电 子 科 技 大 学

实 验 报 告

学生姓名：ALAMBO TAREKEGN KELTA	**学 号：2018080701016**
一、实验室名称：NO PHYSICAL LAB	
二、实验项目名称：Implementation of Eratosthenes sieve Method and Performance optimization based on MPI	

Sieve of Eratosthenes is a simple and ancient algorithm (over 2200 years old) used to find the prime numbers up to any given limit. It is one of the most efficient ways to find small prime numbers

For a given upper limit the algorithm works by iteratively marking the multiples of primes as composite, starting from **2**. Once all multiples of 2 have been marked composite, the multiples of next prime, i.e., 3 are marked composite. This process continues until p < sqrt(N) where p is a prime number.

This algorithm was devised by **Eratosthenes**, a Greek mathematician, astronomer, poet, geographer. Eratosthenes was born in 276 BC, so this method was over 2200 years old. Therefore, it is many times called the **ancient efficient method for primes** and it is still used heavily today.

The following are detailed steps used to find prime numbers equal or less than a given integer η.

a) List all consecutive numbers from 2 to n, i.e. (2, 3, 4, 5, ……, n).
b) Assign the first prime number letter p.
c) Beginning with p^2, perform an incremental of p and mark the integers equal or greater than p^2 in the algorithm. These integers will be $p\ (p + 1), p\ (p + 2), p\ (p + 3), p\ (p + 4)$ …
d) The first unmarked number greater than p is identified from the list. If the number does not exist in the list, the procedure is halted. p is equated to the number and step 3 is repeated.
e) The Sieve of Eratosthenes is stopped when the square of the number being tested exceeds the last number on the list.
f) All numbers in the list left unmarked when the algorithm ends are referred to as prime numbers.

The pseudo code of Eratosthenes sieve method is as follows:

```
find primes up to N
For all numbers a: from 2 to sqrt(n)
    IF a is unmarked THEN
        a is prime
        For all multiples of a (a < n)
            mark multiples of as composite
The unmarked numbers in the list are prime numbers.
```

Benchmark version of parallel code:

Define an array marked, the subscript of each element corresponds to an integer, and its value indicates whether the integer is a prime number, a value of 1 is a prime number, and a value of 0 is not a prime number.

- Assuming that all numbers are prime numbers, set the marked array to 0. Select the first integer 2, and mark the multiples of 2 from its corresponding array element 2*2=4, and mark it until the last number.
- Next, select the next unmarked number, which must be a prime number (for instance, 3), and notify each process in the form of broadcast to filter out multiples of this prime number. And the loop continues until all pr
- At the end of the loop, the sum of the unmarked numbers in all processes is all the prime numbers in 1-n.

四、实验目的：

1. Using MPI programming to implement parallel Sieve of Eratosthenes algorithm.

2. Perform performance analysis and tuning of the program.

五、实验内容：

Operating System: Linux

Programming Environment: Windows PC, XShell 7, UESTC Computer Cluster (VM)

1). Using the remote connection tool to log in to the cluster in SSH mode. Only campus network (UESTC-WIFI) can be used to connect to cluster computer!

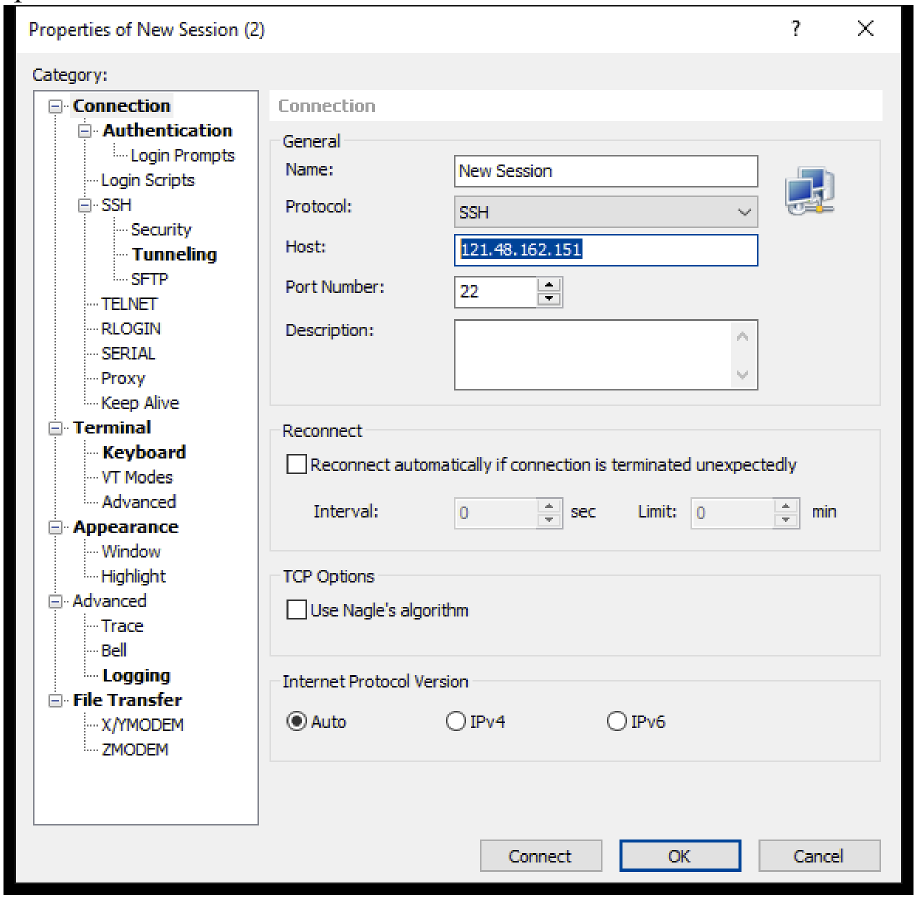
IP: 121.48.162.151

Port: 22

Each student's account name: a+student id

六、实验器材（环境配置）：

- ➢ First, go to the official XShell Free licensing website https://www.netsarang.com/en/free-for-home-school/ download and install XShell latest version software.
- ➢ After the installation is complete, open the XShell software select File>New the setup window appears
- ➢ Then Enter the session name and given IP address (121.48.162.151 as a Host name) and Select 22 as port number and click Connect as shown below:



Properties of New Session (2)

Category: Connection

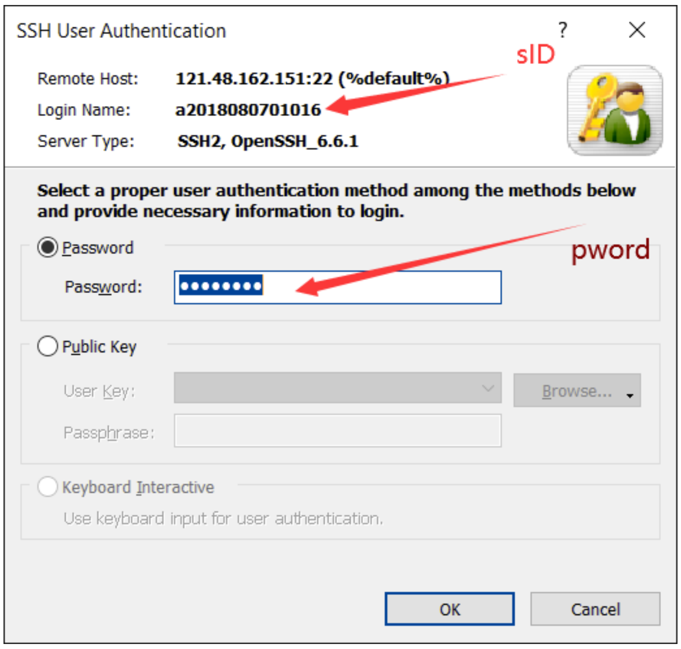
General
Name: New Session
Protocol: SSH
Host: 121.48.162.151
Port Number: 22
Description:

Reconnect
☐ Reconnect automatically if connection is terminated unexpectedly
Interval: 0 sec Limit: 0 min

TCP Options
☐ Use Nagle's algorithm

Internet Protocol Version
● Auto ○ IPv4 ○ IPv6

The connection is established using student ID and given computer cluster IP address then password.


SSH User Authentication

Remote Host: 121.48.162.151:22 (%default%)

Login Name: a2018080701016

Server Type: SSH2, OpenSSH_6.6.1

Select a proper user authentication method among the methods below and provide necessary information to login.

Enter cu01 or cu02 mode to start compiling and running your programs

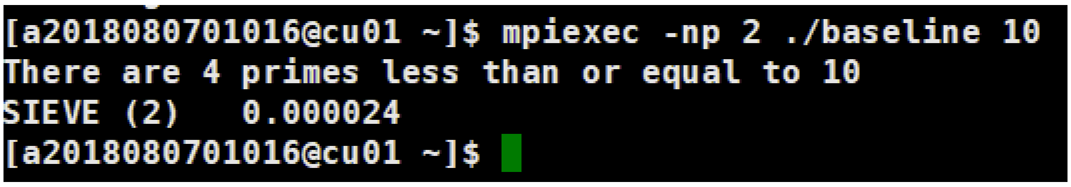
七、实验步骤及操作：

First, make sure you have prepared the required source code files and upload them to the server by using different methods (for instance, using xftp protocol of xShell software or you can type rz on ssh command). Name the benchmark code into baseline.cpp; name the optimization 1 code into optimizer1.cpp; name the optimization 2 code into optimizer2.cpp; name the optimization 3 code into optimizer3.cpp. Use XShell ssh command line parameters to compile and run the initial source code and optimization programs respectively. For instance, compile the benchmark source code using ***mpic++ -o baseline baseline.cpp*** to generate executable file. One can see the list of files by entering l*s* command.

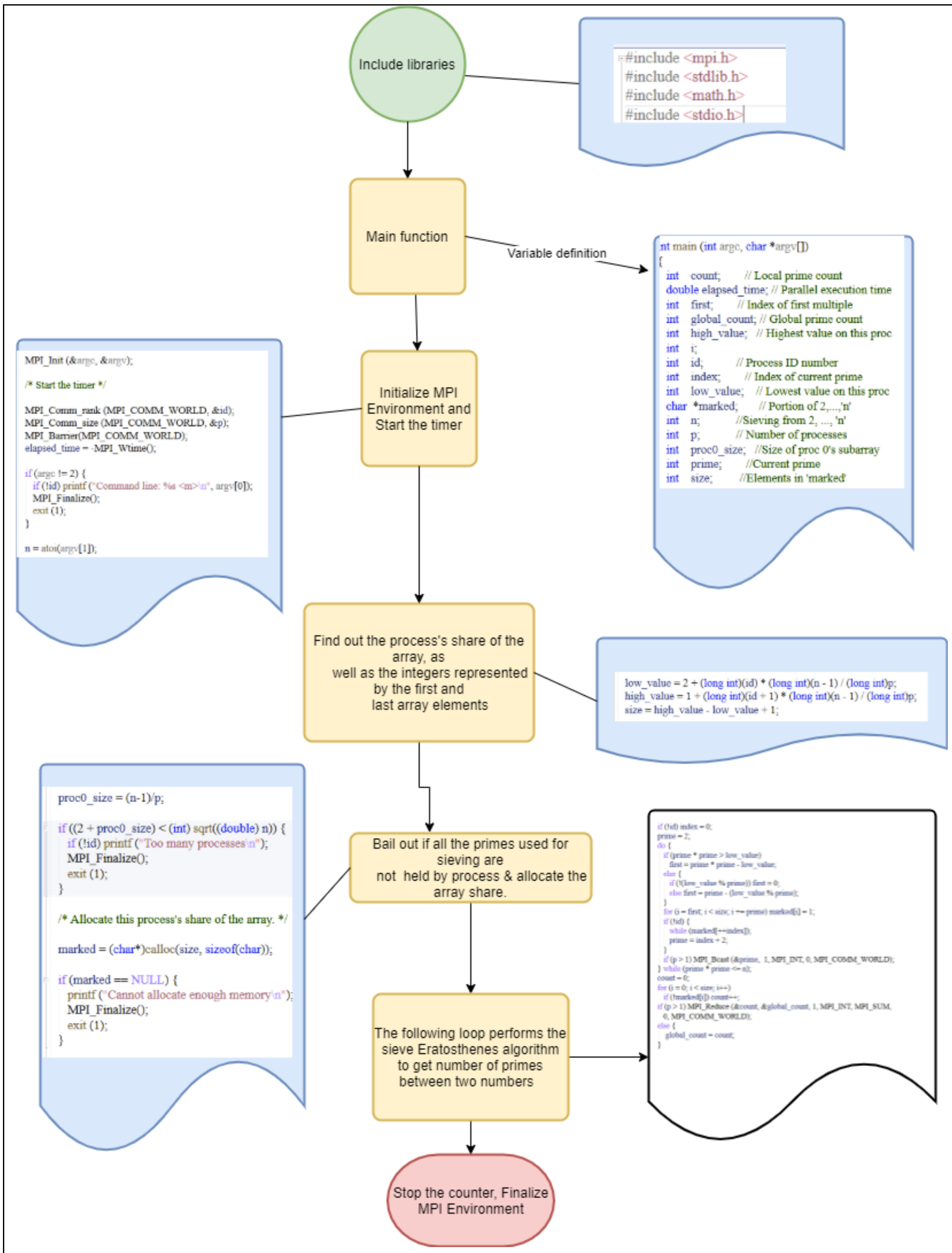
And use ***mpiexec -np 1 ./baseline 1000000*** to run and see the results. Repeat this procedure for the subsequent source code files such as *Optmizer1.cpp to Optmizer2*.cpp. and Add chunk value after data scale for Optmize3.cpp (e.g. ***mpiexec -np 1 ./baseline 1000000 100***.) After running the code one can record results using different techniques. For instance, taking screenshots, exporting runtime results into text document and analyzing them accordingly.

a) Benchmark Program Description

This baseline C++ parallel program performs sieve of Eratosthenes algorithm to get the number of even numbers between any two given numbers using MPI programming (for instance, the number of primes between 1 and 10 is 4)



```
[a2018080701016@cu01 ~]$ mpiexec -np 2 ./baseline 10
There are 4 primes less than or equal to 10
SIEVE (2)   0.000024
[a2018080701016@cu01 ~]$
```


Include libraries
Main function
Variable definition
Initialize MPI Environment and Start the timer
Find out the process's share of the array, as well as the integers represented by the first and last array elements
Bail out if all the primes used for sieving are not held by process & allocate the array share.
The following loop performs the sieve Eratosthenes algorithm to get number of primes between two numbers
Stop the counter, Finalize MPI Environment

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

```
nt 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]);
```

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

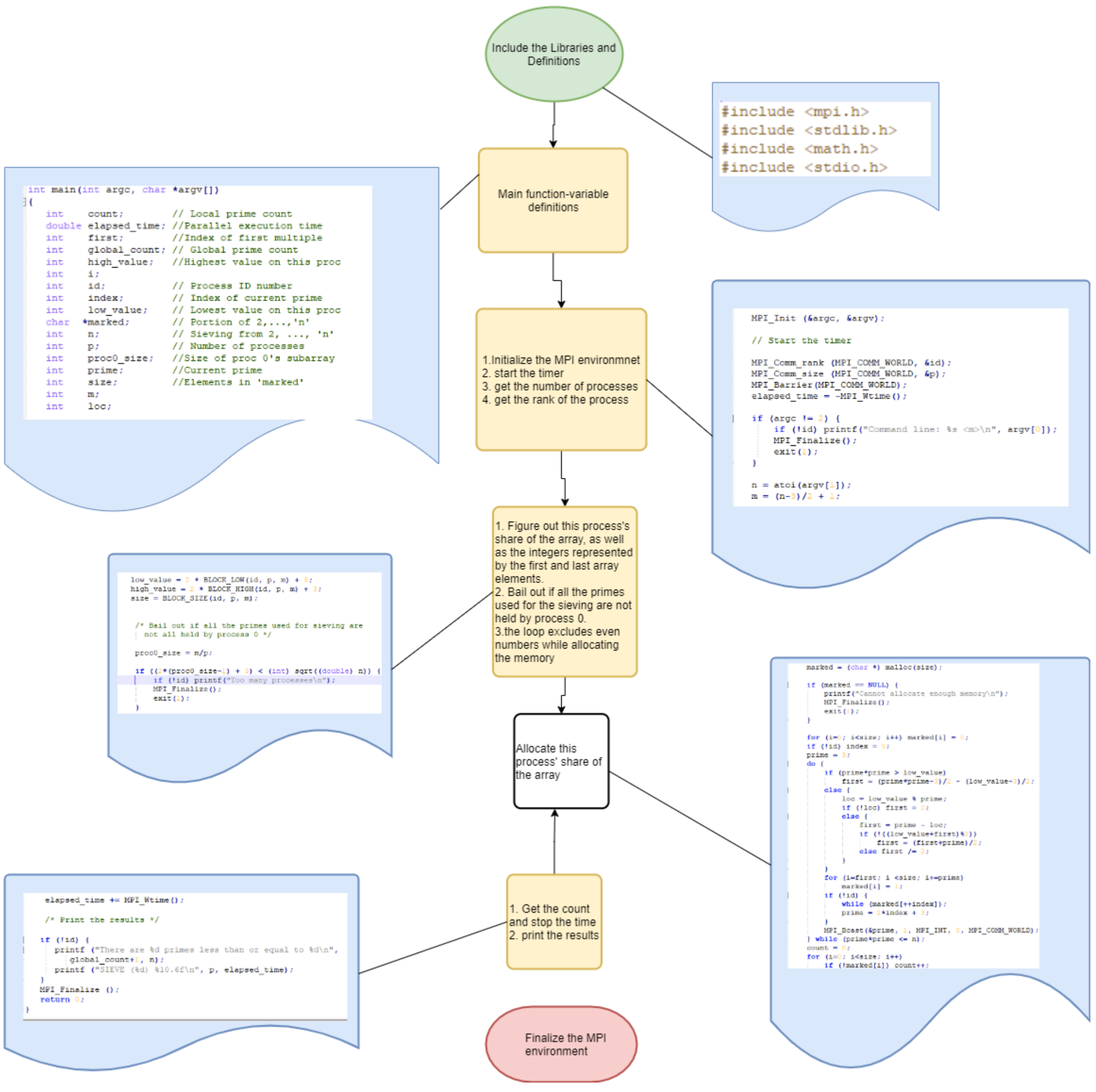
```
proc0_size = (n-1)/p;

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;
}
```

b. First Optimization Program Description

Modify the benchmark parallel Sieve of Eratosthenes program given above to incorporate the first improvement described as removing even numbers (i.e., the program should not set aside memory for even integers.) The reason that the algorithm works is the that:

Clearly it is sufficient to mark all multiples of 2, 3, ..., floor(sqrt(n)), as not prime, and then what remains is prime. This is what the Sieve of Eratosthenes essentially does, however it eliminates some redundancies by only marking off the multiples of each k starting at k^2. The reason that this shortcut works is because if a number is a multiple of k smaller than k^2, then it has a value given by k * m for some integer m with m < k, and we have already tested for multiples of m. the program accepts an argument n for the set {2, 3, ..., n} in which we search for prime numbers.


Include the Libraries and Definitions
Main function-variable definitions
1.Initialize the MPI environmnet
2. start the timer
3. get the number of processes
4. get the rank of the process
1. Figure out this process's share of the array, as well as the integers represented by the first and last array elements.
2. Bail out if all the primes used for the sieving are not held by process 0.
3.the loop excludes even numbers while allocating the memory
Allocate this process' share of the array
1. Get the count and stop the time
2. print the results
Finalize the MPI environment

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

```
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;
```

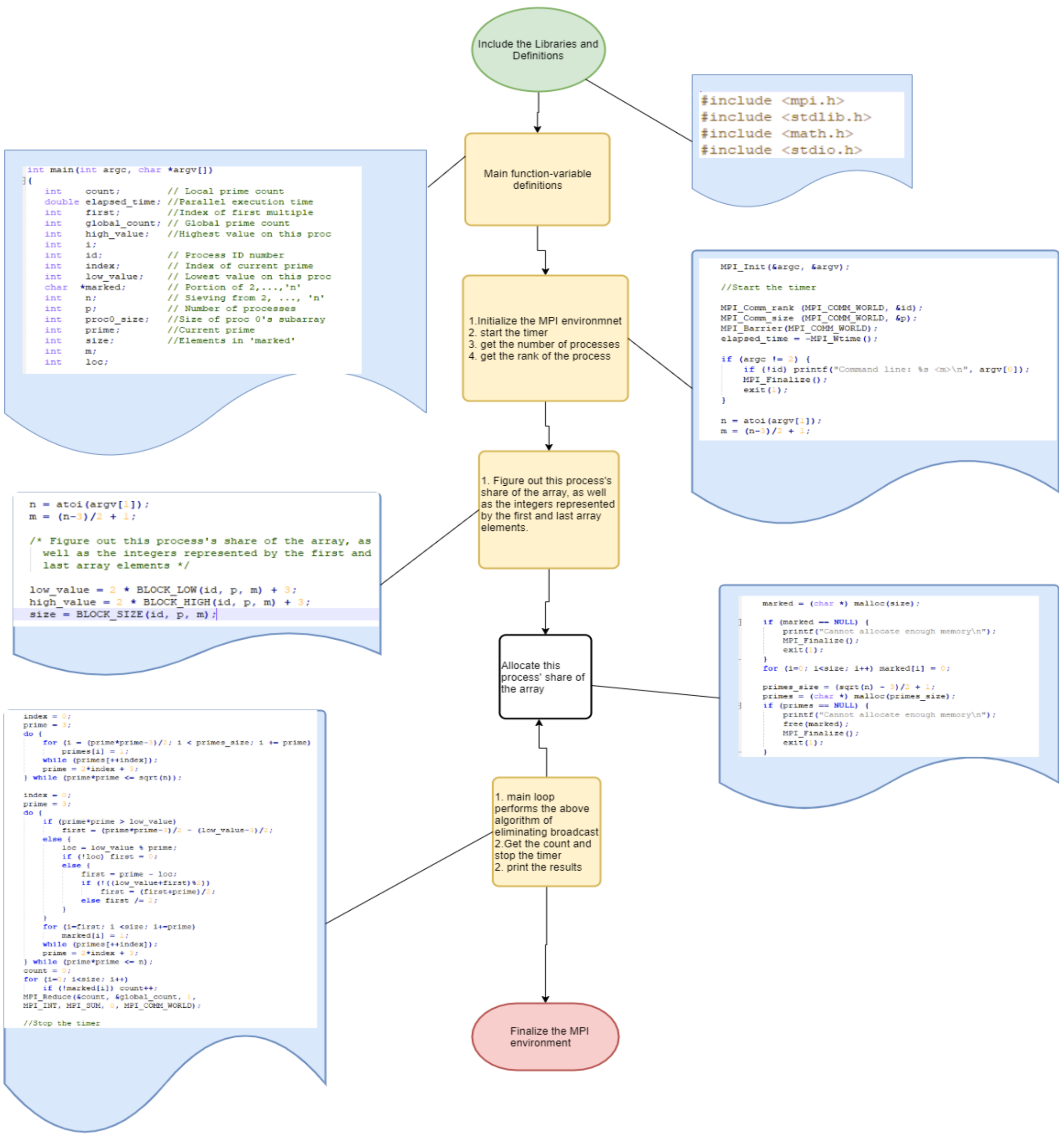
```
   low_value = 2 * BLOCK_LOW(id, p, m) + 3;
   high_value = 2 * BLOCK_HIGH(id, p, m) + 3;
   size = BLOCK_SIZE(id, p, m);


   /* 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);
   }
```

```
   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-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++;
```

```
    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;
}
```

c. Second Optimization Program Description

Modify the benchmark parallel Sieve of Eratosthenes program given above to incorporate the first two improvements such as removing even numbers (i.e., the program should not set aside memory for even integers) and eliminating the radio (i.e., each process should use the sequential Sieve of Eratosthenes algorithm on a separate array to find all primes between 3 and floor(sqrt(n)) rather than waiting for a broadcast from the 0-th process). With this information, the call to MPI_Bcast can be eliminated. The Program accepts an argument n for the set {2, 3, ..., n} in which we search for prime numbers.



Include the Libraries and Definitions
Main function-variable definitions
1.Initialize the MPI environmnet
2. start the timer
3. get the number of processes
4. get the rank of the process
1. Figure out this process's share of the array, as well as the integers represented by the first and last array elements.
Allocate this process' share of the array
1. main loop performs the above algorithm of eliminating broadcast
2.Get the count and stop the timer
2. print the results
Finalize the MPI environment

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

```c
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;
```

```c
    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;
```

```c
    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);
```

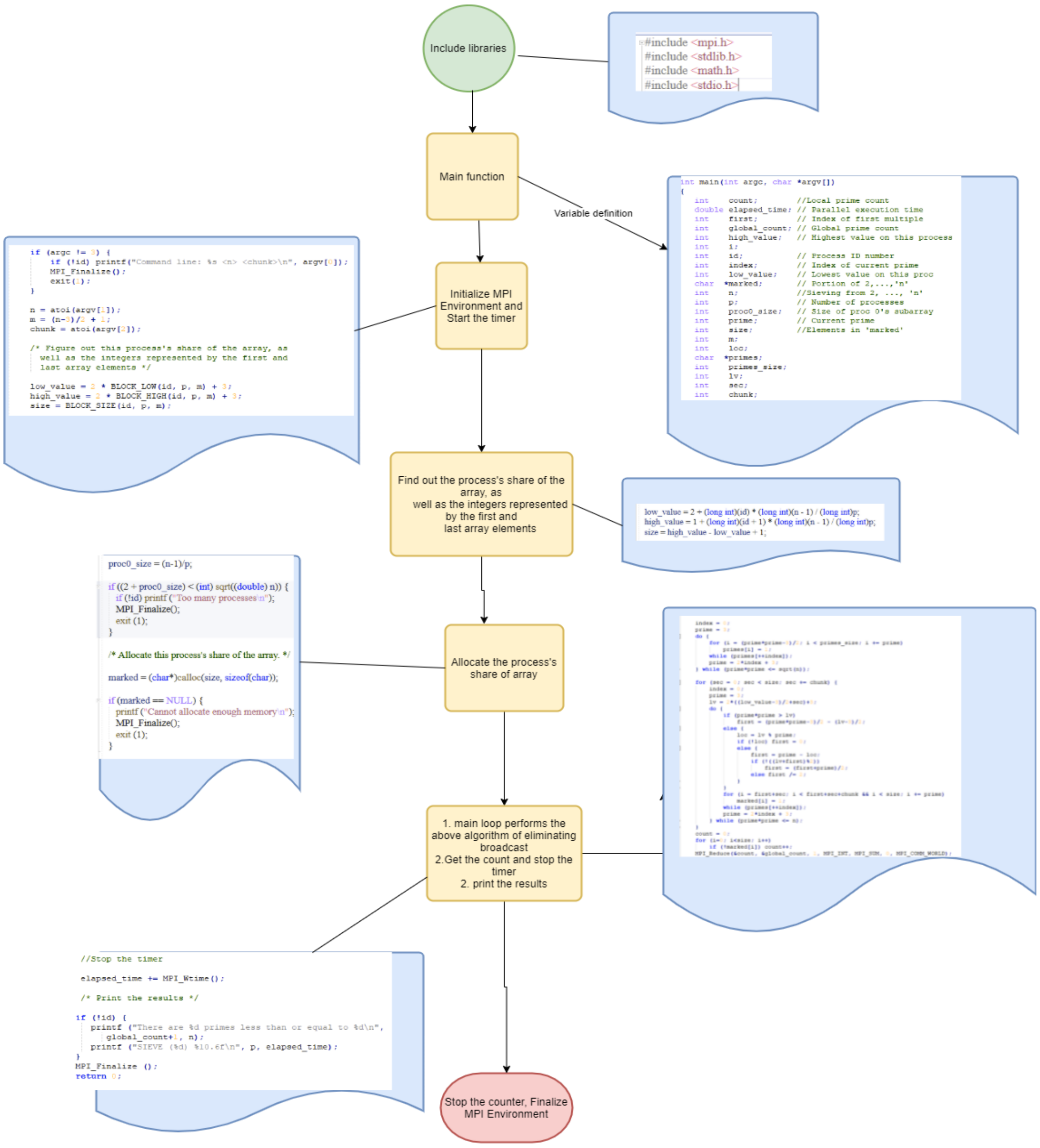
```c
    marked = (char *) malloc(size);

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

    primes_size = (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);
    }
```

```c
    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
```

d. Third Optimization Program Description

Modify the Sieve algorithm to improve the cache hit rate by decomposing the section of numbers each process is responsible for into further sub-blocks.

We regard the core content of the algorithm as two loops. The outer loop is iterated between prime numbers from 3 to √n, and the inner loop is iterated between integers from 3 to n belonging to the process. If we swap the inner and outer layers of the loop, the cache hit rate of the program can be improved. We can put a part of the array into the Cache, mark all multiples of prime numbers less than √n, and then read the next part of the array.


Include libraries
Main function
Variable definition
Initialize MPI Environment and Start the timer
Find out the process's share of the array, as well as the integers represented by the first and last array elements
Allocate the process's share of array
1. main loop performs the above algorithm of eliminating broadcast
2.Get the count and stop the timer
2. print the results
Stop the counter, Finalize MPI Environment

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

```
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 process
   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;
```

```
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);
```

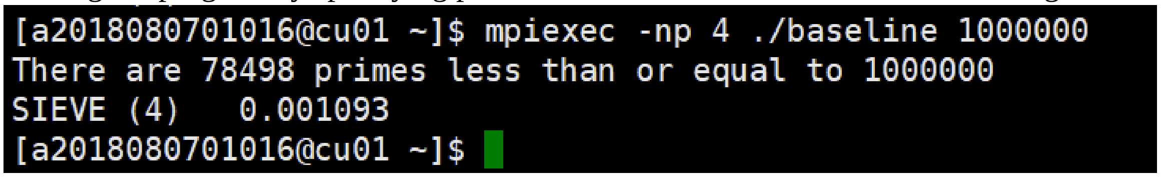
```
low_value = 2 + (long int)(id) * (long int)(n - 1) / (long int)p;
high_value = 1 + (long int)(id + 1) * (long int)(n - 1) / (long int)p;
size = high_value - low_value + 1;
```

```
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);
}
```

```
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;
```

Call mpiexec file (e.g. /baseline) directly and specify the number of processes and the filter range to run the program: the command "*mpiexec – np 4./baseline* 10^6 " can specify to start n processes and filter the prime number in the positive integer range less than or equal to number range(10^6). After running the program by specifying parameters, and the results are shown in the figure below

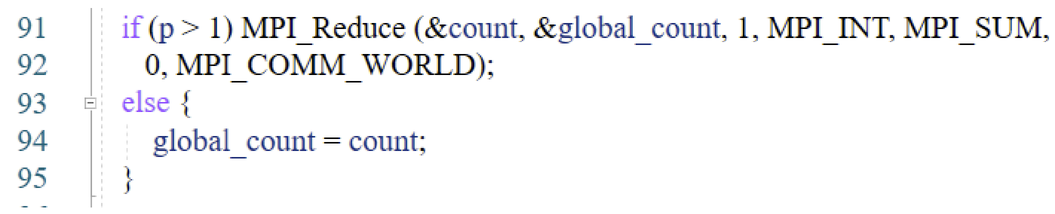


```
[a2018080701016@cu01 ~]$ mpiexec -np 4 ./baseline 1000000
There are 78498 primes less than or equal to 1000000
SIEVE (4)   0.001093
[a2018080701016@cu01 ~]$
```

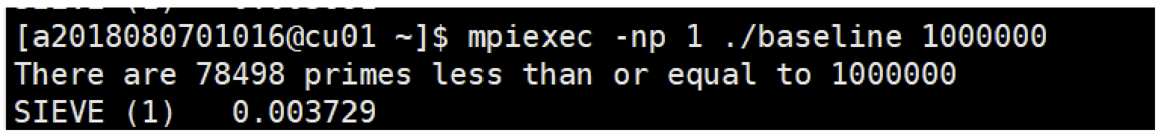
- **Debugging Benchmark Program**

- Before modifying line 91-94 in baseline code, the benchmark results were not as expected. After modification it displays expected output.

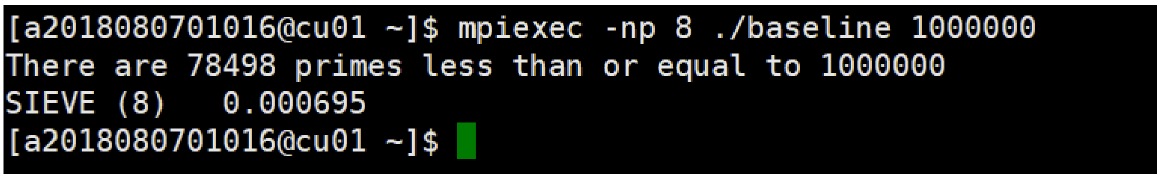
  On the baseline.cpp source code, notice that only "P > 1" is judged and processed at lines 91 and 92 of the source code, as shown in the following figure, line 93-95 are modifications.



```
91  if (p > 1) MPI_Reduce (&count, &global_count, 1, MPI_INT, MPI_SUM,
92      0, MPI_COMM_WORLD);
93  else {
94      global_count = count;
95  }
```



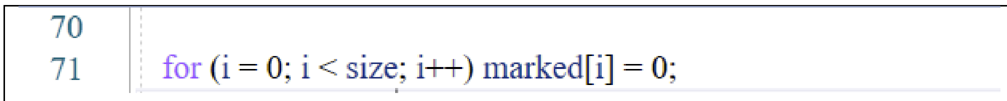
```
[a2018080701016@cu01 ~]$ mpiexec -np 1 ./baseline 1000000
There are 78498 primes less than or equal to 1000000
SIEVE (1)   0.003729
```



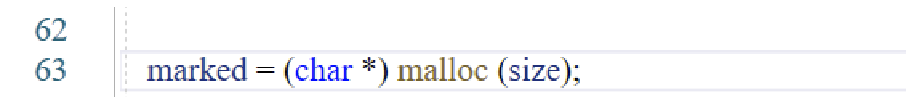
```
[a2018080701016@cu01 ~]$ mpiexec -np 8 ./baseline 1000000
There are 78498 primes less than or equal to 1000000
SIEVE (8)   0.000695
[a2018080701016@cu01 ~]$
```

- Solving the problem of wrong statistics of prime numbers:

In the source code baseline.cpp. note that when counting the number of primes, you make statistics by judging whether the value of each element in the array marked [] is 0. It is speculated from experience that for this similar situation where the number of statistics differs by one, the reason is often the initialization of the array or the subscript of the array, and errors will occur under certain conditions or boundary values. Further observation shows that the memory space allocated by the marked [] array is located in line 63 of the source code. Then, in line 71, the initial value 0 is given to the elements in the array, as shown in the following figure:

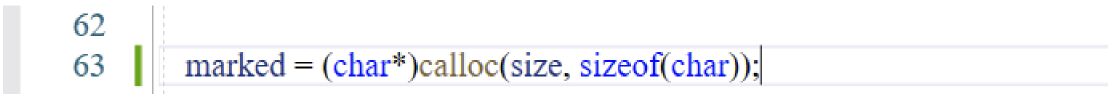


```
70
71    for (i = 0; i < size; i++) marked[i] = 0;
```

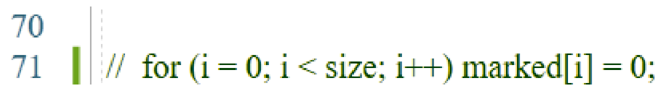


```
62
63    marked = (char *) malloc (size);
```

Modify the program after further debugging: comment out or directly delete the assignment operation on line 71, and modify the code on line 63 as follows:

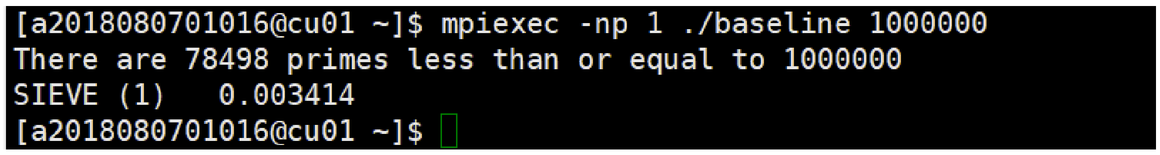


```
62
63    marked = (char*)calloc(size, sizeof(char));
```

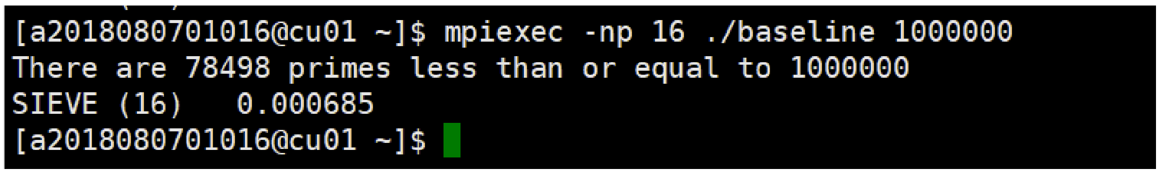


```
70
71    // for (i = 0; i < size; i++) marked[i] = 0;
```

Save and exit after modification, recompile and run the program according to the above command, and the program runs correctly, as shown in the figure:

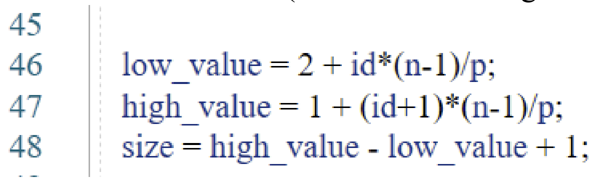


```
[a2018080701016@cu01 ~]$ mpiexec -np 1 ./baseline 1000000
There are 78498 primes less than or equal to 1000000
SIEVE (1)   0.003414
[a2018080701016@cu01 ~]$
```

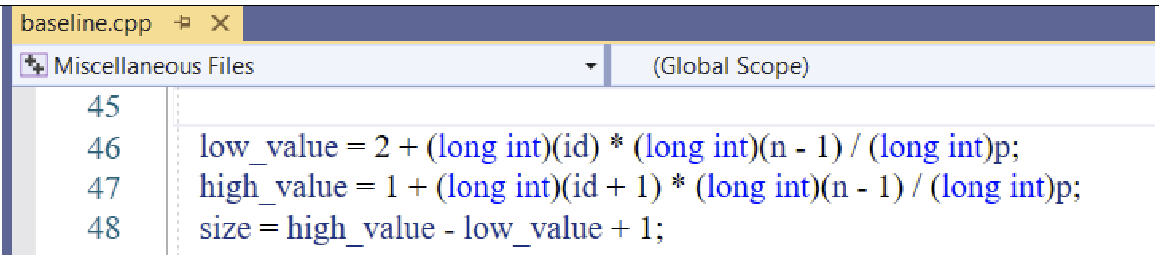


```
[a2018080701016@cu01 ~]$ mpiexec -np 16 ./baseline 1000000
There are 78498 primes less than or equal to 1000000
SIEVE (16)   0.000685
[a2018080701016@cu01 ~]$
```

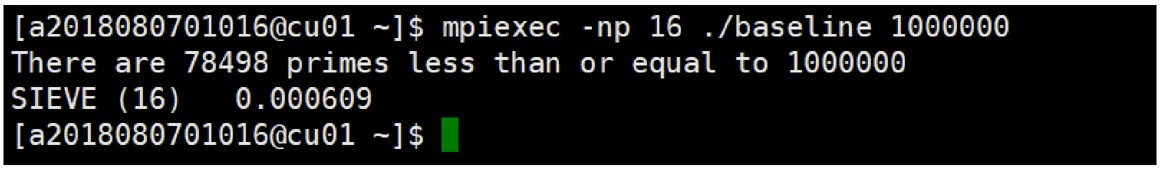
➢ The program has been able to run successfully and get correct results, but we can further expand the program so that the program can handle a wider range of data: modify the "low_value" and "high_value" variables in the source code to "long int" variables, and modify the code from lines 46 to 48 in the source code (as shown in the figure below):



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



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



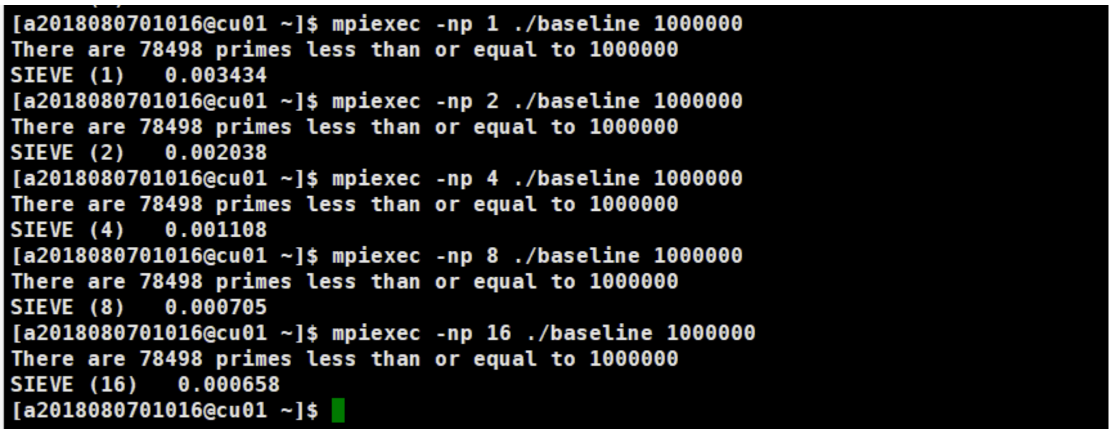
```
[a2018080701016@cu01 ~]$ mpiexec -np 16 ./baseline 1000000
There are 78498 primes less than or equal to 1000000
SIEVE (16)   0.000609
[a2018080701016@cu01 ~]$
```

➢ The performance of benchmark program under different data scales and different process configurations is tested. In this test, the number of test processes is 1, 2, 4, 8 and 16 respectively, and the selected data scale is 1,000,000. In order to avoid contingency, each group of data is tested 5 times in the same test environment, and the average value is taken as the final result. The test results are listed as follows:

Test group	np	Data Scale	Average time (s)
1	1	1000,000	0.003388
2	2	1000,000	0.001954
3	4	1000,000	0.001046
4	8	1000,000	0.000678
5	16	1000,000	0.000562

Table 1. baseline.cpp mpi sieve of Erathosthenes test results

Please see Text Documents Attached with Submission file for Average results I put in this tables. The screenshots are given to demonstrate the running results.



```
[a2018080701016@cu01 ~]$ mpiexec -np 1 ./baseline 1000000
There are 78498 primes less than or equal to 1000000
SIEVE (1)   0.003434
[a2018080701016@cu01 ~]$ mpiexec -np 2 ./baseline 1000000
There are 78498 primes less than or equal to 1000000
SIEVE (2)   0.002038
[a2018080701016@cu01 ~]$ mpiexec -np 4 ./baseline 1000000
There are 78498 primes less than or equal to 1000000
SIEVE (4)   0.001108
[a2018080701016@cu01 ~]$ mpiexec -np 8 ./baseline 1000000
There are 78498 primes less than or equal to 1000000
SIEVE (8)   0.000705
[a2018080701016@cu01 ~]$ mpiexec -np 16 ./baseline 1000000
There are 78498 primes less than or equal to 1000000
SIEVE (16)   0.000658
[a2018080701016@cu01 ~]$
```

➢ Next, based on the baseline program, the performance of the algorithm is optimized.

Optimization 1: Removing even numbers:

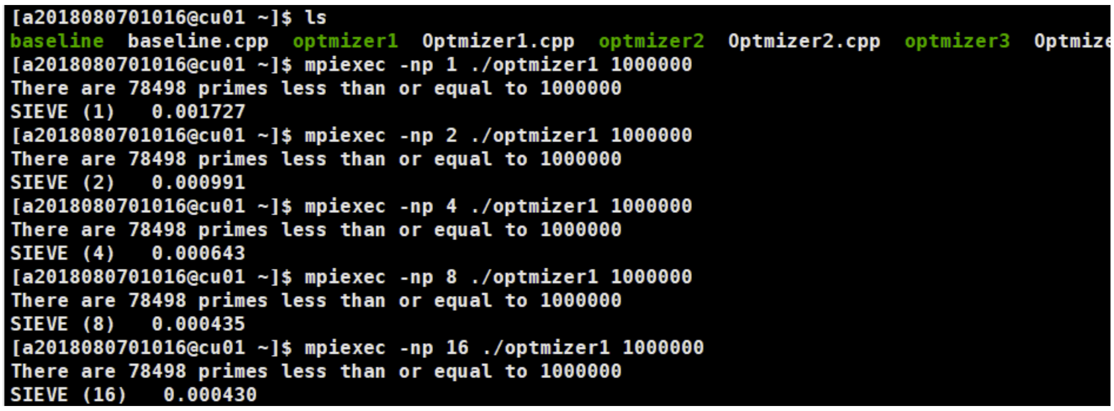
As we know that all even numbers except 2 are not prime numbers, we can halve the total number to be screened, so as to improve the screening efficiency.

Based on the above ideas, modify the source code **baseline.cpp** (Note: since there are many code changes, they are not listed here one by one, and the optimized complete code will be attached with final submission file). Note that the code after removing the even number to be screened after the first optimization is **Optmizer1.cpp**.

Then, the performance of optimization 1 program under given data sizes and different process configurations is tested. In this test, the number of test processes is 1, 2, 4, 8 and 16 respectively, and the selected data scale is 1,000,000. In other words, a total of 5 * 1 = 5 groups of data need to be tested. In order to avoid contingency, each group of data is tested 5 times in the same test environment, and the average value is taken as the final result. The test results are listed as follows

Test group	np	Data Scale	Average time (s)
1	1	1000,000	0.001585
2	2	1000,000	0.000915
3	4	1000,000	0.000569
4	8	1000,000	0.000414
5	16	1000,000	0.000396

Table 2 Optmizer1.cpp mpi sieve Eratosthenes test results


```
[a2018080701016@cu01 ~]$ ls
baseline  baseline.cpp  optmizer1  Optmizer1.cpp  optmizer2  Optmizer2.cpp  optmizer3  Optmize
[a2018080701016@cu01 ~]$ mpiexec -np 1 ./optmizer1 1000000
There are 78498 primes less than or equal to 1000000
SIEVE (1)   0.001727
[a2018080701016@cu01 ~]$ mpiexec -np 2 ./optmizer1 1000000
There are 78498 primes less than or equal to 1000000
SIEVE (2)   0.000991
[a2018080701016@cu01 ~]$ mpiexec -np 4 ./optmizer1 1000000
There are 78498 primes less than or equal to 1000000
SIEVE (4)   0.000643
[a2018080701016@cu01 ~]$ mpiexec -np 8 ./optmizer1 1000000
There are 78498 primes less than or equal to 1000000
SIEVE (8)   0.000435
[a2018080701016@cu01 ~]$ mpiexec -np 16 ./optmizer1 1000000
There are 78498 primes less than or equal to 1000000
SIEVE (16)   0.000430
```

➢ **Optimization 2: Eliminate the Radio:**

The initial code broadcasts the prime of the next filter multiple through process 0. MPI is required between processes_ BCAST function to communicate. Communication is bound to have overhead, so we let each process find the prime numbers in their first sqrt (n) numbers, and filter the remaining prime numbers through these prime numbers. In this way, there is no need to broadcast prime numbers in each cycle between processes, and the performance is improved.

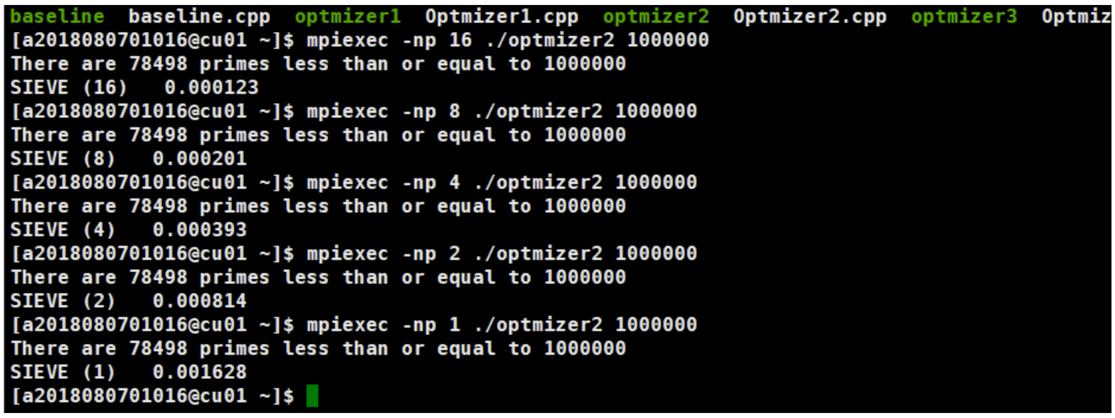
It is assumed that in addition to N / P integers, each task also has a separate array containing integers 3,5, 7,.. >=n. Before finding the prime number of 3 ~ n, each task can first calculate the prime number of 3 ~ n by serial algorithm. Once this step is completed, each task has all the arrays containing all the primes of 3 ~ √ n. In these tasks, the entire array can be filtered without the broadcast step.

Based on the above ideas, modify the code Optmizer1.cpp after the first optimization. It is noted that after the second optimization, the code after further removing the broadcast communication is Optmizer2.cpp.

Next, the performance of modification2 program under a given data size and different process configurations is tested. In this test, the number of test processes is 1, 2, 4, 8 and 16 respectively, and the selected data scale is 1000,000. The test results are listed as follows:

Test group	np	Data Scale	Average time (s)
1	1	1000,000	0.001579
2	2	1000,000	0.000744
3	4	1000,000	0.000369
4	8	1000,000	0.000200
5	16	1000,000	0.000110

Table 3 mpi Optmizer2.cpp sieve Eratosthenes test results- Eliminate the radio



```
baseline  baseline.cpp  optmizer1  Optmizer1.cpp  optmizer2  Optmizer2.cpp  optmizer3  Optmiz
[a2018080701016@cu01 ~]$ mpiexec -np 16 ./optmizer2 1000000
There are 78498 primes less than or equal to 1000000
SIEVE (16)   0.000123
[a2018080701016@cu01 ~]$ mpiexec -np 8 ./optmizer2 1000000
There are 78498 primes less than or equal to 1000000
SIEVE (8)   0.000201
[a2018080701016@cu01 ~]$ mpiexec -np 4 ./optmizer2 1000000
There are 78498 primes less than or equal to 1000000
SIEVE (4)   0.000393
[a2018080701016@cu01 ~]$ mpiexec -np 2 ./optmizer2 1000000
There are 78498 primes less than or equal to 1000000
SIEVE (2)   0.000814
[a2018080701016@cu01 ~]$ mpiexec -np 1 ./optmizer2 1000000
There are 78498 primes less than or equal to 1000000
SIEVE (1)   0.001628
[a2018080701016@cu01 ~]$
```

- **Optimization 3: Cache Optimization**

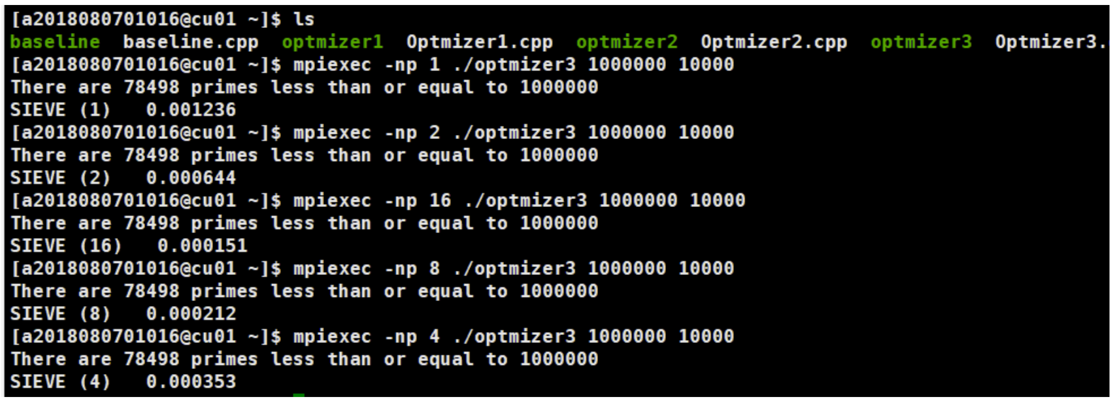
Each process further divides the data to be filtered into blocks according to the size of the machine's Cache Block, and uses prime numbers in 3-sqrt(n) to mark and filter in each block, thereby increasing the cache hit rate.

Most of the execution time of the parallel sieve algorithm is spent on marking the scattered elements of a very huge array, resulting in a very poor cache hit rate. I explained more about this section in program description above.

The results are as follows:

Test group	np	Data Scale	Average time (s)
1	1	1000,000	0.001096
2	2	1000,000	0.000581
3	4	1000,000	0.000312
4	8	1000,000	0.000183
5	16	1000,000	0.000151

Table 3 mpi Optmizer3.cpp sieve Eratosthenes test results-Cache Optimizer



```
[a2018080701016@cu01 ~]$ ls
baseline  baseline.cpp  optmizer1  Optmizer1.cpp  optmizer2  Optmizer2.cpp  optmizer3  Optmizer3.
[a2018080701016@cu01 ~]$ mpiexec -np 1 ./optmizer3 1000000 10000
There are 78498 primes less than or equal to 1000000
SIEVE (1)   0.001236
[a2018080701016@cu01 ~]$ mpiexec -np 2 ./optmizer3 1000000 10000
There are 78498 primes less than or equal to 1000000
SIEVE (2)   0.000644
[a2018080701016@cu01 ~]$ mpiexec -np 16 ./optmizer3 1000000 10000
There are 78498 primes less than or equal to 1000000
SIEVE (16)   0.000151
[a2018080701016@cu01 ~]$ mpiexec -np 8 ./optmizer3 1000000 10000
There are 78498 primes less than or equal to 1000000
SIEVE (8)   0.000212
[a2018080701016@cu01 ~]$ mpiexec -np 4 ./optmizer3 1000000 10000
There are 78498 primes less than or equal to 1000000
SIEVE (4)   0.000353
```

九、实验结论：

For different specific tasks, the problem should be analyzed first, and efficient algorithms should be designed and adopted, which is conducive to the improvement of program performance. A good algorithm improves the performance of the program very obviously, and can achieve the effect of getting twice the result with half the effort.

It is not that the more processes the parallel computing is better, the multi-process should not be used blindly. For tasks with a small scale of calculation data, multiple processes will cause a waste of system resources, and also affect program running time and efficiency, which may result in half the effort.

In different situations and different environments, the performance of the same program may have obvious differences. The task data scale, machine software and hardware, current state of the system, performance and resources should be comprehensively considered to determine how to apply parallel programs. Experience and experimentation are required.

十、总结及心得体会：

The simultaneous growth in availability of **big data** and in the number of simultaneous users on the Internet places particular pressure on the need to carry out computing tasks in parallel, or simultaneously. Distributed and Parallel computing allows computers to handle very large-scale problems, and it is widely used today. Through this experiment, combined with the knowledge learned in the course, I came into contact with, understood, and learned distributed parallel computing programs for the first time, and also carried out the writing, modification, debugging and application of parallel computing programs, which broadened the scope of knowledge and exercised Improved programming ability and hands-on ability. Through practice, I have exercised my ability to solve problems, analyze data, and draw conclusions, and I have benefited a lot.

十一、*对本实验过程及方法、手段的改进建议：

None

报告评分：

指导教师签字：