试。

九、实验结论：

1.针对不同的具体任务，应该先分析问题，并设计和采用高效的算法，这有利于程序性能的提升。好的算法对程序性能的提升是非常明显的，能达到事半功倍的效果。

2.并不是越多进程并行计算就越好，不应该盲目应用多进程。对于计算数据规模较小的任务，多进程反而会造成系统资源的浪费，也影响程序运行时间和效率，可能事倍功半。

3.在不同情况、不同环境下，同样的程序性能可能有明显差异，应该综合考虑任务数据规模、机器软硬件、系统当前状态、性能和资源情况来决定应该如何应用并行程序，需要经验和尝试。

十、总结及心得体会：

分布式并行计算让计算机能够处理很大规模的问题，在现如今的应用是十分广泛的。通过本次实验，结合课程所学知识，我第一次接触、认识、学习了分布式并行计算程序，也进行了并行计算程序的编写、修改、调试和应用，拓宽了知识面的同时，锻炼了编程能力与动手能力。通过实践，锻炼了解决问题、分析数据、提炼结论的能力，获益匪浅。

十一、*对本实验过程及方法、手段的改进建议：

暂无

报告评分：

指导教师签字：

附录：实验程序源代码

一、sieve1.c（已修复已知 Bug）

```
#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' */

    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 + id*(n-1)/p;
    high_value = 1 + (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 *) calloc(size, sizeof(char));

    if (marked == NULL) {
        printf ("Cannot allocate enough memory\n");
        MPI_Finalize();
```

```
        exit (1);
    }

    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++;
    if (p > 1) MPI_Reduce (&count, &global_count, 1, MPI_INT, MPI_SUM,
        0, MPI_COMM_WORLD);
    else{
        global_count = 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, n);
        printf ("SIEVE (%d) %10.6f\n", p, elapsed_time);
    }
    MPI_Finalize ();
    return 0;
}
```

二、modification1.c

```
/* Modification 1: 筛除除 2 以外的偶数 */

#include <mpi.h>
#include <math.h>
#include <stdio.h>
#define MIN(a,b) ((a)<(b) ? (a) : (b))
#define BLOCK_LOW(id,p,n)    ((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)

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;
int        loc;

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]);
m = (n-3)/2 + 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 * BLOCK_LOW(id, p, m) + 3;
high_value = 2 * BLOCK_HIGH(id, p, m) + 3;
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 ((2*(proc0_size-1) + 3) < (int) sqrt((double) n)) {
        if (!id) printf("Too many processes\n");
        MPI_Finalize();
        exit(1);
  }

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

  marked = (char *) calloc(size, sizeof(char));

  if (marked == NULL) {
        printf("Cannot allocate enough memory\n");
        MPI_Finalize();
        exit(1);
  }

  if (!id) index = 0;
  prime = 3;
  do {
        if (prime*prime > low_value)
              first = (prime*prime-3)/2 - (low_value-3)/2;
        else {
              loc = low_value % prime;
              if (!loc) first = 0;
              else {
                    first = prime - loc;
                    if (!((low_value+first)%2))
                          first = (first+prime)/2;
                    else first /= 2;
              }
```

```
            }
            for (i=first; i <size; i+=prime)
                marked[i] = 1;
            if (!id) {
                while (marked[++index]);
                prime = 2*index + 3;
            }
            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;
}
```

三、modification2.c

```
/* Modification 1:  筛除除 2 以外的偶数  */
/* Modification 2:  去掉广播通信  */

#include <mpi.h>
#include <math.h>
#include <stdio.h>
#define MIN(a,b) ((a)<(b) ? (a) : (b))
#define BLOCK_LOW(id,p,n)    ((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)

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;
    int       loc;
    char    *primes;
    int       primes_size;

     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 <n>\n", argv[0]);
        MPI_Finalize();
        exit(1);
}

n = atoi(argv[1]);
m = (n-3)/2 + 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 * BLOCK_LOW(id, p, m) + 3;
high_value = 2 * BLOCK_HIGH(id, p, m) + 3;
size = BLOCK_SIZE(id, p, m);

/* Allocate this process' 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 = (sqrt(n) - 3)/2 + 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;

index = 0;
prime = 3;
do {
        for (i = (prime*prime-3)/2; i < primes_size; i += prime)
                primes[i] = 1;
        while (primes[++index]);
        prime = 2*index + 3;
} while (prime*prime <= sqrt(n));

index = 0;
prime = 3;
do {
        if (prime*prime > low_value)
                first = (prime*prime-3)/2 - (low_value-3)/2;
        else {
                loc = low_value % prime;
                if (!loc) first = 0;
                else {
                        first = prime - loc;
                        if (!((low_value+first)%2))
                                first = (first+prime)/2;
                        else first /= 2;
                }
        }
        for (i=first; i <size; i+=prime)
                marked[i] = 1;
        while (primes[++index]);
        prime = 2*index + 3;
```

```
    } 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;
}
```

四、modification3.c

```
/* Modification 1: 筛除除 2 以外的偶数 */
/* Modification 2: 去掉广播通信 */
/* Modification 3: 重新组织循环，提高 cache 块命中率 */

#include <mpi.h>
#include <math.h>
#include <stdio.h>
#define MIN(a,b) ((a)<(b) ? (a) : (b))
#define BLOCK_LOW(id,p,n)    ((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)

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;
    int       loc;
    char    *primes;
    int       primes_size;
    int       lv;
    int       sec;
    int       chunk;

    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 != 3) {
        if (!id) printf("Command line: %s <n> <chunk>\n", argv[0]);
        MPI_Finalize();
        exit(1);
}

n = atoi(argv[1]);
m = (n-3)/2 + 1;
chunk = atoi(argv[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 = 2 * BLOCK_LOW(id, p, m) + 3;
high_value = 2 * BLOCK_HIGH(id, p, m) + 3;
size = BLOCK_SIZE(id, p, m);

/* Allocate this process' 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 = (sqrt(n) - 3)/2 + 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;

index = 0;
prime = 3;
do {
        for (i = (prime*prime-3)/2; i < primes_size; i += prime)
                primes[i] = 1;
        while (primes[++index]);
        prime = 2*index + 3;
} while (prime*prime <= sqrt(n));

for (sec = 0; sec < size; sec += chunk) {
        index = 0;
        prime = 3;
        lv = 2*((low_value-3)/2+sec)+3;
        do {
                if (prime*prime > lv)
                        first = (prime*prime-3)/2 - (lv-3)/2;
                else {
                        loc = lv % prime;
                        if (!loc) first = 0;
                        else {
                                first = prime - loc;
                                if (!((lv+first)%2))
                                        first = (first+prime)/2;
                                else first /= 2;
                        }
                }
                for (i = first+sec; i < first+sec+chunk && i < size; i += prime)
                        marked[i] = 1;
                while (primes[++index]);
                prime = 2*index + 3;
        } 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;
}
```