Lab 3 Report

Andres Calderon

acald013@ucr.edu

November 7, 2016

1. Part 1

Implementation of Part 1 can be seen in section A.1 of the appendix.

2. Part 2

Implementation of Part 2 can be seen in section A.2 of the appendix.

3. Part 3

Implementation of Part 3 can be seen in section A.3 of the appendix.

4. Analysis

4.1. Block size estimation

Before to run the benchmark we have to set the values for the block sizes for the prime and odd arrays for Part 3. A combination of different values were deployed to estimate those values empirically. Figure 1 shows the execution time with values for the odd block size ranging from 256 to 1024 and for the prime block size ranging from 32 to 1024.

For this experiment 8 nodes and 32 processors per node were used ($n = 10^{10}$). Red cell background in figure 1 illustrates worst performance and green color highlights the best ones. This shows that the best combination is 512 and 256 for the odd and prime arrays respectively. These values were used in the implementation of the third optimization (see lines 99 and 100 in appendix A.3).

4.2. Experiments

Two sets of experiments were run to measure the performance of the optimizations. The first one run five iterations of the job presented in appendix B.1 ranging the number of processors from 4 to 30 on steps of 2. The average of the five runs are shown in table 1 and figure 2. This experiment run over 10 nodes using 3 processors for each node.

We can see than the best response is achieved by optimization 3 as expected. It has to be noted that there is no much difference between optimization 1 and 2. This can be explained by the high speed networking infrastructure of Tardis. In this case, the communication cost does not have much impact in the time execution.

The performance gain between the baseline and the optimizations shows an increase of 5x. Similarly, an increase in the number of cores will give a performance increase of 8x. Overall, between a baseline algorithm running over a reduced number of cores and the implementation of the 3 optimizations using more number of cores the performance increase is the 40x.

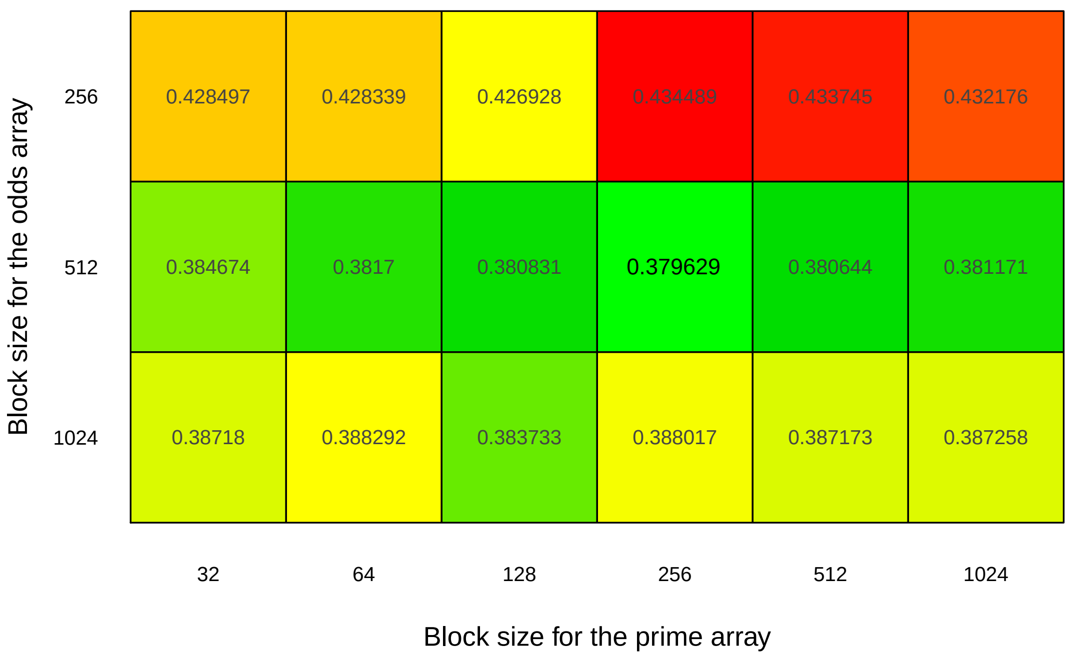

Figure 1: Execution time in seconds of different cache sizes for prime and odds arrays.

Number of Cores	Optimization				
	Baseline	Remove evens	Remove Bcast	Reorder loops	
4	92.40	44.75	44.61	18.92	
6	63.15	30.96	30.31	12.45	
8	47.10	23.17	22.41	9.11	
10	39.85	18.95	18.11	7.04	
12	33.27	16.00	15.22	5.99	
14	27.97	13.64	12.99	5.11	
16	24.91	12.11	11.43	4.51	
18	22.07	10.71	10.13	4.01	8x
20	19.52	9.56	9.10	3.54	
22	17.59	8.67	8.28	3.24	
24	16.17	7.92	7.67	2.85	
26	14.96	7.34	7.05	2.75	
28	13.85	6.79	6.55	2.54	
30	13.12	6.40	6.20	2.39	
			5x		

Table 1: Comparing the four versions of the Sieve of Erastosthenes.

The second experiment follows the requirements of the project. It uses 32, 64, 128 and 256 processors running under 1, 2, 4 and 8 nodes respectively. Similarly, five runs were deployed running the script shown in section B.2 and the averages were taken. Table 2 and figure 3 illustrate the results.

We can see a similar behavior that the previous experiment. However, the difference between optimization 1 and 2 almost disappear when a high number of processors are used. In addition, table 2 shows that the performance increase between baseline and the three implementations growth to 17x. Overall, the total performance gain using this configuration is almost 120x.

Number of Cores	Optimization				
	Baseline	Remove evens	Remove Bcast	Reorder loops	
32	28.18	13.82	13.74	2.72	
64	14.28	6.99	6.77	1.36	
128	13.03	6.32	6.31	0.73	7x
256	6.55	3.17	3.11	0.37	
			17x		

Table 2: Comparing the four versions of the Sieve of Erastosthenes.

Examples of the execution of the implementations and their outputs can be seen in sections B and C of the appendix. Additional material, figures and datasets can be found at `https://github.com/aocalderon/PhD/tree/master/Y2Q1/HPC/Project3`.

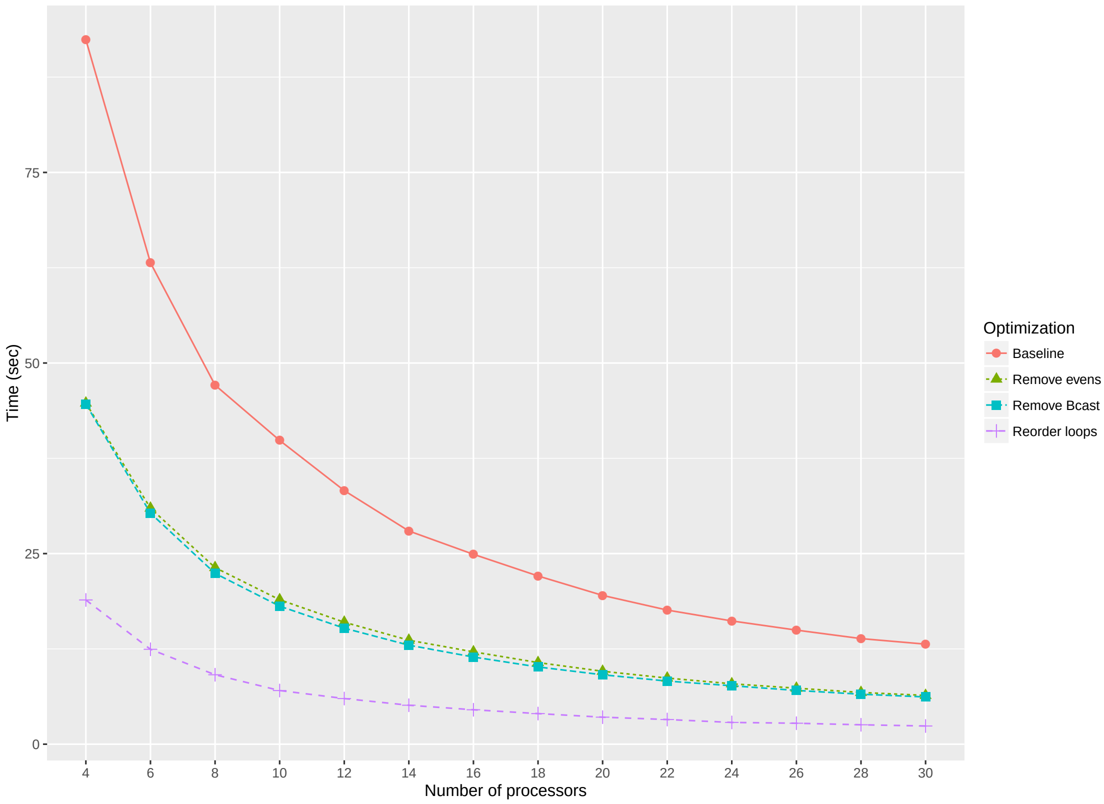

Figure 2: Execution time of the Sieve of Erastosthenes and its optimizations.

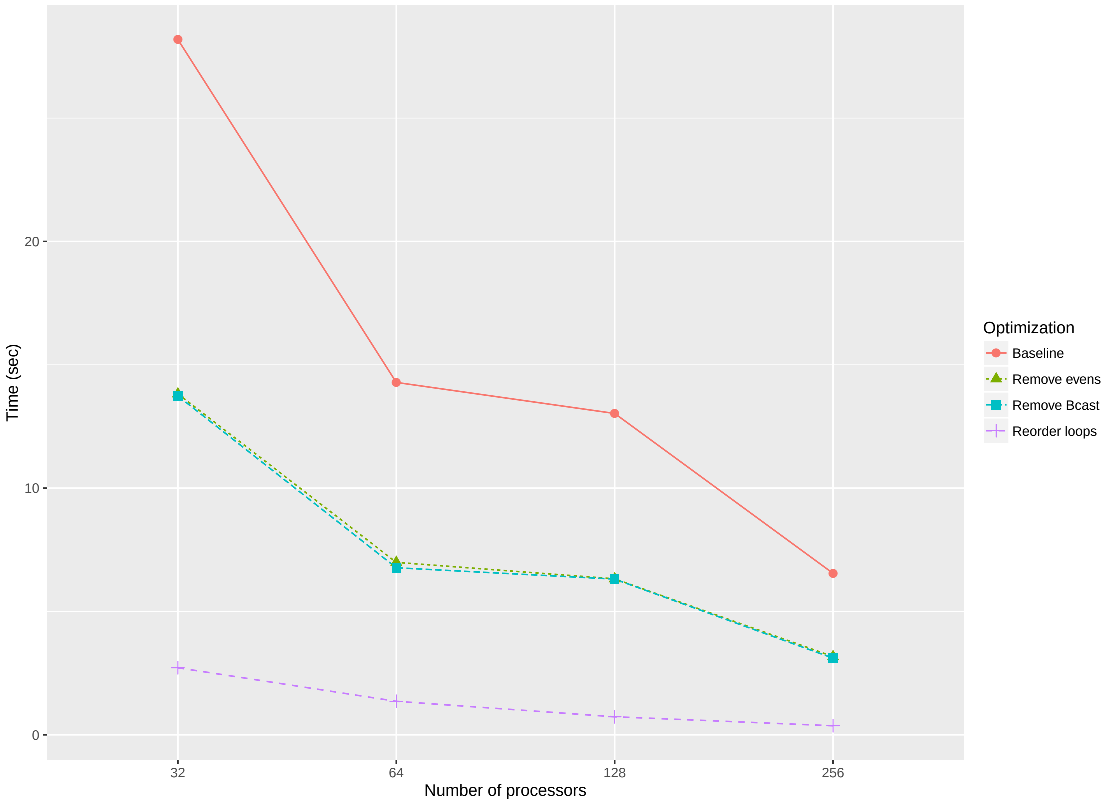

Figure 3: Execution time of the Sieve of Erastosthenes and its optimizations.

A. Source code

A.1. sieve1.c (Removing evens...)

```c
/*
 * Sieve of Eratosthenes
 *
 * Programmed by Michael J. Quinn
 *
 * Last modification: 7 September 2001
 */
#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[]) {
  unsigned int count; /* Local prime count */
  double elapsed_time; /* Parallel execution time */
  unsigned long  first; /* Index of first multiple */
  unsigned int global_count; /* Global prime count */
  unsigned long long high_value; /* Highest value on this proc */
  unsigned long long i;
  int id; /* Process ID number */
  unsigned long index; /* Index of current prime */
  unsigned long long low_value; /* Lowest value on this proc */
  char *marked; /* Portion of 2,...,'n' */
  unsigned long long int  n; /* Sieving from 2, ..., 'n' */
  int p; /* Number of processes */
  unsigned long proc0_size; /* Size of proc 0's subarray */
  unsigned long prime; /* Current prime */
  unsigned long long size; /* Elements in 'marked' */
  unsigned long long n_size; /* Number of odds between 3 to 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);
  elapsed_time = -MPI_Wtime();
  if (argc != 2) {
    if (!id) printf ("Command line: %s <m>\n", argv[0]);
    MPI_Finalize();
    exit (1);
  }
  /* Read N as a unsigned long long from the arguments */
  char *e;
  n = strtoull(argv[1], &e, 10);

  /* Compute number of odds between 3 to n */
  n_size = (n + 1) / 2;

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

  /* Adjust the formula to remove the even positions */
  low_value  = 2 + (2 * (id * (n_size - 1) / p)) + 1;
  high_value = 2 + (2 * (((id + 1) * (n_size - 1) / p) - 1)) + 1;
  /* Now we only need half of the size */
  size = (high_value - low_value + 1) / 2;
  /* Bail out if all the primes used for sieving are not all held by process 0 */
  proc0_size = (n_size - 1) / p;
  if ((2 + (2 * proc0_size) + 1) < (int) sqrt((double) n)) {
    if (!id) printf ("Too many processes\n");
    MPI_Finalize();
    exit (1);
  }
  /* Allocate this process's share of the array. */
```

```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;
  if (!id) index = 0;

  /* Start from 3 */
  prime = 3;
  do {
    /* Divide by 2 to manage only odd positions */
    if (prime * prime > low_value)
      first = (prime * prime - low_value) / 2;
    else {
      if (!(low_value % prime)) first = 0;
      else{
        first = prime - (low_value % prime);
        if(!(first % 2)) first = first / 2;
        else first = (first + prime) / 2;
      }
    }
    for (i = first; i <= size; i += prime){
      marked[i] = 1;
    }
    if (!id) {
      while (marked[++index]);
      /* Pick up the next odd prime */
      prime = 2 + (2 * index + 1);
    }
    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);
  /* Stop the timer */
  elapsed_time += MPI_Wtime();
  /* Print the results */
  if (!id) {
    global_count++;  //Counting 2 as prime...
    printf("S1, %llu, %d, %d, %10.6f\n", n, p, global_count, elapsed_time);
  }
  MPI_Finalize ();
  return 0;
}
```

A.2. sieve2.c (Removing Bcast...)

```c
/*
 * Sieve of Eratosthenes
 *
 * Programmed by Michael J. Quinn
 *
 * Last modification: 7 September 2001
 */
#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[]) {
  unsigned int count; /* Local prime count */
  double elapsed_time; /* Parallel execution time */
  unsigned long  first; /* Index of first multiple */
```

```c
unsigned int global_count; /* Global prime count */
unsigned long long high_value; /* Highest value on this proc */
unsigned long long i;
int id; /* Process ID number */
unsigned long index; /* Index of current prime */
unsigned long long low_value; /* Lowest value on this proc */
char *marked; /* Portion of 2,...,'n' */
unsigned long long int  n; /* Sieving from 2, ..., 'n' */
int p; /* Number of processes */
unsigned long proc0_size; /* Size of proc 0's subarray */
unsigned long prime; /* Current prime */
unsigned long kprime; /* Prime in marked0 */
unsigned long long size; /* Elements in 'marked' */
unsigned long long n_size; /* Number of odds between 3 to n */
unsigned long sqrtn; /* Square root of n */
char *marked0; /* Primes in between 3 to 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);
elapsed_time = -MPI_Wtime();
if (argc != 2) {
  if (!id) printf ("Command line: %s <m>\n", argv[0]);
  MPI_Finalize();
  exit (1);
}
/* Read N as a unsigned long long from the arguments */
char *e;
n = strtoull(argv[1], &e, 10);

/* Compute number of odds between 3 to n */
n_size = (n + 1) / 2;

/* Finding how many primes from 3 to sqrt(n) and allocating space*/
sqrtn = ceil(sqrt((double) n))/2;
marked0 = (char *) malloc(sqrtn + 1);
for(i = 0; i <=sqrtn; i++) marked0[i] = 0;
index = 0;

/* Finding the primes and store them in marked0 */
kprime = 3;
do{
  first = (kprime * kprime - 3) / 2;
  for(i = first; i <= sqrtn; i += kprime) marked0[i] = 1;
  while(marked0[++index]);
  kprime = 2 + (2 * index + 1);
} while(kprime * kprime <= sqrtn);

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

  /* Adjust the formula to remove the even positions */
low_value  = 2 + (2 * (id * (n_size - 1) / p)) + 1;
high_value = 2 + (2 * (((id + 1) * (n_size - 1) / p) - 1)) + 1;
size = (high_value - low_value + 1) / 2;

/* Bail out if all the primes used for sieving are
  not all held by process 0 */
proc0_size = (n_size - 1) / p;
if ((2 + (2 * proc0_size) + 1) < (int) sqrt((double) n)) {
  if (!id) printf ("Too many processes\n");
  MPI_Finalize();
  exit (1);
}
/* Allocate this process's share of the array. */
marked = (char *) malloc (size);
if (marked == NULL) {
```

```
    printf ("Cannot allocate enough memory\n");
    MPI_Finalize();
    exit (1);
  }
  for (i = 0; i <= size; i++) marked[i] = 0;
  index = 0;
  /* Start from 3 */
  prime = 3;
  do {
    /* Divide by 2 to manage only odd positions */
    if (prime * prime > low_value)
      first = (prime * prime - low_value) / 2;
    else {
      if (!(low_value % prime)) first = 0;
      else{
        first = prime - (low_value % prime);
        if(!(first % 2)) first = first / 2;
        else first = (first + prime) / 2;
      }
    }
    for (i = first; i <= size; i += prime){
      marked[i] = 1;
    }
    /* Pick up the next prime from marked0 */
    while(marked0[++index]);
    prime = 2 + (2 * index + 1);
    /* We do not need the Bcast anymore */
  } 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);
  /* Stop the timer */
  elapsed_time += MPI_Wtime();
  /* Print the results */
  if (!id) {
    global_count++;  //Counting 2 as prime...
    printf("S2, %llu, %d, %d, %10.6f\n", n, p, global_count, elapsed_time);
  }
  MPI_Finalize ();
  return 0;
}
```

A.3. sieve3.c (Reorder loops...)

```
/*
 * Sieve of Eratosthenes
 *
 * Programmed by Michael J. Quinn
 *
 * Last modification: 7 September 2001
 */
#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[]) {
  unsigned int count; /* Local prime count */
  double elapsed_time; /* Parallel execution time */
  unsigned long first; /* Index of first multiple */
  unsigned long position; /* Position of the prime in the block */
  unsigned int global_count; /* Global prime count */
  unsigned long long high_value; /* Highest value on this proc */
  unsigned long long i;
  int id; /* Process ID number */
```

```c
unsigned long index; /* Index of current prime */
unsigned long long low_value; /* Lowest value on this proc */
char *marked; /* Portion of 2,...,'n' */
unsigned long long int  n; /* Sieving from 2, ..., 'n' */
int p; /* Number of processes */
unsigned long proc0_size; /* Size of proc 0's subarray */
unsigned long prime; /* Current prime */
unsigned long kprime; /* Prime in marked0 */
unsigned long long size; /* Elements in 'marked' */
unsigned long long n_size; /* Number of odds between 3 to n */
unsigned long sqrtn; /* Square root of n */
char *marked0; /* Primes in between 3 to 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);
elapsed_time = -MPI_Wtime();
if (argc != 2) {
  if (!id) printf ("Command line: %s <m>\n", argv[0]);
  MPI_Finalize();
  exit (1);
}
/* Read N as a unsigned long long from the arguments */
char *e;
n = strtoull(argv[1], &e, 10);

/* Compute number of odds between 3 to n */
n_size = (n + 1) / 2;

/* Finding how many primes from 3 to sqrt(n) and allocating space*/
sqrtn = ceil(sqrt((double) n))/2;
marked0 = (char *) malloc(sqrtn + 1);
for(i = 0; i <=sqrtn; i++) marked0[i] = 0;
index = 0;

/* Finding the primes and store them in marked0 */
kprime = 3;
do{
  first = (kprime * kprime - 3) / 2;
  for(i = first; i <= sqrtn; i += kprime) marked0[i] = 1;
  while(marked0[++index]);
  kprime = 2 + (2 * index + 1);
} while(kprime * kprime <= sqrtn);

/* 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 + (2 * (id * (n_size - 1) / p)) + 1;
high_value = 2 + (2 * (((id + 1) * (n_size - 1) / p) - 1)) + 1;
size = (high_value - low_value + 1) / 2;
/* Bail out if all the primes used for sieving are
  not all held by process 0 */
proc0_size = (n_size - 1) / p;
if ((2 + (2 * proc0_size) + 1) < (int) sqrt((double) n)) {
  if (!id) printf ("Too many processes\n");
  MPI_Finalize();
  exit (1);
}
/* Allocate this process's share of the array. */
marked = (char *) malloc (size);
if (marked == NULL) {
  printf ("Cannot allocate enough memory\n");
  MPI_Finalize();
  exit (1);
}
for (i = 0; i <= size; i++) marked[i] = 0;

/* Set block sizes for the odd and prime arrys */
```

```
  unsigned long long start_odd_block;
  unsigned long long end_odd_block;
  unsigned long odd_block_size;
  unsigned long prime_block_size;
  unsigned long iprime_block_size;

  odd_block_size = 512 * 512;
  prime_block_size = 256 * 256;
  /* Iterate the odd array by blocks */
  for(start_odd_block = 0; start_odd_block <= size; start_odd_block += odd_block_size + 1){
    end_odd_block = start_odd_block + odd_block_size;
    if(end_odd_block > size) end_odd_block = size;
    position = 2 + 2 * start_odd_block + 1;
    /* Start from 3 */
    prime = 3;
    /* Iterate the prime array by blocks */
    for(iprime_block_size = 0; iprime_block_size <= sqrtn; iprime_block_size += prime_block_size + 1){
      index = iprime_block_size;
      do {
        if(prime * prime > 2 * end_odd_block + low_value)
          /* prime^2 outside of the block */
          break;
        /* Find the position of the next prime multiple in the block */
        position = 2 * start_odd_block + low_value;
        if(prime * prime > position)
          first = (prime * prime - position) / 2;
        else {
          if (!(position % prime)) first = 0;
          else{
            first = prime - (position % prime);
            if(!(first % 2)) first = first / 2;
            else first = (first + prime) / 2;
          }
        }
        /* Mark the multiple position in the block */
        first += start_odd_block;
        for (i = first; i <= end_odd_block; i += prime){
          marked[i] = 1;
        }
        /* Pick up the next prime from marked0 */
        while(marked0[++index]);
        prime = 2 + (2 * index + 1);
      } while (index <= iprime_block_size + prime_block_size);
    }
  }
  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);
  /* Stop the timer */
  elapsed_time += MPI_Wtime();
  /* Print the results */
  if (!id) {
    global_count++;  //Counting 2 as prime...
    printf ("S3, %llu, %d, %d, %10.6f\n", n, p, global_count, elapsed_time);
  }
  MPI_Finalize ();
  return 0;
}
```

B. Jobs

B.1. Job 1

```
#!/bin/sh
#PBS -l nodes=10:nogpu:ppn=3,walltime=01:00:00

module load gcc-4.7.2
module load mvapich2-1.9/gcc-4.7.2

n=10000000000
for i in {4..30..2}
do
  mpirun -np $i ./sieve0 $n
  mpirun -np $i ./sieve1 $n
  mpirun -np $i ./sieve2 $n
  mpirun -np $i ./sieve3 $n
done
```

B.2. Job 2

```
#!/bin/sh
#PBS -l nodes=8:nogpu:ppn=32,walltime=01:00:00

module load gcc-4.7.2
module load mvapich2-1.9/gcc-4.7.2

n=10000000000
np=256
mpirun -np $np ./sieve0 $n
mpirun -np $np ./sieve1 $n
mpirun -np $np ./sieve2 $n
mpirun -np $np ./sieve3 $n
```

C. Outputs

To compile:

```
[acald013@head ~]$ mpicc -O3 -o sieve0 sieve0.c -lm
[acald013@head ~]$ mpicc -O3 -o sieve1 sieve1.c -lm
[acald013@head ~]$ mpicc -O3 -o sieve2 sieve2.c -lm
[acald013@head ~]$ mpicc -O3 -o sieve3 sieve3.c -lm
```

The outputs of the two jobs are similar. They have five columns. The first one is a tag to identify the optimization (S0:Baseline, S1:Removing evens, S2:Remeving Bcast and S3:Reorder loops). The second one is the size of n (for these cases $n = 10^{10}$). Then, the third column shows the number of processors used. Fourth column shows the total number of primes less or equal to n (note that all the implementations computed the same number). The last column shows the execution time in seconds.

C.1. Output 1

```
[acald013@head ~]$ ./sieve_1.job
S0, 10000000000, 4, 455052511,  91.500934
S1, 10000000000, 4, 455052511,  44.175050
S2, 10000000000, 4, 455052511,  44.115029
S3, 10000000000, 4, 455052511,  17.694667
S0, 10000000000, 6, 455052511,  63.018663
S1, 10000000000, 6, 455052511,  30.871054
S2, 10000000000, 6, 455052511,  30.178889
S3, 10000000000, 6, 455052511,  12.781019
S0, 10000000000, 8, 455052511,  47.642696
S1, 10000000000, 8, 455052511,  23.740788
S2, 10000000000, 8, 455052511,  22.385219
S3, 10000000000, 8, 455052511,   8.654183
...
S0, 10000000000, 20, 455052511,  19.911593
S1, 10000000000, 20, 455052511,   9.753038
S2, 10000000000, 20, 455052511,   9.086027
S3, 10000000000, 20, 455052511,   3.541346
S0, 10000000000, 22, 455052511,  17.506931
S1, 10000000000, 22, 455052511,   8.505286
S2, 10000000000, 22, 455052511,   8.251157
S3, 10000000000, 22, 455052511,   3.237131
S0, 10000000000, 24, 455052511,  16.069959
S1, 10000000000, 24, 455052511,   7.877941
S2, 10000000000, 24, 455052511,   7.677885
S3, 10000000000, 24, 455052511,   2.313836
S0, 10000000000, 26, 455052511,  14.918970
S1, 10000000000, 26, 455052511,   7.306115
S2, 10000000000, 26, 455052511,   7.062704
S3, 10000000000, 26, 455052511,   2.799723
S0, 10000000000, 28, 455052511,  13.915105
S1, 10000000000, 28, 455052511,   6.771050
S2, 10000000000, 28, 455052511,   6.562376
S3, 10000000000, 28, 455052511,   2.542712
S0, 10000000000, 30, 455052511,  12.981133
S1, 10000000000, 30, 455052511,   6.332164
S2, 10000000000, 30, 455052511,   6.169185
S3, 10000000000, 30, 455052511,   2.373205
```

C.2. Output 2

```
[acald013@head ~]$ ./sieve_2.job
S0, 10000000000, 256, 455052511,   6.532716
S1, 10000000000, 256, 455052511,   3.173873
S2, 10000000000, 256, 455052511,   3.107938
S3, 10000000000, 256, 455052511,   0.368718
```