电 子 科 技 大 学

实 验 报 告

学生姓名：董文龙	学 号：2018081309003

一、实验室名称：立人楼 B209

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：

埃拉托斯特尼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

其伪代码如下图 1 所示：

```
Create list of unmarked natural numbers 2, 3, …, n
k <= 2
Repeat
   Mark all multiples of k between k^2 and n
   k<=smallest unmarked number > k
until k^2 > n
The unmarked numbers are primes
```

图 1

四、实验目的：

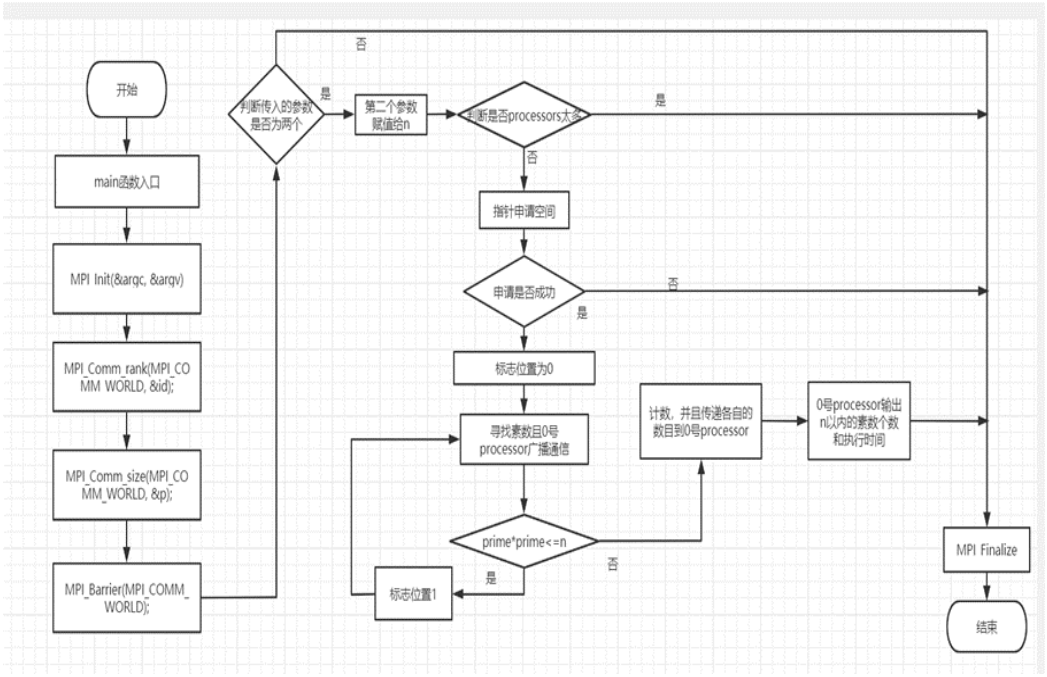
1. 使用 MPI 编程实现埃拉托斯特尼筛法并行算法。
2. 对程序进行性能分析以及调优。

五、实验内容：

本实验分为四个部分：

1. 安装部署 MPI 实验环境，并调试完成基准代码，并实测在不同进程规模（1，2，4，8，16）加速比，并合理分析原因。
2. 完成优化 1，去除偶数优化，并实测在不同进程规模（1，2，4，8，16）加速比，并合理分析原因。
3. 完成优化 2，消除广播优化，并实测在不同进程规模（1，2，4，8，16）加速比，并合理分析原因。
4. 完成优化 3，cache 优化，并实测在不同进程规模（1，2，4，8，16）加速比，并合理分析原因。
5. 性能得分：在完成优化 3 的基础上，可以利用课内外知识，全面优化代码性能。

程序基础框图如下图所示：



六、实验器材（设备、元器件）：

Windows：

CPU: Inte(R) Core(TM) i7-8550U CPU @ 1.80GHz 2.00GHZ

内存：16G

开发环境：CLion 2020.1、OpenMpi、Cygwin64。

七、实验步骤及操作：

1.Cygwin64 的安装

1.1 首先进入官网（http://www.cygwin.com/），找到 setup-x86_64.exe 文件进行下载并且安装，全部选用默认选项，Root Directory 我存放的位置为 D:\cygwin64，Local Package Directory 我存放的位置为 D:\cygwin_repertory，采用的镜像网为(https://mirrors.tuna.tsinghua.edu.cn) 。
1.2 cygwin 模块安装如下图 2 所示，由于 CLion 的 gdb 版本不支持 cygwin 中太高的版本，所以这里下载了可以供 Clion 使用的 8.0.1 版本。

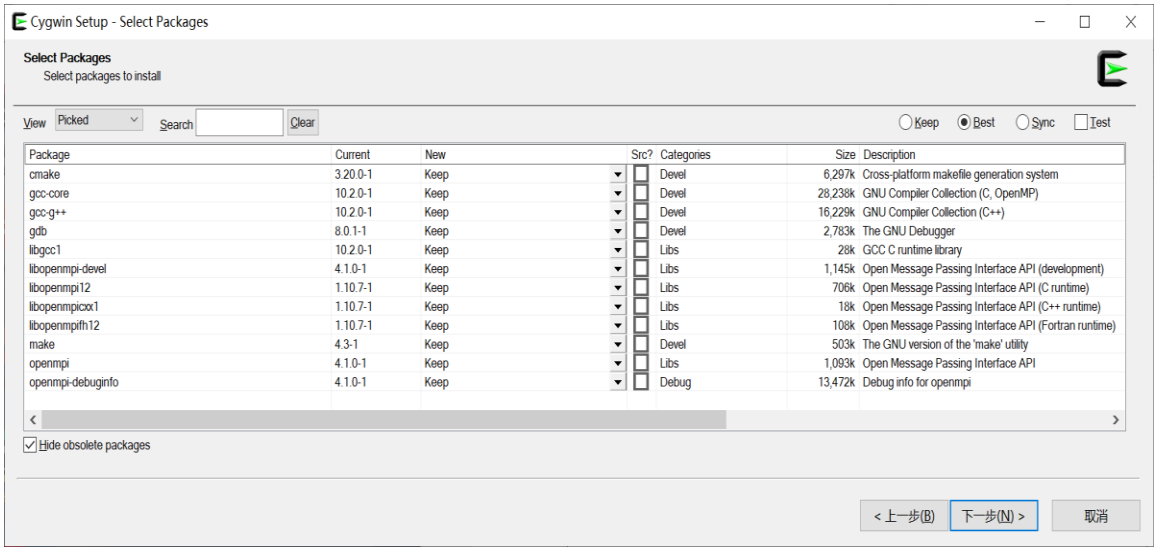


图 2

1.3 安装完成之后，可以在 Cygwin64 Terminal 中输入 mpicc,mpirun -np 命令，出现下图 3 所示

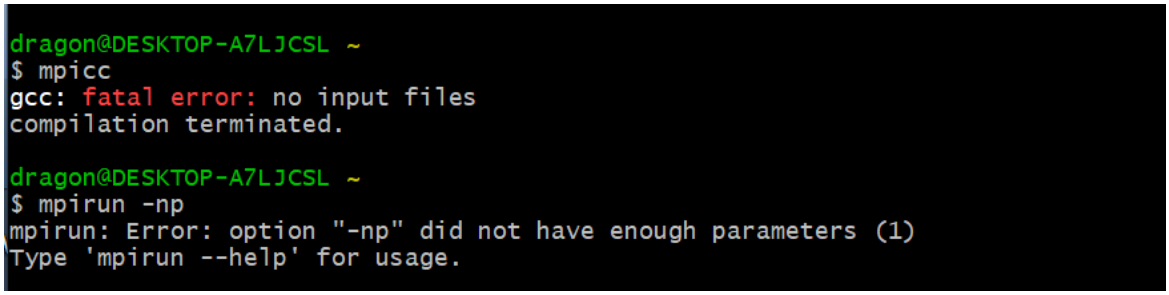


图 3

2. Clion 的安装

首先进入官网(https://www.jetbrains.com/clion/),进行下载 exe 文件，之后进行安装。

3. 配置外部工具 mpicc 和 mpirun

3.1. File->Tools->External Tools,点击“+”创建 MPI 的编译和执行工具。总体效果如图 4 所示（在 3.2 和 3.3 完成之后）。

图 4

3.2.配置 MPI complier。(Working directory 内容为$ProjectFileDir$)，如图 5 所示。

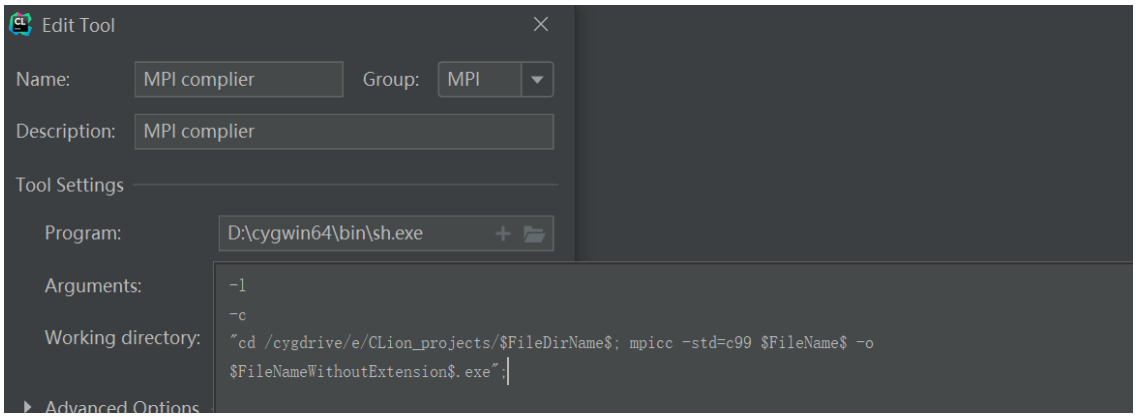


图 5

3.3.配置 MPI runner。(Working directory 内容为$ProjectFileDir$)，如图 6 所示。这里的 100000000 是后面所需要求的问题规模，即 1 亿以内的素数。

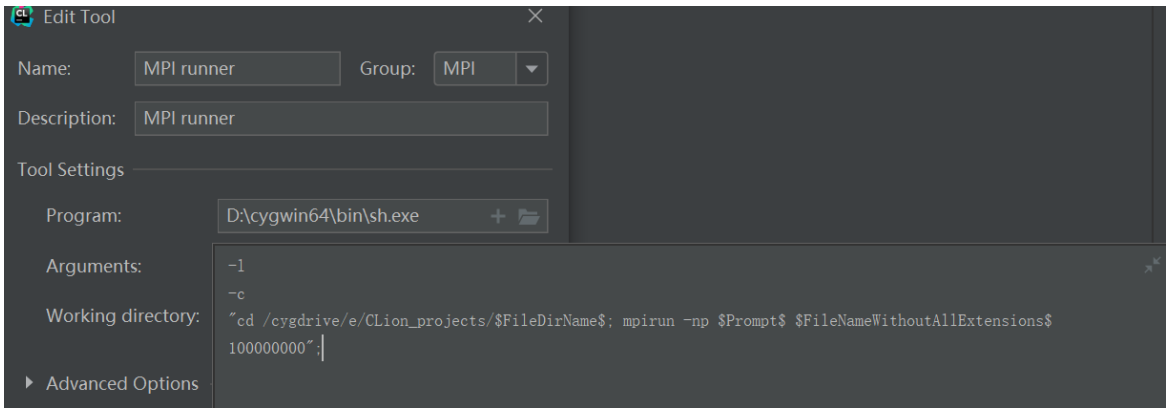


图 6

注：上述两个配置中的 e/CLion_projects 为所要创建的工程目录（E:\CLion_projects）

4.创建项目

4.1.在工程目录中创建项目(E:\CLion_projects\MPI_project)。

4.2. CLion 的 File->Settings->Tools->Build,Execution,Depolyment->Toolschains,配置如下图 7 所示。

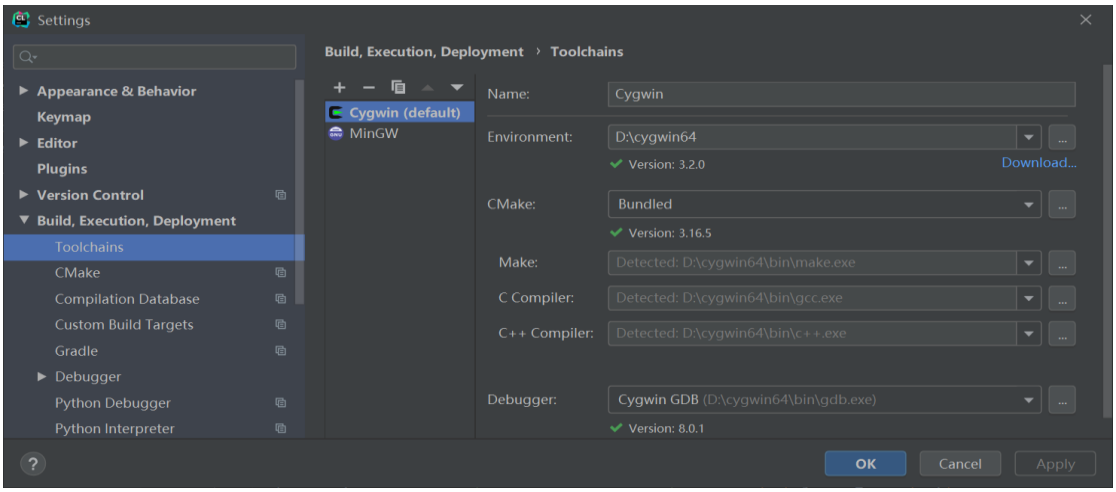


图 7

现在可以使用 include<mpi.h>了，并且可以在 clion 中进行 mpi 的编译和运行工作了。如图 8 所示。

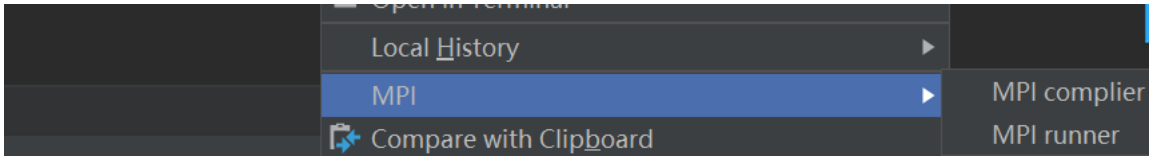


图 8

5. 基准程序

程序说明（复制代码时候黑色的背景太黑了，我这里及以后所有的代码部分更改了主题为 Github）：

```
#include "mpi.h"
#include <math.h>
#include <stdio.h>
#include <stdlib.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' */
    //Init，MPI 程序启动时“自动”建立两个通信器：MPI_COMM_WORLD:包含程序中所有 MPI 进程，MPI_COMM_SELF：有单个进程独自构成，仅包含自己
    MPI_Init(&argc, &argv);
    /* Start the timer */
    //MPI_COMM_RANK 得到本进程的进程号，进程号取值范围为 0, ..., np-1
    MPI_Comm_rank(MPI_COMM_WORLD, &id);
    // MPI_COMM_SIZE 得到所有参加运算的进程的个数
    MPI_Comm_size(MPI_COMM_WORLD, &p);
    // MPI_Barrier 是 MPI 中的一个函数接口,表示阻止调用直到 communicator 中所有进程完成调用
    MPI_Barrier(MPI_COMM_WORLD);
    //表示从过去某一时刻到调用时刻所经历的时间
```

```
elapsed_time = -MPI_Wtime();
//判断参数是否为 2，第一个参数为文件名字，第二个参数为要寻找 n 以内的数
if (argc != 2) {
        if (!id) {
                // 结束 MPI 系统
                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;//进程的第一个数
high_value = 1 + (id + 1) * (n - 1) / p;//进程的最后一个数
size = high_value - low_value + 1;//进程处理的大小

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

//进程如果太多
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);
}
//假设所有的数都为素数，mark 数组全部置为 0
for (i = 0; i < size; i++) {
```

```
            marked[i] = 0;
        }

        //id=0 的 processor 的 index 置位 0
        if (!id) {
            index = 0;
        }

        prime = 2;
        do {
            //寻找到第一个素数倍数的索引 first
            if (prime * prime > low_value) {
                first = prime * prime - low_value;//索引值为 prime*prime-low_value

            } else {
                if (!(low_value % prime)) first = 0;//若 low_value 就是素数的倍数，索引值为 0
                else first = prime - (low_value % prime);//若 low_value 不是素数的倍数，索引值为
prime-(low_value%prime)
            }

            //  从第一个素数的倍数开始，标记该素数的倍数为非素数
            for (i = first; i < size; i += prime) {
                marked[i] = 1;
            }
            //id=0 的 processor 直到 mark 数组中的标志为 0，index+2 赋值给 prime，因为 index
从 0 开始，prime 从 2 开始
            if (!id) {
                while (marked[++index]);
                prime = index + 2;
            }
            //若 processors 的数量大于 1，id=0 的 processor 把素数广播出去。
            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++;
        }
        //若 processors 的数量大于 1，id=0 的 processor 复杂接受其他 id 的 processor 的 count，
并合为 global_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;
}
```

调试说明：

1. 运行过程中发现如果是单进程的话，golbal_count 的值运行有误，所以对代码进行了更改，为

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

2. 在运行 8 个核的时候，会报错 There are not enough slots available in the system to satisfy the 8 slots，于是改变命令，将 mpirunner 的参数改为（后面也采用如下参数了）：

```
-l-c"cd /cygdrive/e/CLion_projects/$FileDirName$; mpirun -oversubscribe -np $Prompt$ $FileNameWithoutAllExtensions$ 100000000";
```

程序运行结果见实验结果及数据分析。

6. 优化一：去掉偶数

程序说明：

进行去掉偶数的操作，因为把一半的空间用来存放偶数没有意义，所以仅处理奇数可以将所需空间减少一半，并把标记特定素数倍数的速度提高一倍。 代码如下：

```
#include "mpi.h"
#include <math.h>
#include <stdio.h>
#include <stdlib.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' */
    int low_index;      /* Lowest index on this proc */
    int high_index;     /* Highest index on this proc */
    //Init，MPI 程序启动时“自动”建立两个通信器：MPI_COMM_WORLD:包含程序中所有 MPI 进程，MPI_COMM_SELF：有单个进程独自构成，仅包含自己
    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);
    //开始计时
    elapsed_time = -MPI_Wtime();
    //判断参数是否为 2，第一个参数为文件名字，第二个参数为要寻找 n 以内的数
    if (argc != 2) {
        if (!id) {
            // 结束 MPI 系统
            printf("Command line: %s <m>\n", argv[0]);
        }
        MPI_Finalize();
        exit(1);
    }
    //计算 3 到 N 之间的数
    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 */
    int N = (n - 1) / 2;
    low_index = id * (N / p) + MIN(id, N % p); // 进程的第一个数的索引
    high_index = (id + 1) * (N / p) + MIN(id + 1, N % p) - 1; // 进程的最后一个数的索引
    low_value = low_index * 2 + 3; //进程的第一个数
    high_value = (high_index + 1) * 2 + 1;//进程的最后一个数
    size = (high_value - low_value) / 2 + 1;       //进程处理的数组大小
```

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

//进程如果太多
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 = 3;
do {
      if (prime * prime > low_value) {
            first = (prime * prime - low_value)/2;

      } else {
            if (!(low_value % prime)) first = 0;
            else if (low_value % prime % 2 == 0)
                  first = prime - ((low_value % prime)/2);
            else
                  first=(prime- (low_value % prime)) /2;

      }
      //筛选奇数
      for (i = first; i < size; i += prime) marked[i] = 1;
      if (!id) {
            while (marked[++index]);
```

```
                prime = index*2+3;
            }
            if (p > 1)
                MPI_Bcast(&prime, 1, MPI_INT, 0, MPI_COMM_WORLD);
        } while (prime * prime <= n);
        //分别计算局部数组中素数的个数
        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;
        if (n>=2)
            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, n);
            printf("SIEVE (%d) %10.6f\n", p, elapsed_time);
        }
        MPI_Finalize();
        return 0;
}
```

调试说明：

第一次运算出来的数字与实际所不符，后来发现是因为在去偶数的过程中，把 2 也去掉了，导致测出来的数据与实际不符。

程序运行结果见实验结果及数据分析。

7. 优化二：消除广播

程序说明：

让一个进程计算 k, 然后广播到其他进程的方法令这个步骤并行化，在程序执行的时候，这个广播重复了大概 n^1/2 / ln(n^1/2)次，故消除广播会提高并行程序的性能，代码如下：

```
#include "mpi.h"
#include <math.h>
#include <stdio.h>
#include <stdlib.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' */
char *marked_sub;       /*the marked array*/
int first_sub;          /*Subject of adjunct array*/
int num;                    /*the num of 2 to n*/
int sqrt_n;

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);
//开始计时
elapsed_time = -MPI_Wtime();
//判断参数是否为 2，第一个参数为文件名字，第二个参数为要寻找 n 以内的数
if (argc != 2) {
      if (!id) {
            // 结束 MPI 系统
            printf("Command line: %s <m>\n", argv[0]);
      }
      MPI_Finalize();
      exit(1);
}

n = atoi(argv[1]);
sqrt_n=(int) sqrt(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 = 2 + id * (n - 1) / p;//进程的第一个数
    high_value = 1 + (id + 1) * (n - 1) / p;//进程的最后一个数
    size = high_value - low_value + 1;//进程处理的大小
    num=(int) sqrt(high_value);

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

    //进程如果太多
    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);
    marked_sub = (char *) malloc(num);
    if (marked == NULL||marked_sub==NULL) {
        printf("Cannot allocate enough memory\n");
        MPI_Finalize();
        exit(1);
    }

    for (i = 0; i < size; i++) marked[i] = 0;
    for (i = 0; i < num; i++) marked_sub[i] = 0;

    index = 0;

    prime = 2;
    do {
        //寻找到第一个素数倍数的索引 first
        if (prime * prime > low_value) {
            first = prime * prime - low_value;//索引值为 prime*prime-low_value

        } else {
            if (!(low_value % prime)) first = 0;//若 low_value 就是素数的倍数，索引值为 0
            else first = prime - (low_value % prime);//若 low_value 不是素数的倍数，索引值为
prime-(low_value%prime)
        }
```

```
            // 从第一个素数的倍数开始，标记该素数的倍数为非素数
            for (i = first; i < size; i += prime) marked[i] = 1;
            first_sub =(prime*prime-2);
            for(i=first_sub;i<num;i+=prime) marked_sub[i]=1;
            while (marked_sub[++index]);
            prime = index + 2;
            //若 processors 的数量大于 1，id=0 的 processor 把素数广播出去。
        } while (prime <=sqrt_n);

        count = 0;
        for (i = 0; i < size; i++) {
            if (!marked[i]) count++;
        }
        //若 processors 的数量大于 1，id=0 的 processor 复杂接受其他 id 的 processor 的 count，
并合为 global_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;
}
```

调试说明：

本次优化无特别现象。

程序运行结果见实验结果及数据分析。

8. 优化三：Cache 优化(此次优化在优化 1 和优化 2 的基础上完成)

程序说明：

进行循环组织，并行算法大部分时间都用在对一个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，所以将内外层循环交换一下，就可以改进程序的 cache 命中率（循环语句嵌套时，大循环为内循环时，效率更高），可以通过计算得出，代码如下：

```
#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>
```

```
#include <math.h>
#include <string.h>
//The default cache size set 8MB
int main(int argc, char* argv[]){
	int count;                  /* Local prime count */
	double elapsed_time;        /* Parallel execution time */
	int global_count;           /* Global prime count */
	int high_value;             /* Highest value on this proc */
	int id;                     /* Process ID number */
	int index;                  /* Index of current prime */
	int low_value;              /* Lowest value on this proc */
	char* marked;               /*The marked odd from 3 to n*/
	int n;                      /* Sieving from 2, ..., 'n' */
	int p;                      /* Number of processes */
	int proc0_size;             /* Size of proc 0's subarray */
	char* labeled;              /*The signal from 2 to sqrt(n) with no even*/
	int* storage;               /*The prime store 2 to sqrt n*/
	int prime;                  /* Current prime */
	int size;                   /*The total size*/
	int i;                      /*The cyclic variable i*/
	int j;                      /*The cyclic variable j*/
	int k;                      /*The cyclic variable k*/
	int cache_size;             /*The size of cache*/
	//Init
	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);
	int LEVEL1_CACHE_SIZE = 32768;          // default 32768,32KB
	int LEVEL2_CACHE_SIZE = 262144;         // default 262144,256KB
	int LEVEL3_CACHE_SIZE = 8388608;        // default 8388608,8MB

	elapsed_time = -MPI_Wtime();
	if (argc == 2) {
		cache_size=LEVEL3_CACHE_SIZE;
	} else if (argc==3){
		cache_size=atoll(argv[2]);
	} else{
		if (!id) printf("Command line: %s <m>\n", argv[0]);
		MPI_Finalize();
		exit(1);
	}
	n = atoi(argv[1]);
```

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

n=n+1;
int number= (int)sqrt((double)n);
int local_size = number / 2;
labeled =(char*)malloc(local_size);
if (labeled == NULL)
{
        printf("Cannot allocate enough memory \n");
        MPI_Finalize();
        exit(1);
}
memset(labeled, 0, local_size);
prime = 3;//Search from prime 3
index = 0;
do{
        for (i = index + prime; i < local_size; i += prime) labeled[i] = 1;
        while (labeled[++index]);
        prime = 2 * index + 3;
} while (prime <= number);

//第一轮筛选
int count0=0;
for (i = 0; i < local_size; i++){
        if (!labeled[i]) count0++;
}
storage = (int*)malloc(sizeof(int) * count0);
if (storage == NULL){
        printf("Cannot allocate enough memory \n");
        MPI_Finalize();
        exit(1);
}
count0 = 0;
for (i = 0; i < local_size; i++){
        if (!labeled[i]) storage[count0++] = 2 * i + 3;
```

```
    }
    size = (n - 2) / 2;
    int num=size / cache_size;
    int low;
    int first;
    int res;
    int blo_size;
    count = 0;
    for (i = 0; i < num; i++){
        low = i * cache_size;
        low_value = low + (id * (cache_size)) / (p);
        high_value = low + (((id + 1) * (cache_size)) / (p)) - 1;
        blo_size = high_value - low_value + 1;
        low_value = 2 * low_value + 3;
        high_value = 2 * high_value + 3;
        marked = (char*)malloc(sizeof(char*) * blo_size);
        if (marked == NULL){
            printf("Cannot allocate enough memory\n");
            MPI_Finalize();
            exit(1);
        }
        memset(marked, 0, blo_size);
        for (j = 0; j < count0; j++){
            prime = storage[j];
            if (prime * prime > low_value) first = (prime * prime - low_value) / 2;
            else{
                res = low_value % prime;
                if (res == 0) first = 0;
                else{
                    if (res % 2 == 0) first = prime - res / 2;
                    else first = (prime - res) / 2;
                }
            }
            for (k = first; k < blo_size; k += prime) marked[k] = 1;
        }
        for (j = 0; j < blo_size; j++){
            if (!marked[j]){
                count++;
            }
        }
    }
    low = num * cache_size;
    int remain;
    remain = size - low;
```

```
low_value = low + (id * (remain)) / (p);
high_value = low + (((id + 1) * (remain)) / (p)) - 1;
blo_size = high_value - low_value + 1;
low_value = 2 * low_value + 3;
high_value = 2 * high_value + 3;
marked = (char*)malloc(sizeof(char) * blo_size);
if (marked == NULL){
      printf("Cannot allocate enough memory\n");
      MPI_Finalize();
      exit(1);
}
memset(marked, 0, blo_size);

for (i = 0; i < count0; i++){
      prime = storage[i];
      if (prime * prime > low_value) first = (prime * prime - low_value) / 2;
      else{
            res = low_value % prime;
            if (res == 0) first = 0;
            else{
                  if (res % 2 == 0) first = prime - res / 2;
                  else first = (prime - res) / 2;
            }
      }
      for (j = first; j < blo_size; j += prime) marked[j] = 1;
}
for (i = 0; i < blo_size; i++){
      if (!marked[i]){
            count++;
      }
}
MPI_Reduce(&count, &global_count, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);
//Stop timing
elapsed_time += MPI_Wtime();
n=n-1;
if (n>=2)
      global_count = global_count + 1;//Add the prime 2
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);
}
free(labeled);
free(storage);
```

```
    free(marked);
    MPI_Finalize();
    return 0;
}
```

调试说明：

1. 在 cygwin 的命令行中输入下面命令，可以获得 cache 的信息。如下图 9 所示。

```
dragon@DESKTOP-A7LJCSL ~
$ getconf -a | grep CACHE
LEVEL1_ICACHE_SIZE                 32768
LEVEL1_ICACHE_ASSOC                8
LEVEL1_ICACHE_LINESIZE             64
LEVEL1_DCACHE_SIZE                 32768
LEVEL1_DCACHE_ASSOC                8
LEVEL1_DCACHE_LINESIZE             64
LEVEL2_CACHE_SIZE                  262144
LEVEL2_CACHE_ASSOC                 4
LEVEL2_CACHE_LINESIZE              64
LEVEL3_CACHE_SIZE                  8388608
LEVEL3_CACHE_ASSOC                 16
LEVEL3_CACHE_LINESIZE              64
LEVEL4_CACHE_SIZE                  0
LEVEL4_CACHE_ASSOC                 0
LEVEL4_CACHE_LINESIZE              0
```

图 9

2. 为了程序更好的健壮性，采用两种方式，如果输入的参数不带有 cache 的大小，那么就会默认为我的三级 cache 大小，如果有 cache 的参数，就会把参数传入 cache 中。

程序运行结果见实验结果及数据分析。

八、实验数据及结果分析：

以下均为 100000000 内的素数，分别在 n=1,2,4,8,16 的运行结果。

8.1 基准程序：

编译：D:\cygwin64\bin\sh.exe -l -c "cd /cygdrive/e/CLion_projects/MPI_project; mpicc -std=c99 main.c -o main.exe;"

执行：D:\cygwin64\bin\sh.exe -l -c "cd /cygdrive/e/CLion_projects/MPI_project; mpirun -np 1 main 100000000;"

```
There are 5761455 primes less than or equal to 100000000
SIEVE (1)   3.001730
```

执行：D:\cygwin64\bin\sh.exe -l -c "cd /cygdrive/e/CLion_projects/MPI_project; mpirun -np 2 main 100000000;"

```
There are 5761455 primes less than or equal to 100000000
SIEVE (2)   2.016771
```

执行：D:\cygwin64\bin\sh.exe -l -c "cd /cygdrive/e/CLion_projects/MPI_project; mpirun -np 4 main 100000000;"

```
There are 5761455 primes less than or equal to 100000000
SIEVE (4)   1.437099
```

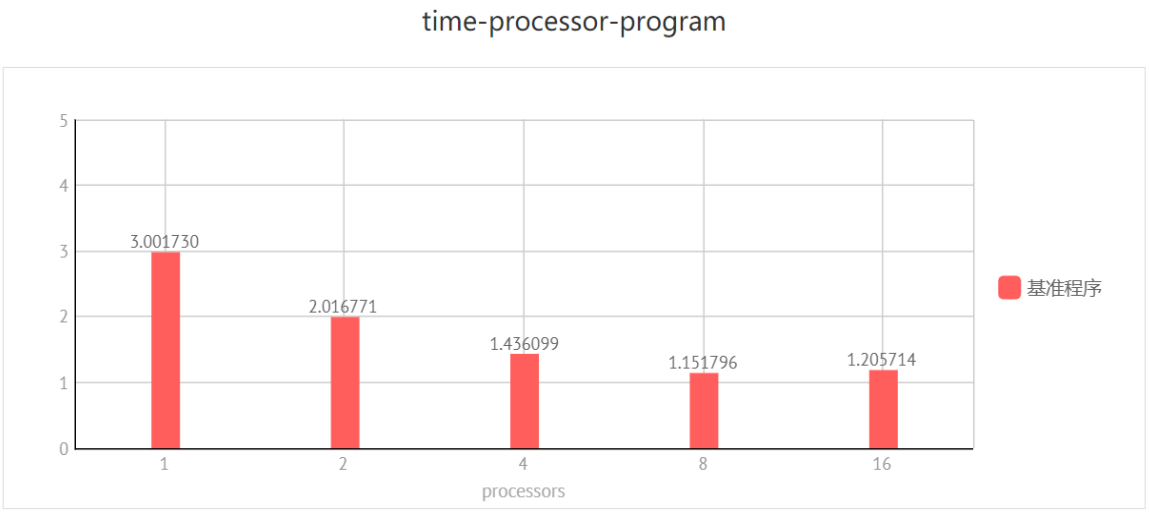
执行：D:\cygwin64\bin\sh.exe -l -c "cd /cygdrive/e/CLion_projects/MPI_project; mpirun -np 8 main 100000000;"

```
There are 5761455 primes less than or equal to 100000000
SIEVE (8)   1.151796
```

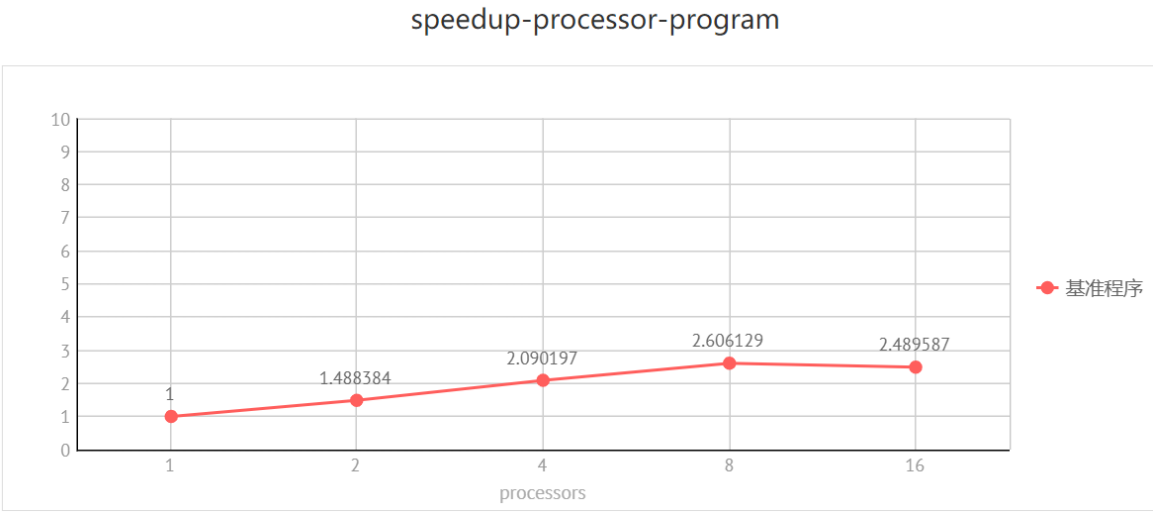
执行： D:\cygwin64\bin\sh.exe -l -c "cd /cygdrive/e/CLion_projects/MPI_project; mpirun -np 8 main 100000000;"

```
There are 5761455 primes less than or equal to 100000000
SIEVE (16)   1.205714
```

做出运行时间柱状图如图所示：



做出加速比曲线图如图所示：



结果分析：

1.随着 processors 的增加，运行时间在增加，加速比也在增加

2.但随着 processors 的过大，processor 间的通信也在增加，运行时间略有减少，加速比有所下降。

8.2 优化 1：去掉偶数

编译： D:\cygwin64\bin\sh.exe -l -c "cd /cygdrive/e/CLion_projects/MPI_project; mpicc -std=c99 removeEven.c -o removeEven.exe;"

执行： D:\cygwin64\bin\sh.exe -l -c "cd /cygdrive/e/CLion_projects/MPI_project; mpirun -np 1 removeEven 100000000;"

```
There are 5761455 primes less than or equal to 100000000
SIEVE (1)   1.455520
```

执行： D:\cygwin64\bin\sh.exe -l -c "cd /cygdrive/e/CLion_projects/MPI_project; mpirun -np 2 removeEven 100000000;"

```
There are 5761455 primes less than or equal to 100000000
SIEVE (2)   0.872227
```

执行： D:\cygwin64\bin\sh.exe -l -c "cd /cygdrive/e/CLion_projects/MPI_project; mpirun -np 4 removeEven 100000000;"

```
There are 5761455 primes less than or equal to 100000000
SIEVE (4)   0.621907
```

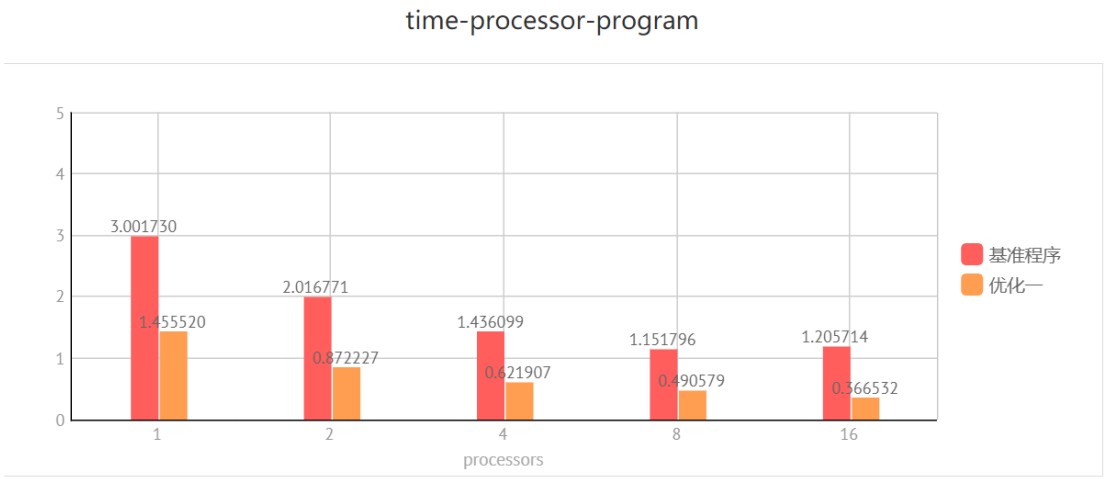
执行： D:\cygwin64\bin\sh.exe -l -c "cd /cygdrive/e/CLion_projects/MPI_project; mpirun -np 8 removeEven 100000000;"

```
There are 5761455 primes less than or equal to 100000000
SIEVE (8)   0.490579
```

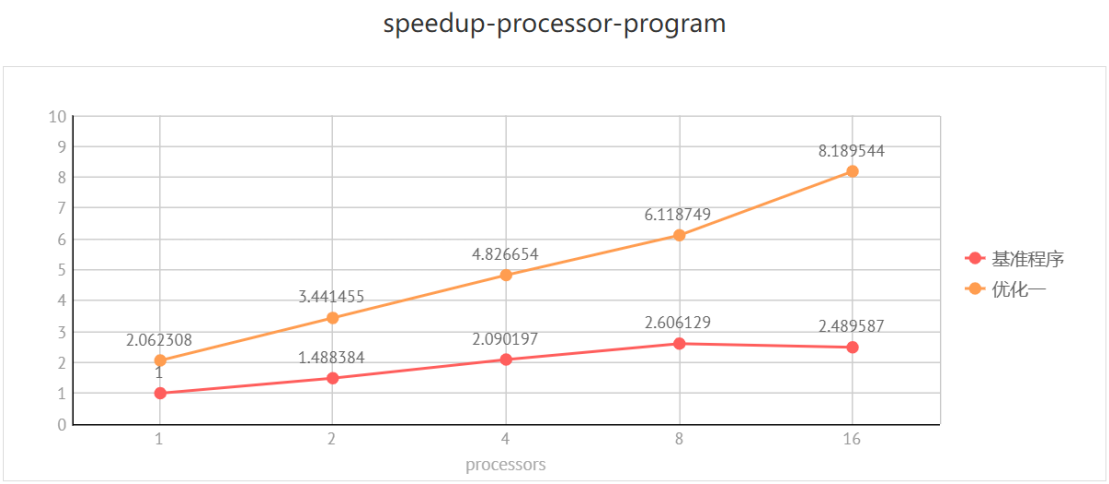
执行： D:\cygwin64\bin\sh.exe -l -c "cd /cygdrive/e/CLion_projects/MPI_project; mpirun -np 16 removeEven 100000000;"

```
There are 5761455 primes less than or equal to 100000000
SIEVE (16)   0.366532
```

做出运行时间柱状图如图所示：



做出加速比曲线图如图所示：



结果分析：

1.经过去掉偶数的优化后，运行时间在每一个 processors 都有很大的缩短，接近为 1/2。

2.随着 processors 的增加，整体的运行速度都在变快，加速比也在增大。

8.3 优化 2：消除广播

编译： D:\cygwin64\bin\sh.exe -l -c "cd /cygdrive/e/CLion_projects/MPI_project; mpicc -std=c99 eliminateBroadcast.c -o eliminateBroadcast.exe;"

执行： D:\cygwin64\bin\sh.exe -l -c "cd /cygdrive/e/CLion_projects/MPI_project; mpirun -np 1 eliminateBroadcast 100000000;"

```
There are 5761455 primes less than or equal to 100000000
SIEVE (1)   1.546989
```

执行： D:\cygwin64\bin\sh.exe -l -c "cd /cygdrive/e/CLion_projects/MPI_project; mpirun -np 2 eliminateBroadcast 100000000;"

```
There are 5761455 primes less than or equal to 100000000
SIEVE (2)   1.250760
```

执行： D:\cygwin64\bin\sh.exe -l -c "cd /cygdrive/e/CLion_projects/MPI_project; mpirun -np 4 eliminateBroadcast 100000000;"

```
There are 5761455 primes less than or equal to 100000000
SIEVE (4)   1.109031
```

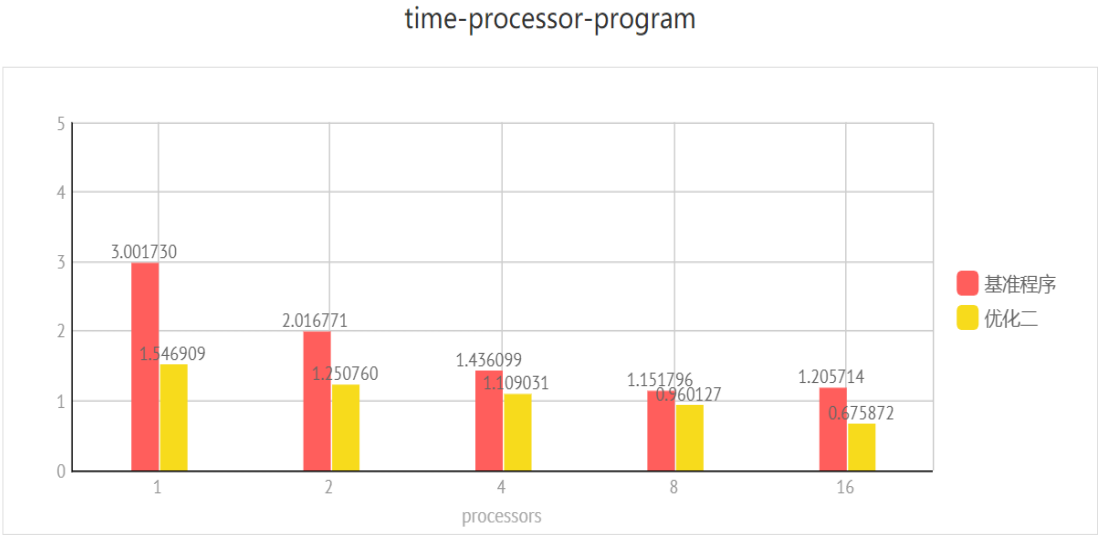
执行： D:\cygwin64\bin\sh.exe -l -c "cd /cygdrive/e/CLion_projects/MPI_project; mpirun -np 8 eliminateBroadcast 100000000;"

```
There are 5761455 primes less than or equal to 100000000
SIEVE (8)   0.960127
```

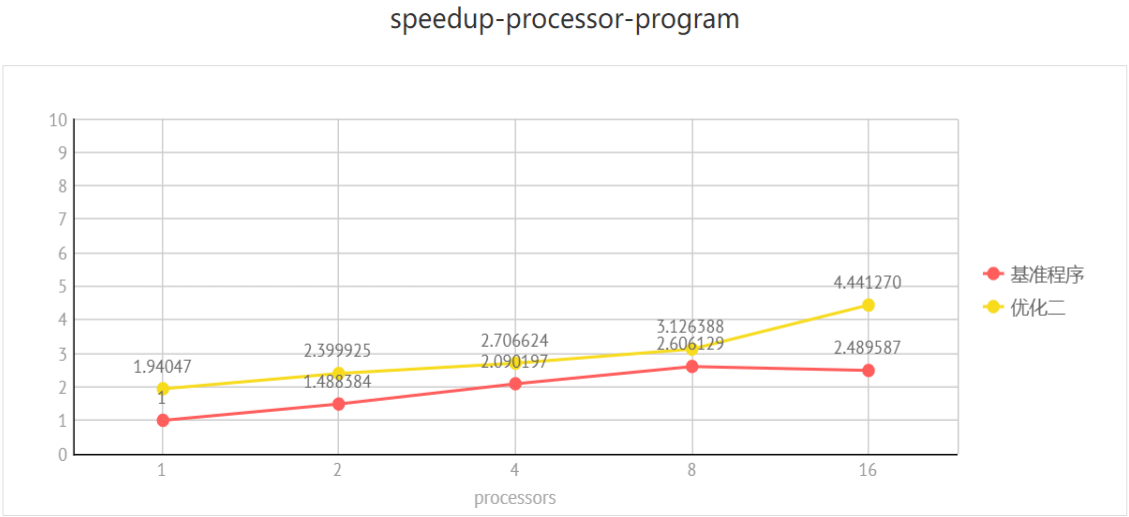
执行： D:\cygwin64\bin\sh.exe -l -c "cd /cygdrive/e/CLion_projects/MPI_project; mpirun -np 16 eliminateBroadcast 100000000;"

```
There are 5761455 primes less than or equal to 100000000
SIEVE (16)   0.675872
```

做出运行时间柱状图如图所示：



做出加速比曲线图如图所示：



结果分析：

1.消除广播之后，每个 processors 的运行时间都有所减少，处理能力增加。

2.随着 processors 的增加，运行时间减少，加速比增加。

8.4 优化 3：Cache 优化(此次优化在优化 1 和优化 2 的基础上完成)

编译： D:\cygwin64\bin\sh.exe -l -c "cd /cygdrive/e/CLion_projects/MPI_project; mpicc -std=c99 cacheOptimization.c -o cacheOptimization.exe;"

执行： D:\cygwin64\bin\sh.exe -l -c "cd /cygdrive/e/CLion_projects/MPI_project; mpirun -np 1 cacheOptimization 100000000;"

```
There are 5761455 primes less than or equal to 100000000
SIEVE (1)   0.441826
```

执行： D:\cygwin64\bin\sh.exe -l -c "cd /cygdrive/e/CLion_projects/MPI_project; mpirun -np 2

cacheOptimization 100000000;"

```
There are 5761455 primes less than or equal to 100000000
SIEVE (2)   0.221986
```

执行： D:\cygwin64\bin\sh.exe -l -c "cd /cygdrive/e/CLion_projects/MPI_project; mpirun -np 4 cacheOptimization 100000000;"

```
There are 5761455 primes less than or equal to 100000000
SIEVE (4)   0.133229
```

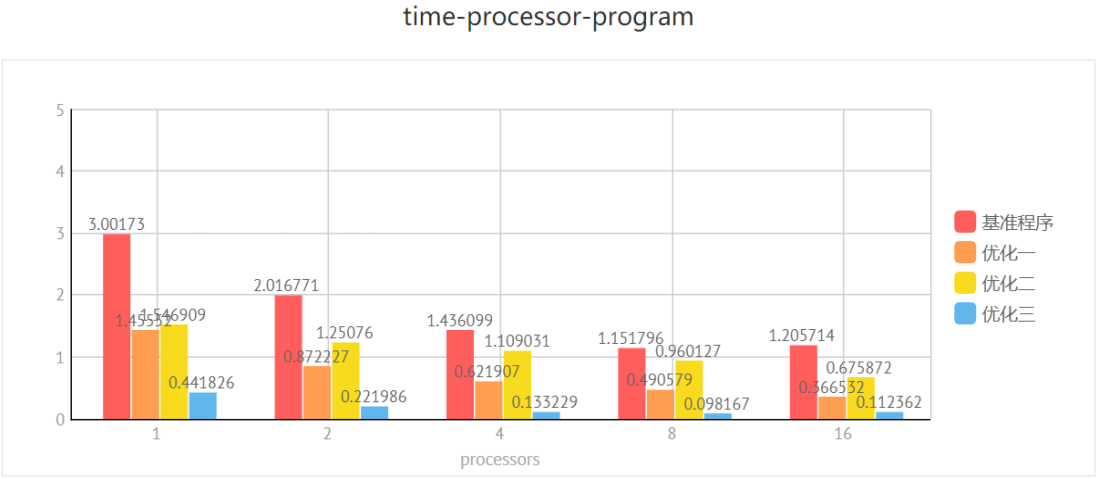
执行： D:\cygwin64\bin\sh.exe -l -c "cd /cygdrive/e/CLion_projects/MPI_project; mpirun -np 8 cacheOptimization 100000000;"

```
There are 5761455 primes less than or equal to 100000000
SIEVE (8)   0.098167
```

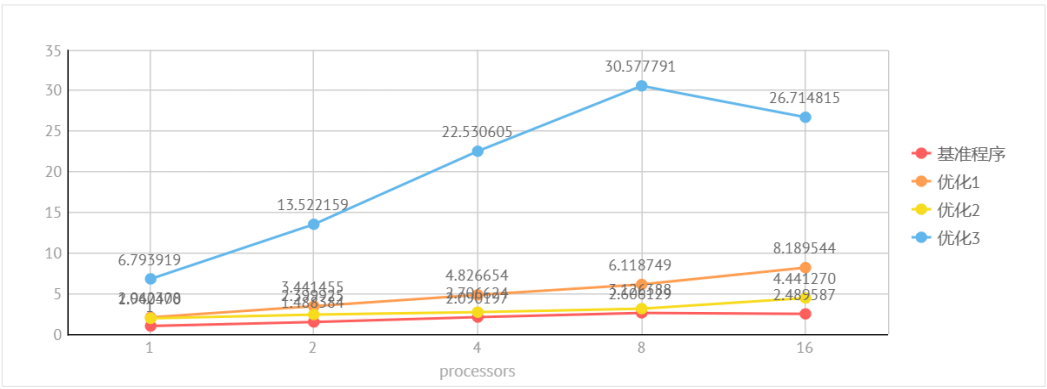
执行： D:\cygwin64\bin\sh.exe -l -c "cd /cygdrive/e/CLion_projects/MPI_project; mpirun -np 16 cacheOptimization 100000000;"

```
There are 5761455 primes less than or equal to 100000000
SIEVE (16)   0.112362
```

做出基准代码，优化 1，优化 2，优化 3（优化 1 和 2 基础）的运行时间柱状图如下图所示：



做出基准代码，优化 1，优化 2，优化 3（优化 1 和 2 基础）的加速比折线图如下图所示：



结果分析（总）：

1.去偶数优化其实只是在数量上线性的减少了一半问题规模，然而通信和广播模式没有改变，故在不同进程规模下所需的时间之比也是是线性的，并且接近 2 倍。

2. 因为进程数越大，通信时造成的广播代价也会增大，故在进程数多的时候，广播代价增大，去广播优化的效果也更加显著，加速比也更高。

3.Cache 优化是这三种优化里面最好的，在不同大小的进程下优化最好，耗时最短，加速比最大，所以能优化尽量进行 Cache 优化。

4. 在不同进程数下，广播代价、计算代价均不同这是导致加速比不同的一个重要的原因。

九、实验结论：

对原始代码通过多种优化最终使得代码运行时间得到了很大的降低，并且通过 cache 命中优化极大地提高了并行化的效率。

十、总结及心得体会：

1.复习了基本的 Linux 命令和学习了 cygwin 的安装及 CLion 的其他功能的使用。

2.学习了 MPI 编程实现埃拉托斯特尼筛法。

3.学习到了 MPI 调优的方法。

4.学习了如何在 Windows 的环境下使用 Clion 借助 cygwin 来进行 MPI 编程

5.体会到了分布式计算的强大之处。

6.锻炼了自己调试 bug 的能力

十一、对本实验过程及方法、手段的改进建议：

希望可以提供一些更多的参考资料，特别是 cache 有关的部分。

报告评分：

指导教师签字：

学生签字：