Parallelization of the Sieve of Eratosthenes

Mário Cordeiro

Faculdade de Engenharia da Universidade do Porto
Rua Dr. Roberto Frias, s/n 4200-465 Porto PORTUGAL
pro11001@fe.up.pt

Abstract. Algorithm parallelization plays and important role in modern multi-core architectures. The present paper presents the implementation and optimization of the algorithm to find prime numbers called Sieve of Eratosthenes. The performance, efficiency and scalability analysis was made in distinct improvements over a serial and two parallel versions. The parallelization of the algorithm was made employing two distinct technologies OpenMP and MPI. The obtained results shown that the parallel versions outperform the serial version in most cases with a factor higher than 100.

Keywords: sieve of eratosthenes, parallel computing, OpenMP, MPI

1 Introduction

Prime calculation plays nowadays an important role on secure encryption. The data encryption/decryption by public keys such as the RSA algorithm [3] make use of prime numbers. Basically the “public key” consists in the product of two large primes used to encrypt a message, and the “secret key” consists on the primes itself used to decrypt the message. Once that the two multiplied prime number only have four factors: one, itself and the two primes, the time spent to discover those primes and find the “secret key” it is proportional to the time needed to find the two primes. This means that the security of RSA is based on the very hard and deterministic process of factoring large numbers. Using bigger prime numbers increase the security of the “public key” but also increase the computational cost needed to generate those prime numbers. That is one of the main reasons why finding prime numbers is so important in the computer world and shows how important is nowadays to find bigger prime numbers efficiently.

Finding prime numbers was initially described by the Greek mathematician Eratosthenes more than two thousand years ago [6]. In their algorithm called the Sieve of Eratosthenes [7] it was described a procedure to find all prime numbers in a given range. The algorithm is based in a check table of integers used to sift multiples of known prime numbers. This eliminates the need of redundant checks on numbers that cannot be prime.

In the present paper it will be evaluated implementations of the Sieve of Eratosthenes using both serial and parallel versions. The parallelization of the algorithm will be implemented using two distinct technologies OpenMP [5] and MPI [4].
Pseudocode 2.1 Sieve of Eratosthenes algorithm

1. create a list of unmarked natural numbers 2, 3, ..., N:
   
   \[
   \text{primes}[1] := \text{false} \\
   \text{for } i := 2 \text{ to } N \text{ do} \\
   \hspace{1em} \text{primes}[i] := \text{true} \\
   \text{end}
   \]

2. initialize the seed:
   
   \[
   k := 2
   \]

3. perform the sieve until \( k^2 < N \):
   
   \[
   \text{while } (k^2 < N) \text{ do} \\
   \hspace{1em} i := k^2 \\
   \hspace{1em} (a) \text{ mark all multiples of } k \text{ between } k^2 \text{ and } N: \\
   \hspace{2em} \text{while } (i \leq N) \text{ do} \\
   \hspace{3em} \text{primes}[i] := \text{false} \\
   \hspace{3em} i := i + k \\
   \hspace{1em} \text{end} \\
   \hspace{1em} (b) \text{ find the smallest unmarked number greater than } k: \\
   \hspace{2em} \text{while } (\text{primes}[k] == \text{true}) \text{ do} \\
   \hspace{3em} k := k + 1 \\
   \hspace{1em} \text{end} \\
   \text{end}
   \]

4. unmarked numbers are primes

2 Sequential Sieve of Eratosthenes

The pseudo-algorithm used to calculate the \( k \) prime numbers below \( N \) is shown in Pseudocode 2.1.

An example of the sieve to find the all prime numbers bellow 50 is shown in Figure 1. In Step 1 it is defined the list of all natural numbers from 2 to 50. The elimination of all multiples of 2, 3, 5 and 7 is made in steps 2 to 5. Once that the square of the next unmarked number in the table \((11 \times 11)\) is grater than 50 all unmarked numbers are primes.

The complexity of the sequential Sieve of Erathosthenes algorithm is \( O(n \ln \ln n) \) and \( n \) is exponential in the number of digits.

All algorithms were implemented in C++, the program was developed and tested on Linux. For user simplicity, command line arguments are available to control the number of primes to within a specified bound and the number of threads or processes to be used in the calculation.

2.1 Single processor implementation

The single processor implementation is a serial version that executes the sieve using only one thread or process. In the present work there were evaluated three distinct versions:
2.1.1 Base algorithm

This version of the algorithm was implemented by dividing each element of the array by \( k \) checking if the remainder of the division is zero (Code 1.8 line 78). In case of the rest of the division being zero it means that \( j \) is not a prime number and it should be marked (Code 1.8 line 81). The Pseudocode 2.2 shows the basis algorithm to perform this operation.

\[
\text{Pseudocode 2.2 Checking prime numbers by division}
\]

\[
\text{for } j := k^2 \text{ to } N \text{ step 1 do}
\]
\[
\text{if } j \mod k = 0 \text{ then}
\]
\[
\text{it is not a prime}
\]
\[
\text{mark } j
\]
\[
\text{fi}
\]
\[
\text{end}
\]

In Code 1.8 line 84 it is found the smallest unmarked number greater than \( k \). The algorithm is repeated until \( k^2 < N \) (Code 1.8 line 87).
2.1.2 Optimization 1 This algorithm improvement, also known as fast marking, find \( j \) the first multiple of \( k \) on the block: \( j, j+k, j+2k, j+3k, \) etc instead of performing the checking of the remaining of the division of \( j \) by \( k \). With this change only the multiples of \( k \) are computed (2k, 3k, 4k, etc.) and marked as not being primes, this will avoid the checking if the multiples of \( j+k \) are primes, because they are not. The Pseudocode 2.3 shows the fast marking algorithm.

\[
\text{Pseudocode 2.3 Fast marking}
\]
\[
\text{for } j := k^2 \text{ to } N \text{ step } K \text{ do}
\]
\[
\quad \text{if it is not a prime then}
\]
\[
\quad \quad \text{mark } j
\]
\[
\text{end}
\]

The improvement of this algorithm is to change the test done in Code 1.8 line 78 by an fast marking loop defined in Code 1.9 line 81.

2.1.3 Optimization 2 Based on the previous algorithm, another possible improvement is an reorganization in order how loops are performed. The objective here is to allow the searching of several seeds in the same data block. The range of numbers from 2 to \( N \) was divided in equal intervals and subsequently processed in serial manner block by block. The use of smaller blocks will allow the processor optimize the memory access of the list of prime numbers and reduce cache misses. In Code 1.10 57 is defined the outer loop that performs the searching of prime numbers whiting a single block.

3 Parallel Sieve of Eratosthenes

The parallelization of the sieve of Eratosthenes is made by applying a domain decomposition, breaking the array into \( n-1 \) elements and associating a primitive task with each of these elements. Each one of those primitive tasks will mark as composite the elements in the array multiples of a particular prime (mark all multiples of \( k \) between \( k^2 \) and \( N \)). Two distinct data decompositions could be applied [8]:

3.1 Interleaved Data Decomposition

Performing an interleaved decomposition of the array elements, the process 0 will be responsible for checking natural numbers 2, \( 2+p \), \( 2+2p \), etc, processor 1 will check natural numbers 3, \( 3+p \), \( 3+2p \), etc. The main advantage of the interleaved approach lies in the easiness of finding witch process controls a given index (easily computed by \( imodp \) where \( i \) is the index number and \( p \) the process
Pseudocode 3.1 Block Data Decomposition

\[ r := n \mod p \]
\[ \text{if } r = 0 \text{ then} \]
\[ \quad \text{all blocks have same size} \]
\[ \text{else} \]
\[ \quad \text{First } r \text{ blocks have size } n/p \]
\[ \quad \text{Remaining } p - r \text{ blocks have size } n/p \]
\[ \text{fi} \]

number). The main disadvantage of this method is that such decomposition lead to a significant load imbalances among processes. It also requires some sort of reduction or broadcast operations.

3.2 Block Data Decomposition Method

This method divides the array into \( p \) contiguous blocks of roughly equal size. Let \( N \) is the number of array elements, and \( n \) is a multiple of the number of processes \( p \), the division is straightforward. This can be a problem if \( n \) is not a multiple of \( p \). Suppose \( n = 17 \), and \( p = 7 \), therefore it will give 2.43. If we give every process 2 elements, then we need 3 elements left. If we give every process 3 elements, then the array is not that large. We cannot simply give every process \( p - 1 \) processes \( \lfloor n/p \rfloor \) combinations and give the last process the left over because there may not be any elements left. If we allocate no elements to a process, it can complicate the logic of programs in which processes exchange values. Also it can lead to a less efficient utilization of the communication network.

This approach solves the block allocation but it is also needed to be retrieved witch range of elements are controlled by a particular process and also witch process controls a particular element.

3.2.1 Method #1 Suppose \( n \) is the number of elements and \( p \) is the number of processes. The first element controlled by process \( i \) is given by:

\[ i \lfloor n/p \rfloor + \min(i, r) \] (1)

The last element controlled by process \( i \) is the element before the first element controlled by process \( i + 1 \):

\[ (i + 1) \lfloor n/p \rfloor + \min(i + 1, r) - 1 \] (2)

The process controlling a particular array element \( j \) is:

\[ \min(\lfloor j/(\lfloor n/p \rfloor + 1) \rfloor, \lfloor (j - r)/(\lfloor n/p \rfloor) \rfloor) \] (3)

Figure 2 shown the block data decomposition using method #1 of an array of 17 elements in configurations of 7, 5 and 3 processes.
3.2.2 Method #2 The second scheme does not focus on all of larger blocks among the smaller-numbered processes. Suppose $n$ is the number of elements and $p$ is the number of processes. The first element controlled by process $i$ is:

$$\lfloor \frac{in}{p} \rfloor$$  \hspace{1cm} (4)

The last element controlled by process $i$ is the element before the first element controlled by process $i + 1$:

$$\lfloor \frac{(i + 1)n}{p} \rfloor - 1$$  \hspace{1cm} (5)

The process controlling a particular array element $j$ is:

$$\lfloor \frac{(p(j + 1) - 1)}{n} \rfloor$$  \hspace{1cm} (6)
3.2.3 Macros
Comparing block decompositions, we choose the second scheme because it has fewer operations in low and high indexes. The first scheme the larger blocks are held by the lowest numbered tasks; in the second scheme the larger blocks are distributed among the tasks.

C/C++ macros can be used in any of our parallel programs where a group of data items is distributed among a set of processors using block decomposition. Code 1.7 include the definition of those macros.

3.3 OpenMP implementation

With the objective of increase the performance of the algorithm taking the advantage of being executed in processor architectures with multiple CPU-cores sharing the same global memory, it will be presented the following OpenMP versions of the algorithm:

3.3.1 Base algorithm
This version is based on the optimized version described in 2.1.3 and adapted in order to run in parallel threads. In Code 1.11 line 69 there were created num_threads threads using an omp parallel for section. Each one of the separated threads runs in parallel (Code 1.11 line 70). Each one of the data blocks used have the lower value and block size defined using the BLOCK_LOW() (Code 1.11 line 72) and BLOCK_SIZE() (Code 1.11 line 73) macros. Once that only thread 0 is calculating the smallest unmarked number greater than \( k \) it was needed to include two omp barrier (Code 1.11 line 107 and line 114) to synchronize the \( k \) on all the running threads.

Finally the total number of primes found by each of the threads should be calculated. To perform this it was defined an omp atomic section (Code 1.11 line 128) to sum all the partial prime number counts.

3.3.2 Optimization 1
Taking advantage on the fact that there is only a single even prime number (number 2), the previous algorithm was modified in order to eliminate all even numbers from the list and performed computation. This will allows not only to speed-up the process of finding prime numbers but will also require the half amount of space to store the list of prime numbers.

The main changes over the previous algorithm are related with index adaptation in order to deal only with odd numbers. The lower value, high value and block size were adapted to deal only with odd numbers in the array (Code 1.12 lines 81 to 106). The fist index maintained by each thread need also to be adapted (Code 1.12 lines 126 to 148). Finally the finding of the smallest unmarked number greater than \( k \), calculated only by thread 0, should skip also all even numbers (Code 1.12 line 156).

3.3.3 Optimization 2
A drawback of the two pervious implementations is the need of perform thread synchronization when updating the value of \( k \). In this optimization were removed the two OpenMP barrier directives from the
code (Code 1.11 line 107 and line 114). These are synchronization directives that force all threads to reach a common point before any thread can continue. Since they were used to synchronize the value of \( k \), the program is changed such that every thread keeps track of \( k \) on its own. This is achieved by pre-calculating every prime from \( 3 - \sqrt{N} \) (remember that even numbers are disregarded). Each thread is then given a copy of this list of primes – \( k \) is changed by iterating through the list. This optimization allows each thread to move at its own pace, further reducing execution time.

To implement this in Code 1.13 lines 68 to 90 to each thread is given every prime from \( 3 - \sqrt{N} \). The task of finding the smallest unmarked number greater than \( k \), calculated only by thread 0, should skip also all even numbers (Code 1.12 line 156) was removed. Finally, in Code 1.12 line 180, each thread keeps track of \( k \) on its own.

### 3.4 MPI implementation

The MPI versions of the three algorithms presented are adapted version of the ones presented in section 3.3. The main objective is the use of a multiple shared-memory-systems (MPI) without OpenMP. The three MPI versions presented also consider the optimizations described in sections 3.3.2 – elimination of all even numbers from the lists/computation – 3.3.3 – each thread to move at its own pace by calculating the \( k \) values.

In terms of parallelization the keys aspects and challenges of the implementations using MPI are the switching between a multi-threaded environment to a message passing interface running on several distributed processes.

#### 3.4.1 Base algorithm

This version is based on the optimized version described in 3.3.1 and adapted in order to run in a multiple process / multiple node environment using the MPI instead of OpenMP.

The process identification and number of processes to be used by the `BLOCK_LOW()` (Code 1.14 line 44) and `BLOCK_SIZE()` (Code 1.14 line 45) macros are retrieved using the `MPI_Comm_rank()` and `MPI_Comm_size()` in Code 1.14 line 44 and 44 respectively.

Each time the process with id 0 finds and updated the smallest unmarked number greater than \( k \), it is required to broadcast that value to the other processes. The Code 1.14 line 111 show the `MPI_Bcast()` instruction used to perform that task.

Finally to calculate the sum of all prime numbers found by each of the processes it is necessary to perform a reduction operation that will sum the partial prime number counts of each process. This reduction is made by the `MPI_Reduce()` included in Code 1.14 line 126.

#### 3.4.2 Optimization 1

The changes needed to adapt the algorithm to perform the elimination of all even numbers from the lists/computation is equal of the one described in section 3.3.2. The main differences are the changing of the
two `omp barrier` (Code 1.11 line 107 and line 114) to synchronize the $k$ by an `MPI_Bcast()` instruction Code 1.15 line 152 and the `omp atomic` section (Code 1.11 line 128) to sum all the partial prime number counts by an `MPI_Reduce()` in Code 1.15 line 167.

3.4.3 Optimization 2 Once that each process in this optimization keeps the track of $k$ on its own, the pre-calculating of every prime from $3 - \sqrt{N}$ (remember that even numbers are disregarded) should be done by each process. This will avoid the need of the `MPI_Bcast()` included in the Code 1.15 line 152. The adaption of the algorithm according the rules defined in sections 3.3.2, 3.3.3 adapted to MPI will remove the dependency between processes when finding prime numbers. To sum all the partial prime number counts found by each process it is still necessary to include an `MPI_Reduce()` in Code 1.16 line 189.

4 Results

4.1 Computing platform configurations

To perform the evaluation of the nine algorithms defined in sections 2.1, 3.3 and 3.4 were used three distinct configurations using machines with an Intel(R) Core(TM)2 Quad CPU Q9300 running at 2.50GHz. The Table 1 shows the detailed information about processor cache.

<table>
<thead>
<tr>
<th>Cache Size</th>
<th>L1</th>
<th>L2</th>
<th>L3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processor Q9300</td>
<td>4 x 32 KB</td>
<td>2 x 3 MB</td>
<td>-</td>
</tr>
</tbody>
</table>

Table 1: Processor cache size information [2]

Regarding the network interface the hardware configured allowed to use gigabit ethernet.

4.1.1 Single computing node using only one core The sequential Sieve of Eratosthenes algorithms described in sections 2.1.1, 2.1.2 and 2.1.3 only required one thread to be executed.

4.1.2 Single computing node using up to 4 cores The OpenMP parallel Sieve of Eratosthenes algorithms described in sections 3.3.1, 3.3.2 and 3.3.3 were tested in four distinct configurations using 1 to 4 threads. The objective was to test the scalability of those algorithms in a multi-core platform. This configuration allowed to distribute up to 1 thread per processor core in order to distribute the load among all processor cores.
4.1.3 Up to 16 cores in 4 distributed computing nodes  The MPI parallel Sieve of Eratosthenes algorithms described in sections 3.4.1, 3.4.2 and 3.4.3 were tested distinct configurations using 1 to 16 threads. The objective was to test the scalability of those algorithms in a multi-core multi-node configuration. This configuration allowed to distribute up to 1 thread per processor core and also distribute the load among the 4 available computing nodes. The Code 1.1 show the **hostfile** configuration for the cluster of 4 nodes.

```plaintext
192.168.33.151 slots=4
192.168.33.150 slots=4
192.168.33.144 slots=4
192.168.33.142 slots=4
```

**Code 1.1: MPI hostfile configuration file**

4.2 Test scenarios

4.2.1 Testing the Single processor implementation  With the objective of testing the performance and scalability of the sequential Sieve of Eratosthenes algorithm the three implementations were executed using distinct ranges of numbers. The maximum interval was defined as being 2 to $2^{25}$. The algorithms were tested against 16 ranges with the maximum value being $\frac{1}{n}2^{25}$ with $n$ from 1 to 16. To perform this operation it was created a shell script to execute a batch operation for the 16 intervals (Code 1.2). The time and number of primes found was retrieved to an results file using the command line listed in Code 1.3. The output file generated by the script is shown in Code 1.4.

```plaintext
for i in {1..16} do
    arg='expr $i \* 2097152'
    ./bin/sieve $arg
done
```

**Code 1.2: Batch run for the sequential Sieve of Eratosthenes algorithm**

```plaintext
./run.sh > output.txt
```

**Code 1.3: Retrieve results for the batch**

```plaintext
155612 primes found between 2 and 2097152
Time: 1.963 seconds
295948 primes found between 2 and 4194304
Time: 5.303 seconds
431503 primes found between 2 and 6291456
Time: 9.470 seconds
[...]
2063690 primes found between 2 and 33554432
Time: 104.669 seconds
```

**Code 1.4: Example of the output with information on the interval, primes found and time spent by the algorithm in seconds**

This procedure was repeated for each one of the three sequential algorithms Base algorithm, Optimization 1 and Optimization 2 using the computing platform configuration defined in section 4.1.1.
4.2.2 Testing the OpenMP implementation The scalability and performance of the OpenMP implementations was done using the computing platform configuration defined in section 4.1.2. The three distinct OpenMP implementations Base algorithm, Optimization 1 and Optimization 2, were tested in configuration of processes varying from 1 to 4. The Code 1.5 shows the shell script used to retrieve the results using 1 thread. The second argument of the program is the number of threads (1 in the given example). The measures (time and number of primes found) was retrieved using the same method defined in section 4.2.1.

```bash
for i in {1..16} do
    arg='expr $i \* 2097152' .
    /bin/sieve $arg
done
```

Code 1.5: Batch run for the OpenMP Sieve of Eratosthenes algorithm with one thread

4.2.3 Testing the MPI implementation The scalability and performance of the MPI implementations was done using the computing platform configuration defined in section 4.1.3. The three distinct MPI implementations Base algorithm, Optimization 1 and Optimization 2, were tested in configuration of processes varying from 1 to 16 using 4 computing nodes. The Code 1.6 shows the shell script used to retrieve the results using 8 processes (argument -np 8). The measures (time and number of primes found) was retrieved using the same method defined in section 4.2.1.

```bash
for i in {1..16} do
    arg='expr $i \* 2097152'
    mpirun.openmpi -mca btl -openib -np 8 . /bin/sieve $arg
done
```

Code 1.6: Batch run for the MPI Sieve of Eratosthenes algorithm with 8 processes

4.3 Algorithm Evaluation

To evaluate the performance of the algorithms it was decided to use a measure based on the number of primes found per second for each algorithm implementation and configuration used. Once that the number of primes in the same interval, should be equal for all the algorithms, algorithm that found more primes per second have more performance.

4.3.1 Single processor results The obtained execution times and average prime numbers found per second are listed in the Table 2. Each row of the table contains the run for the respective interval from 2 to $1/n^{25}$. The values obtained for each one of the sequential algorithms are shown in table columns Base algorithm, Optimization 1 and Optimization 2.
The plot of Figure 4 show the compared performance obtained by each one of the algorithms, in number of primes found per second, when increasing the range interval of numbers.

4.3.2 OpenMP results In Table 3 are shown the obtained results for the OpenMP implementations. Each row represents the values obtained in the respective number of cores configuration (1 to 4). The values obtained for each one of the MPI algorithms are shown in table columns Base algorithm, Optimization 1 and Optimization 2. The values obtained are relative to the range of numbers between 2 and $2^{25}$.
The plot of Figure 6 show the compared performance obtained by each one of the OpenMP algorithms, in number of primes found per second, when increasing the number of cores (1 to 4).

Fig. 5: Evolution of the performance of the OpenMP implementations by changing the number of processor cores (1 to 4)

4.3.3 MPI results In Table 4 are shown the obtained results for the MPI implementations. Each row represents the values obtained in the respective number of cores configuration (1 to 16). The values obtained for each one of the MPI algorithms are shown in table columns Base algorithm, Optimization 1 and Optimization 2. The values obtained are relative to the range of numbers between 2 and $2^{25}$.

![Single processor implementation](image1)

![OpenMP implementation](image2)
<table>
<thead>
<tr>
<th>Found Primes</th>
<th>N</th>
<th>2^25</th>
<th>Cores</th>
<th>Base algorithm</th>
<th>Optimization 1</th>
<th>Optimization 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>2063690</td>
<td>33554432</td>
<td>1</td>
<td>1</td>
<td>0.888</td>
<td>2323974</td>
<td>0.450</td>
</tr>
<tr>
<td>2063690</td>
<td>33554432</td>
<td>1</td>
<td>2</td>
<td>0.793</td>
<td>2602382</td>
<td>0.359</td>
</tr>
<tr>
<td>2063690</td>
<td>33554432</td>
<td>1</td>
<td>3</td>
<td>0.769</td>
<td>2683601</td>
<td>0.368</td>
</tr>
<tr>
<td>2063690</td>
<td>33554432</td>
<td>1</td>
<td>4</td>
<td>0.769</td>
<td>2683601</td>
<td>0.349</td>
</tr>
<tr>
<td>2063690</td>
<td>33554432</td>
<td>1</td>
<td>5</td>
<td>0.668</td>
<td>3089355</td>
<td>0.308</td>
</tr>
<tr>
<td>2063690</td>
<td>33554432</td>
<td>1</td>
<td>6</td>
<td>0.478</td>
<td>4317341</td>
<td>0.254</td>
</tr>
<tr>
<td>2063690</td>
<td>33554432</td>
<td>1</td>
<td>7</td>
<td>0.390</td>
<td>5291510</td>
<td>0.138</td>
</tr>
<tr>
<td>2063690</td>
<td>33554432</td>
<td>1</td>
<td>8</td>
<td>0.547</td>
<td>3772740</td>
<td>0.107</td>
</tr>
<tr>
<td>2063690</td>
<td>33554432</td>
<td>1</td>
<td>9</td>
<td>0.280</td>
<td>7370318</td>
<td>0.085</td>
</tr>
<tr>
<td>2063690</td>
<td>33554432</td>
<td>1</td>
<td>10</td>
<td>0.310</td>
<td>6657061</td>
<td>0.064</td>
</tr>
<tr>
<td>2063690</td>
<td>33554432</td>
<td>1</td>
<td>11</td>
<td>0.241</td>
<td>8563025</td>
<td>0.100</td>
</tr>
<tr>
<td>2063690</td>
<td>33554432</td>
<td>1</td>
<td>12</td>
<td>0.295</td>
<td>6995556</td>
<td>0.086</td>
</tr>
<tr>
<td>2063690</td>
<td>33554432</td>
<td>1</td>
<td>13</td>
<td>0.169</td>
<td>12211178</td>
<td>0.051</td>
</tr>
<tr>
<td>2063690</td>
<td>33554432</td>
<td>1</td>
<td>14</td>
<td>0.142</td>
<td>14533021</td>
<td>0.115</td>
</tr>
<tr>
<td>2063690</td>
<td>33554432</td>
<td>1</td>
<td>15</td>
<td>0.173</td>
<td>11928838</td>
<td>0.063</td>
</tr>
<tr>
<td>2063690</td>
<td>33554432</td>
<td>1</td>
<td>16</td>
<td>0.100</td>
<td>20636890</td>
<td>0.060</td>
</tr>
</tbody>
</table>

Table 4: Execution times and average prime numbers found per second for the MPI implementation. Each row shows the measured values distinct core configurations in a 4 computer node configuration.

![Fig. 6: Evolution of the performance of the MPI implementations by changing the number of processor cores (1 to 16 in 4 distributed nodes)](image-url)
The plot of Figure 6 show the compared performance obtained by each one of the OpenMP algorithms, in number of primes found per second, when increasing the number of cores (1 to 16).

4.3.4 Single processor vs OpenMP vs MPI in a single node

The Figure 7 plots the performance of all algorithm versions for the range of numbers between 2 and $2^{25}$. The Sequential algorithms use only one core, the OpenMP use 4 cores in a single computing node and the MPI used 4 cores distributed among 4 distinct computing nodes (1 core per node).

Fig. 7: Evolution of the performance of the three implementations by changing the number of processor cores (1 to 4 cores)

4.4 Discussion

In the present section will be analyzing the data obtained by the Single processor results (section 4.3.1), OpenMP results (section 4.3.2) and MPI results (4.3.3).

Starting by analyzing results of the sequential algorithm implementations described in sections 2.1.1, 2.1.2 and 2.1.3, the Figure 4 shows that the better performance was obtained by the second optimization of the algorithm. The speed up factors plotted in Figure 8 show that the Optimization 1 has speedup factors from a minimum 65 times to a maximum of 115 times more faster than the Base algorithm. The best speedup factors where obtained by the Optimization 2 witch was 218 times faster than the Base algorithm for the range of values between 2 and $2^{25}$, the minimum speed up factor obtained for this algorithm was 93 times faster. Analyzing the trends of the graphic curves (Figure 8a) it can be concluded that continuing to increase the range of values, the performance degradation of the Base algorithm will be more significant than for the other two optimizations. By comparing the Optimization 1 and
Optimization 2 speedup factors in Figure 8b the speedup factor of the blocked algorithm vary from 1 (equal performance) to 2 times faster when considering 16 blocks. The maximum speedup factor is already reached when using blocks of data of 2,097,152 bytes ($2^{25} \times 3/16$ divided in 3 blocks of data). This value is consistent with the size of the processor cache.

![Graph](image)

Fig. 8: Speedup factors of: Optimization 1 and Optimization 2 relative to the Base algorithm (a) and Optimization 2 relative to Optimization 1 (b)

Regarding the OpenMP implementations the Figure 5 shows that the better performance was obtained by the Optimization 2 of the algorithm. The graphic also show that for small core configurations the Optimization 1 has a comparable performance to the Optimization 2, but for configurations with higher number of cores both Base algorithm and Optimization 1 shown a visible degradation (more that 2 cores). The main fact for this is related with concurrency problems of having several threads disputing the same portion of data (reading and writing the value of $k$). Once that in Optimization 2 it was removed the two omp barriers from the algorithm, the performance of is not affected by the scaling to a multi-core environment. Speedup factor range from 2.5 times faster for configurations with 1 or 2 cores to more than 10 times in configurations with 4 cores.

By analyzing Figure 4 it can be concluded that using MPI all of the three implementations scale well when increasing the number of cores. Optimization 2 shown again the better performance of the three implementations.

Finally by comparing the performance of the 9 algorithms (Figure 7) in configurations up to 4 cores it can be concluded that the performance of OpenMP Optimization 2, MPI Optimization 1 and MPI Optimization 2 have similar performance, with OpenMP having better results in lower core count configurations. This fact may be related with the overhead of communication needed by MPI that cannot outperform the OpenMP in such cases.
5 Conclusions

This paper introduced the algorithm for seeking a list of prime numbers using the Sieve of Eratosthenes given an range of numbers from 2 to $N$. In the first sections where revealed some weakness of the algorithm regarding the scaling to higher ranges of numbers.

The parallelization of the algorithm revealed to be a good strategy to scale the algorithm in multi-core architectures using OpenMP. The use of MPI was also addressed to be used in a multiple node computing environment. In both approaches OpenMP and MPI were compared optimizations regarding the elimination of even integers (all primes are odd except 2) and in the removal of thread synchronization / broadcast operations by introducing redundant portions of the code that can be performed by each thread or process.

The algorithms where tested in multiple computing configurations using a quad-core architecture and 4 computing nodes in the MPI versions. The results shown that the OpenMP could be a good solution when using multiple core architecture but programmer should be aware of thread synchronization issues that may degrade the performance. If the objective is to scale the algorithm to a multi node architecture the MPI revealed to have good scaling capabilities over a multi node configuration. Nevertheless the overhead of inter process communication used by the MPI this solution revealed to have performance to OpenMP even in a single computing node configuration.

Atkin and Bernstein [1] described an improved version of the sieve of Eratosthenes, the parallelization of that algorithm using OpenMP and MPI and the respective benchmark over the presented implementations could be pointed as future work.
Bibliography


Code Listings

```
#include <iostream>
#include <cmath>
#include <cstdio>
#include <cstdlib>
#include <sys/time.h>

#define BLOCK_LOW(id, p, n) ((id)*(n)/(p))
#define BLOCK_HIGH(id, p, n) (BLOCK_LOW((id)+1, (p), (n)))
#define BLOCK_SIZE(id, p, n) ((BLOCK_HIGH((id), (p), (n))) - (BLOCK_LOW((id), (p), (n))))

void usage(void)
{
    std::cout << "sieve <max_number>" << std::endl;
    std::cout << "<max_number> range between 2 and N." << std::endl;
}
```

Code 1.7: C/C++ macros used to distribute data items among a set of processors using block decomposition
20 int main(int argc, char ** argv)
19 {
20     if (argc != 2)
21     {
22         std::cout << "Invalid number of arguments!" << std::endl;
23         usage();
24         return 0;
25     }
26
27     int range_max = atoi(argv[1]);
28     if (range_max < 2)
29     {
30         std::cout << "<max_number> Must be greater than or equal to 2."
31             << std::endl;
32         usage();
33         return 0;
34     }
35
36     // Global k
37     int k = 2;
38
39     // Global count
40     int count = 0;
41
42     int low_value = 2;
43
44     // block of data
45     char * marked = (char *) malloc(range_max);
46     if (marked == 0)
47     {
48         std::cout << "Cannot allocated enough memory."
49             << std::endl;
50         exit(1);
51     }
52     for (int i = 0; i < range_max; ++i)
53     { marked[i] = 0; }
54
55     int first_index = 0;
56     do
57     {
58         if (k > low_value)
59             first_index = k - low_value + k;
60         else if (k * k > low_value)
61             first_index = k * k - low_value;
62         else if (low_value % k == 0)
63             first_index = 0;
64         else
65             first_index = k - (low_value % k);
66         for (int i = first_index; i < range_max; i++)
67             if ((i % k) == 0)
68                 marked[i] = 1;
69     } while (marked[k++]);
while ( k * k <= range_max );
for ( int i = 0; i < range_max; ++i )
{
    if ( marked[i] == 0 )
    {
        ++count;
    }
}
free(marked); marked = 0;
std::cout << count << " primes found between 2 and " << range_max << std::endl;
return 0;
}

Code 1.8: Single process Sieve of Eratosthenes with division checking

#include <iostream>
#include <cmath>
#include <cstdio>
#include <cstdlib>
#include <sys/time.h>

#define BLOCKLOW(id,p,n) ((id)*(n)/(p))
#define BLOCK_HIGH(id,p,n) (BLOCKLOW((id) + 1,p,n) + 1)
#define BLOCK_SIZE(id,p,n) (BLOCK_HIGH(id,p,n) - BLOCKLOW(id,p,n) + 1)

void usage(void)
{
    std::cout << "sieve <max_number>" << std::endl;
    std::cout << "<max_number> range between 2 and N." << std::endl;
}

int main(int argc, char ** argv) {
    if ( argc != 2 )
    {
        std::cout << "Invalid number of arguments!" << std::endl;
        usage();
        return 0;
    }
    int range_max = atoi(argv[1]);
    if ( range_max < 2 )
    {
        std::cout << "<max_number> Must be greater than or equal to 2." << std::endl;
        usage();
        return 0;
    }
    // Global k
    int k = 2;
    // Global index
    int prime_index = 0;
    // Global count
    int count = 0;

```cpp
int low_value = 2;

// block of data
char * marked = (char *) malloc(range_max);
if (marked == 0)
    {
        std::cout << "Cannot allocated enough memory." << std::endl;
        exit(1);
    }

for (int i = 0; i < range_max; ++i)
    {
        marked[i] = 0;
    }

int first_index = 0;
do
    {
        if (k > low_value)
            {
                first_index = k - low_value + k;
            }
        else if (k * k > low_value)
            {
                first_index = k * k - low_value;
            }
        else if (low_value % k == 0)
            {
                first_index = 0;
            }
        else
            {
                first_index = k - (low_value % k);
            }
        for (int i = first_index; i < range_max; i += k)
            {
                marked[i] = 1;
            }
        while (marked[++prime_index])
            k = prime_index + 2;
    } while (k * k <= range_max);

for (int i = 0; i < range_max; ++i)
    {
        if (marked[i] == 0)
            {
                ++count;
            }
    }

free(marked); marked = 0;
std::cout << count << " primes found between 2 and " << range_max << std::endl;
return 0;
```

Code 1.9: Single process Sieve of Eratosthenes with fast marking
#include <cstdlib>
#include <sys/time.h>

#define BLOCKLOW(id,p,n) ((id)*(n)/(p))
#define BLOCKHIGH(id,p,n) (BLOCKLOW((id)+1,p,n)-1)
#define BLOCK_SIZE(id,p,n) (BLOCKHIGH(id,p,n)-BLOCKLOW(id,p,n)+1)

void usage(void)
{
    std::cout << "sieve <max_number> <block_count>" << std::endl;
    std::cout << "<max_number> range between 2 and N. " << std::endl;
    std::cout << "<block_count> is the number of blocks to use." << std::endl;
}

int main(int argc, char **argv)
{
    if (argc != 3)
    {
        std::cout << "Invalid number of arguments!" << std::endl;
        usage();
        return 0;
    }

    int range_max = atoi(argv[1]);
    int num_blocks = atoi(argv[2]);

    if (range_max < 2)
    {
        std::cout << "<max_number> Must be greater than or equal to 2."
                   << std::endl;
        usage();
        return 0;
    }

    if (num_blocks < 1)
    {
        std::cout << "<block_count> between 1 and <max_number>" << std::endl;
        usage();
        return 0;
    }

    int temp = (range_max - 1) / num_blocks;
    if ((1 + temp) < (int)sqrt((double)range_max))
    {
        std::cout << "Too many blocks!" << std::endl;
        std::cout << "Block size should be greater equal than sqrt(n)."
                   << std::endl;
        exit(1);
    }

    // Global count
    int count = 0;

    int thread_id = 0;
    for (thread_id = 0; thread_id < num_blocks; ++thread_id)
    {
        int k = 2;
        int prime_index = 0;
        int low_value = 2 + BLOCKLOW(thread_id, num_blocks, range_max - 1);
        int block_size = BLOCK_SIZE(thread_id, num_blocks, range_max - 1);
char * marked = (char *) malloc(block_size);

if (marked == 0)
{
    std::cout << "Thread " << thread_id << " cannot allocated 
    enough memory." << std::endl;
    exit(1);
}

for (int i = 0; i < block_size; ++i) marked[i] = 0;

int first_index = 0;

for (;
    if (k > low_value)
    {
        first_index = k - low_value + k;
    }
    else if (k * k > low_value)
    {
        first_index = k * k - low_value;
    }
    else
    {
        if (low_value % k == 0) first_index = 0;
        else first_index = k - (low_value % k);
    }

    for (int i = first_index; i < block_size; i += k)
    {
        marked[i] = 1;
    }

    while (marked[++prime_index]) ;
    k = prime_index + 2;
} while (k * k <= range_max);

int local_count = 0;
for (int i = 0; i < block_size; ++i)
{
    if (marked[i] == 0)
    {
        ++local_count;
    }
}

free(marked); marked = 0;

count += local_count;

std::cout << count << " primes found between 2 and " << range_max << 
           std::endl;

return 0;
}

Code 1.10: Single process Blocked Sieve of Eratosthenes with fast marking

1  #include <omp.h>
2  #include <iostream>
3  #include <cmath>
4  #include <cstdio>
5  #include <cstdlib>
6  #include <sys/time.h>
# define BLOCK_LOW(id, p, n) ((id)∗(n))/(p)
# define BLOCK_HIGH(id, p, n) (BLOCK_LOW((id)+1,p,n)-1)
# define BLOCK_SIZE(id, p, n) (BLOCK_HIGH(id, p, n)-BLOCK_LOW(id, p, n)+1)

void usage(void)
{
    cout << "sieve <max_number> <thread count>" << endl;
    cout << "<max_number> range between 2 and N." << endl;
    cout << "<thread count> is the number of threads to use." << endl;
}

int main(int argc, char **argv)
{
    if (argc != 3)
    {
        cout << "Invalid number of arguments!" << endl;
        usage();
        return 0;
    }

    int range_max = atoi(argv[1]);
    int num_threads = atoi(argv[2]);

    if (range_max < 2)
    {
        cout << "<max_number> Must be greater than or equal to 2." << endl;
        usage();
        return 0;
    }

    if (num_threads < 1)
    {
        cout << "<thread count> between 1 and <max_number>" << endl;
        usage();
        return 0;
    }

    if (num_threads > omp_get_max_threads())
    {
        num_threads = omp_get_max_threads();
    }

    int temp = (range_max - 1) / num_threads;
    if ((1 + temp) < (int)sqrt((double)range_max))
    {
        cout << "Too many threads!" << endl;
        cout << "Thread should be greater equal than sqrt(n)." << endl;
        exit(1);
    }

    // Global k
    int k = 2;

    // Global count
    int count = 0;

    int thread_id = 0;
    omp_set_num_threads(num_threads);
    #pragma omp parallel for default(shared) private(thread_id)
    for (thread_id = 0; thread_id < num_threads; ++thread_id)
    {
int low_value = 2 + BLOCK_LOW(thread_id, num_threads, range_max - 1);
int block_size = BLOCK_SIZE(thread_id, num_threads, range_max - 1);
char * marked = (char *) malloc(block_size);
if (marked == 0)
{
    std::cout << "Thread " << thread_id << " cannot allocated enough memory." << std::endl;
    exit(1);
}
for (int i = 0; i < block_size; ++i) marked[i] = 0;
if (i != 0; i < block_size; ++i) marked[i] = 0;

int first_index = 0;
do
{
    if (k > low_value)
    {
        first_index = k - low_value + k;
    }
    else if (k * k > low_value)
    {
        first_index = k * k - low_value;
    }
    else
    {
        if (low_value % k == 0) first_index = 0;
        else first_index = k - (low_value % k);
    }
    for (int i = first_index; i < block_size; i += k)
    {
        marked[i] = 1;
    }
    #pragma omp barrier
    if (thread_id == 0)
    {
        while (marked[++prime_index]);
        k = prime_index + 2;
    }
    #pragma omp barrier
} while (k * k <= range_max);
int local_count = 0;
for (int i = 0; i < block_size; ++i)
{
    if (marked[i] == 0)
    {
        ++local_count;
    }
}
free(marked); marked = 0;
#pragma omp atomic
count += local_count;
}
std::cout << count << " primes found between 2 and " << range_max << std::endl;
return 0;
#include <omp.h>
#include <iostream>
#include <cmath>
#include <cstdio>
#include <cstdlib>
#include <sys/time.h>

#define BLOCK_LOW(id, p, n) ((id)*(n)/(p))
#define BLOCK_HIGH(id, p, n) (BLOCK_LOW((id)+1, p, n)-1)
#define BLOCK_SIZE(id, p, n) (BLOCK_HIGH(id, p, n)-BLOCK_LOW(id, p, n)+1)

void usage(void)
{
    std::cout << "sieve <range> <thread count>" << std::endl;
    std::cout << "<max number> range between 2 and N." << std::endl;
    std::cout << "<thread count> is the number of threads to use." << std::endl;
}

int main(int argc, char ** argv)
{
    TimeUtils::ScopedTimer t;
    if (argc != 3)
    {
        std::cout << "Invalid number of arguments!" << std::endl;
        usage();
        return 0;
    }

    int range_max = atoi(argv[1]);
    int num_threads = atoi(argv[2]);
    if (range_max < 2)
    {
        std::cout << "<max number> Must be greater than or equal to 2." << std::endl;
        usage();
        return 0;
    }
    if (num_threads < 1)
    {
        std::cout << "<thread count> between 1 and <max number>" << std::endl;
        usage();
        return 0;
    }
    if (num_threads >omp_get_max_threads())
    {
        num_threads = omp_get_max_threads();
    }

    int temp = (range_max - 1) / num_threads;
    if ((1 + temp) < (int)sqrt((double)range_max))
    {
        std::cout << "Too many threads!" << std::endl;
        std::cout << "Thread should be greater equal than sqrt(n)." << std::endl;
        exit(1);
    }
}
int k = 3;
int prime_index = 0;
int count = 1;
int thread_id = 0;
omp_set_num_threads(num_threads);
#pragma omp parallel for default(shared) private(thread_id)
for (thread_id = 0; thread_id < num_threads; ++thread_id)
{
    int low_value = 2 + BLOCK_LOW(thread_id, num_threads, range_max - 1);
    int high_value = 2 + BLOCK_HIGH(thread_id, num_threads, range_max - 1);
    int block_size = BLOCK_SIZE(thread_id, num_threads, range_max - 1);

    if (low_value % 2 == 0)
    {
        if (high_value % 2 == 0)
        {
            block_size = (int) floor((double)block_size / 2.0);
            high_value--;
        }
        else
        {
            block_size = block_size / 2;
        }

        low_value++;
    }
    else
    {
        if (high_value % 2 == 0)
        {
            block_size = block_size / 2;
            high_value--;
        }
        else
        {
            block_size = (int) ceil((double)block_size / 2.0);
        }
    }
}

char * marked = (char *)malloc(block_size);
if (marked == 0)
{
    std::cout << "Thread " << thread_id << " cannot allocated enough memory." << std::endl;
    exit(1);
}

for (int i = 0; i < block_size; ++i) marked[i] = 0;
int first_index = 0;
do
```c
if (k >= low_value)
{
    first_index = ((k - low_value) / 2) + k;
}
else if (k * k > low_value)
{
    first_index = (k * k - low_value) / 2;
}
else
{
    if (low_value % k == 0)
        first_index = 0;
    else
        first_index = 1;
    while ((low_value + (2 * first_index)) % k != 0)
        ++first_index;
}
for (int i = first_index; i < block_size; i += (k))
{
    marked[i] = 1;
}
#pragma omp barrier
if (thread_id == 0)
{
    while (marked[++prime_index]);
    k = (3 + (prime_index * 2));
}
#pragma omp barrier
while (k * k <= range_max);
int local_count = 0;
for (int i = 0; i < block_size; ++i)
{
    if (marked[i] == 0)
        ++local_count;
}
free(marked); marked = 0;
#pragma omp atomic
count += local_count;
std::cout << count << " primes found between 2 and " << range_max << std::endl;
return 0;
}
```

Code 1.12: OpenMP Sieve of Eratosthenes with all even numbers elimination from the lists/computation

```
#include <iostream>
#include <vector>
#include <cmath>
#include <cstdio>
#include <cstdlib>

#include <sys/time.h>

#define BLOCK_LOW(id, p, n) ((id)*(n)/(p))
#define BLOCK_HIGH(id, p, n) (BLOCK_LOW((id)+1,p,n)-1)
#define BLOCK_SIZE(id, p, n) (BLOCK_HIGH(id,p,n)-BLOCK_LOW(id,p,n)+1)

void usage(void)
{
  std::cout << "Sieve <range> <thread count>" << std::endl;
  std::cout << "<max_number> range between 2 and N." << std::endl;
  std::cout << "<thread count> is the number of threads to use." << std::endl;
}

int main(int argc, char **argv)
{
  TimeUtils::ScopedTimer t;
  if (argc != 3)
  {
    std::cout << "Invalid number of arguments!" << std::endl;
    usage();
    return 0;
  }

  int range_max = atoi(argv[1]);
  int num_threads = atoi(argv[2]);

  if (range_max < 2)
  {
    std::cout << "<max_number> Must be greater than or equal to 2." << std::endl;
    usage();
    return 0;
  }

  if (num_threads < 1)
  {
    std::cout << "<thread count> between 1 and <max_number> " << std::endl;
    usage();
    return 0;
  }

  if (num_threads > omp_get_max_threads())
  {
    num_threads = omp_get_max_threads();
  }

  int temp = (range_max - 1) / num_threads;
  if ((1 + temp) < (int)sqrt((double)range_max))
  {
    std::cout << "Too many threads requested!" << std::endl;
    std::cout << "The first thread must have a block size >= sqrt(n) ." << std::endl;
    exit(1);
  }

  int k = 3;
```c
int count = 1;
int sqrtn = ceil(sqrt((double)range_max));
char * pre_marked = (char *)malloc(sqrtn + 1);
p_re_marked[0] = 1;
p_re_marked[1] = 1;
for (int i = 2; i <= sqrtn; ++i) pre_marked[i] = 0;
int pre_k = 2;
do {
    int base = pre_k * pre_k;
    for (int i = base; i <= sqrtn; i += pre_k) pre_marked[i] = 1;
} while (pre_k * pre_k <= sqrtn);
std::vector<int> kset;
for (int i = 3; i <= sqrtn; ++i)
    if (pre_marked[i] == 0) kset.push_back(i);
free(pre_marked);
if (kset.empty())
    std::cout << "There is 1 prime less than or equal to 2." << std::endl;
    exit(0);
int thread_id = 0;
int kindex = 0;
omp_set_num_threads(num_threads);
#pragma omp parallel default(shared) private(thread_id, kindex, k)
for (thread_id = 0; thread_id < num_threads; ++thread_id)
{
    kindex = 0;
    k = kset[kindex];
    int low_value = 2 + BLOCK_LOW(thread_id, num_threads, range_max - 1);
    int high_value = 2 + BLOCK_HIGH(thread_id, num_threads, range_max - 1);
    int block_size = BLOCK_SIZE(thread_id, num_threads, range_max - 1);
    if (low_value % 2 == 0)
        if (high_value % 2 == 0)
        {
            block_size = (int)floor((double)block_size / 2.0);
            high_value--;
        }
        else
        {
            block_size = block_size / 2;
        }
    if (low_value++)
        else
        {
            if (high_value % 2 == 0)
            {
            }
```
block_size = block_size / 2;
high_value--;}
else
{
    block_size = (int)ceil((double)block_size / 2.0);
}
}

char * marked = (char *)malloc(block_size);
if (marked == 0)
{
    std::cout << "Thread " << thread_id << " cannot allocated enough memory. " << std::endl;
    exit(1);
}

for (int i = 0; i < block_size; ++i) marked[i] = 0;
int first_index = 0;
do
{
    if (k >= low_value)
    {
        first_index = ((k - low_value) / 2) + k;
    }
    else if (k * k > low_value)
    {
        first_index = (k * k - low_value) / 2;
    }
    else
    {
        if (low_value % k == 0)
        {
            first_index = 0;
        }
        else
        {
            first_index = 1;
            while ((low_value + (2 * first_index)) % k != 0)
                ++first_index;
        }
    }
    for (int i = first_index; i < block_size; i += (k))
    {
        marked[i] = 1;
    }
    k = kset[++kindex];
} while (k * k <= range_max && kindex < (int)kset.size());
int local_count = 0;
for (int i = 0; i < block_size; ++i)
{
    if (marked[i] == 0)
    {
        ++local_count;
    }
}
free(marked); marked = 0;
#pragma omp atomic
count += local_count;
```cpp
#include <mpi.h>
#include <iostream>
#include <cmath>
#include <cstdio>
#include <cstdlib>

#define BLOCKLOW(id, p, n) ((id)*(n)/(p))
#define BLOCKHIGH(id, p, n) (BLOCKLOW((id)+1, p, n)−1)
#define BLOCKSIZE(id, p, n) (BLOCKLOW((id)+1, p, n)−BLOCKLOW(id, p, n))

void usage(void)
{
    std::cout << "sieve <max_number>" << std::endl;
    std::cout << "<max_number> range between 2 and 2N." << std::endl;
}

int main(int argc, char *argv[])
{
    double elapsed_time;
    MPI_Init (&argc, &argv);
    MPI_Barrier(MPI_COMM_WORLD);
    elapsed_time = −MPI_Wtime();
    int process_id;
    MPI_Comm_rank (MPI_COMM_WORLD, &process_id);
    int num_processes;
    MPI_Comm_size (MPI_COMM_WORLD, &num_processes);
    if (argc != 2)
    {
        if (process_id == 0)
        {
            usage();
            MPI_Finalize();
            exit (1);
        }
    }
    int range_max = atoi(argv[1]);
    int low_value = 2 + BLOCKLOW(process_id, num_processes, range_max−1);
    int block_size = BLOCKSIZE(process_id, num_processes, range_max−1);
    int temp = (range_max − 1) / num_processes;
    if ((2 + temp) < (int) sqrt((double) range_max))
    {
        if (process_id == 0)
        {
            std::cout << "Too many processed!" << std::endl;
            std::cout << "Process should be greater equal than sqrt(n)." << std::endl;
        }
    }
        std::cout << count << " primes found between 2 and " << range_max << std::endl;
    return 0;
}
```

Code 1.13: OpenMP Sieve of Eratosthenes with each thread maintaining the seed list.
char * marked = (char *)malloc(block_size);
if (marked == NULL)
{
    std::cout << "Process " << process_id << " cannot allocated enough memory." << std::endl;
    MPI_Finalize();
    exit (1);
}

for (int i = 0; i < block_size; i++)
{
    marked[i] = 0;
}

int first_index;
if (process_id == 0)
{
    first_index = 0;
}

int k = 2;
int prime_index = 0;
int count = 0;
do
{
    if (k * k > low_value)
    {
        first_index = k * k - low_value;
    }
    else
    {
        if (low_value % k == 0) first_index = 0;
        else first_index = k - (low_value % k);
    }
    for (int i = first_index; i < block_size; i += k)
    {
        marked[i] = 1;
    }
    if (process_id == 0)
    {
        while (marked[++prime_index]);
        k = prime_index + 2;
    }
    MPI_Bcast(&k, 1, MPI_INT, 0, MPI_COMM_WORLD);
} while (k * k <= range_max);

int local_count = 0;
for (int i = 0; i < block_size; ++i)
{
    if (marked[i] == 0)
    {
        ++local_count;
    }
}
free (marked) ; marked = 0 ;
MPI_Reduce (&local_count, &count, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);

elapsed_time += MPI_Wtime();
if (process_id == 0)
{
    std::cout << count << " primes found between 2 and " <<
                range_max << std::endl;
    char st[100];
    sprintf(st, "Time: %3.3f seconds\n", elapsed_time);
    std::cout << st;
}
MPI_Finalize();
return 0;

Code 1.14: MPI Sieve of Eratosthenes

#include <mpi.h>
#include <iostream>
#include <cmath>
#include <cstdio>
#include <cstdlib>

#define BLOCK_LOW(id, p, n) ((id)*n)/(p)
#define BLOCK_HIGH(id, p, n) (BLOCK_LOW((id)+1, p, n) - 1)
#define BLOCK_SIZE(id, p, n) ((BLOCK_LOW((id)+1, p, n)) - BLOCK_LOW(id, p, n))

void usage (void)
{
    std::cout << "sieve <max_number>" << std::endl;
    std::cout << "<max_number> range between 2 and N." << std::endl;
}

int main (int argc, char *argv[])
{
    double elapsed_time;
    MPI_Init (&argc, &argv);
    MPI_Barrier(MPI_COMM_WORLD);
    elapsed_time = -MPI_Wtime();
    int process_id;
    MPI_Comm_rank (MPI_COMM_WORLD, &process_id);
    int num_processes;
    MPI_Comm_size (MPI_COMM_WORLD, &num_processes);
    if (argc != 2)
    {
        if (process_id == 0)
        {
            usage ( );
            MPI_Finalize ( );
            exit (1);
        }
int range_max = atoi(argv[1]);

int low_value = 2 + BLOCK_LOW(process_id, num_processes, range_max - 1);
int high_value = 2 + BLOCK_HIGH(process_id, num_processes, range_max - 1);
int block_size = BLOCK_SIZE(process_id, num_processes, range_max - 1);

if (low_value % 2 == 0)
{
  if (high_value % 2 == 0)
  {
    block_size = (int)floor((double)block_size / 2.0);
    high_value--;
  } else
  {
    block_size = block_size / 2;
  }
  low_value++;
} else
{
  if (high_value % 2 == 0)
  {
    block_size = block_size / 2;
    high_value--;
  } else
  {
    block_size = (int)ceil((double)block_size / 2.0);
  }
}

int temp = (range_max - 1) / num_processes;

if ((2 + temp) < (int)sqrt((double)range_max))
{
  if (process_id == 0)
  {
    std::cout << "Too many processed!" << std::endl;
    std::cout << "Process should be greater equal than sqrt(n)." << std::endl;
  }
  MPI_Finalize();
  exit (1);
}
char * marked = (char *)malloc(block_size);
if (marked == NULL)
{
  std::cout << "Process " << process_id << " cannot allocated enough memory." << std::endl;
  MPI_Finalize();
  exit (1);
}
for (int i = 0; i < block_size; i++)
{
  marked[i] = 0;
}
int first_index;
```c
if (process_id == 0)
{
    first_index = 0;
}

int k = 3;
int prime_index = 0;
int count = 1;
do
{
    if (k >= low_value)
    {
        first_index = ((k - low_value) / 2) + k;
    }
    else if (k * k > low_value)
    {
        first_index = (k * k - low_value) / 2;
    }
    else
    {
        if (low_value % k == 0)
        {
            first_index = 0;
        }
        else
        {
            first_index = 1;
            while ((low_value + (2 * first_index)) % k != 0)
                ++first_index;
        }
    }
    for (int i = first_index; i < block_size; i += (k))
    {
        marked[i] = 1;
    }
    if (process_id == 0)
    {
        while (marked[++prime_index]);
    }
    k = (3 + (prime_index * 2));
}
MPI_Bcast (&k, 1, MPI_INT, 0, MPI_COMM_WORLD);
} while (k * k <= range_max);
int local_count = 0;
for (int i = 0; i < block_size; ++i)
{
    if (marked[i] == 0)
    {
        ++local_count;
    }
}
free(marked); marked = 0;
MPI_Reduce (&local_count, &count, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);
elapsed_time += MPI_Wtime();
if (process_id == 0)
```
Code 1.15: MPI Sieve of Eratosthenes with all even numbers elimination from the lists/computation

```cpp
#include <mpi.h>
#include <iostream>
#include <vector>
#include <cmath>
#include <cstdio>
#include <cstdlib>

#define BLOCK_LOW(id, p, n) ((id)*(n)/(p))
#define BLOCK_HIGH(id, p, n) ((BLOCK_LOW(((id)+1),p,n)-1)
#define BLOCK_SIZE(id, p, n) ((BLOCK_LOW(((id)+1),p,n))-(BLOCK_LOW(id,p,n)))

void usage(void)
{
    std::cout << "sieve <max_number>" << std::endl;
    std::cout << "<max_number> range between 2 and N." << std::endl;
}

int main (int argc, char *argv[])
{
    double elapsed_time;
    MPI_Init (&argc, &argv);
    MPI_Barrier (MPI_COMM_WORLD);
    elapsed_time = -MPI_Wtime();
    int process_id;
    MPI_Comm_rank (MPI_COMM_WORLD, &process_id);
    int num_processes;
    MPI_Comm_size (MPI_COMM_WORLD, &num_processes);
    if (argc != 2)
    {
        if (process_id == 0)
        {
            usage();
            MPI_Finalize();
            exit (1);
        }
    }
    int range_max = atoi(argv[1]);
    int sqrttn = ceil(sqrt((double)range_max));
    char * pre_marked = (char*)malloc(sqrttn + 1);
    pre_marked[0] = 1;
    pre_marked[1] = 1;
    std::cout << count << " primes found between 2 and " << range_max << std::endl;
    char st[100];
    sprintf(st, "Time: %.3f seconds\n", elapsed_time);
    std::cout << st;
    MPI_Finalize ();
    return 0;
}
```
for (int i = 2; i <= sqrt(n); ++i) pre_marked[i] = 0;
int pre_k = 2;
do {
    int base = pre_k * pre_k;
    for (int i = base; i <= sqrt(n); i += pre_k) pre_marked[i] = 1;
} while (pre_k * pre_k <= sqrt(n));
std::vector<int> kset;
for (int i = 3; i <= sqrt(n); ++i)
    if (pre_marked[i] == 0) kset.push_back(i);
free(pre_marked);
if (kset.empty()) {
    std::cout << "There is 1 prime less than or equal to 2."
        << std::endl;
    exit(0);
}
int low_value = 2 + BLOCK_LOW(process_id, num_processes, range_max - 1);
int high_value = 2 + BLOCK_HIGH(process_id, num_processes, range_max - 1);
int block_size = BLOCK_SIZE(process_id, num_processes, range_max - 1);
if (low_value % 2 == 0) {
    if (high_value % 2 == 0) {
        block_size = (int)floor((double)block_size / 2.0); 
        high_value--;  
    } else {
        block_size = block_size / 2; 
    }
    low_value++;  
} else {
    if (high_value % 2 == 0) {
        block_size = block_size / 2;  
        high_value--;  
    } else {
        block_size = (int)ceil((double)block_size / 2.0);  
    }
}
int temp = (range_max - 1) / num_processes;
if ((2 + temp) < (int)sqrt((double)range_max)) {
    if (process_id == 0) {
        std::cout << "Too many processed!" << std::endl;
    } else {
        std::cout << "Too many processed!" << std::endl;
    }
std::cout << "Process should be greater equal than sqrt(n)." << std::endl;

MPI_Finalize();
exit (1);
}

char * marked = (char *)malloc(block_size);
if (marked == NULL){
    std::cout << "Process " << process_id << " cannot allocated enough memory." << std::endl;
    MPI_Finalize();
    exit (1);
}

for (int i = 0; i < block_size; i++) {
    marked[i] = 0;
}

int first_index;
if (process_id == 0) {
    first_index = 0;
}
int kindex = 0;
int k = kset[kindex];
int count = 1;
do {
    if (k >= low_value) {
        first_index = ((k - low_value) / 2) + k;
    } else if (k * k > low_value) {
        first_index = (k * k - low_value) / 2;
    } else {
        if (low_value % k == 0) {
            first_index = 0;
        } else {
            first_index = 1;
            while ((low_value + (2 * first_index)) % k != 0)
                ++first_index;
        }
    }
    for (int i = first_index; i < block_size; i += (k)) {
        marked[i] = 1;
    }
    k = kset[++kindex];
} while (k * k <= range_max && kindex < (int)kset.size());
int local_count = 0;
for (int i = 0; i < block_size; ++i) {
    if (marked[i] == 0) {
        ++local_count;
    }
}
free(marked); marked = 0;
MPI_Reduce(&local_count, &count, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);
elapsed_time += MPI_Wtime();
if (process_id == 0) {
    std::cout << count << " primes found between 2 and " <<
    range_max << std::endl;
    char st[100];
    sprintf(st, "Time: %3.3f seconds\n", elapsed_time);
    std::cout << st;
}
MPI_Finalize();
return 0;

Code 1.16: MPI Sieve of Eratosthenes with each thread maintaining the seed list