Parallel Implementation of the Prime Number Sieve

Gary Steelman,

Missouri University of Science and Technology

Computer Science 387, Dr. Ercal

Contents

[Background 3](#_Toc285582553)

[Method 4](#_Toc285582554)

[Implementation Specifics 4](#_Toc285582555)

[Data and Task Partitioning Among Processes 4](#_Toc285582556)

[Data Collection and Reports 5](#_Toc285582557)

[Calculation of Average Timings 5](#_Toc285582558)

[Calculation of Speedup and Efficiency 6](#_Toc285582559)

[Use of Optimizations 6](#_Toc285582560)

[Data 7](#_Toc285582561)

[Table of Average Run-Times 7](#_Toc285582562)

[Table of Speedup Ratios 8](#_Toc285582563)

[Table of Efficiency Ratios 8](#_Toc285582564)

[Completely optimized vs Non-Optimized Timings 9](#_Toc285582565)

[Number of Primes in [0, 1 billion] 9](#_Toc285582566)

[First 20 Primes after 1 million 9](#_Toc285582567)

[Best Run-Time for P=16 and N=10 billion 10](#_Toc285582568)

[Highest Prime Number Found 10](#_Toc285582569)

[Analysis of Data 11](#_Toc285582570)

[Analysis of Average Run-Times 11](#_Toc285582571)

[Analysis of Speedup Ratios 11](#_Toc285582572)

[Analysis of Efficiency Ratios 12](#_Toc285582573)

[Appendix 13](#_Toc285582574)

# Background

Prime numbers are important in many fields of research: mathematics, engineering, and computing, specifically computer security. In computer security it is beneficial to have extremely large prime numbers because large prime numbers are extremely difficult to factor and can be used for data encryption and authentication.

A parallel programming implementation of the Sieve of Eratosthenes is desired for fast and efficient searches for large prime numbers. The parallel implementation is shown to be dramatically faster than the sequential implementation for numbers of processes > 2.

The parallel implementation of the Sieve is almost embarrassingly parallel, requiring little communication between processes for location of primes. In the sequential implementation, the search space is iterated over from beginning to end repeatedly, marking out multiples of the least number found unmarked. The parallel implementation isn’t very different. The search space is partitioned from the global lower bound to the square root of the global upper bound; the remainder uniformly based on the number of processes from the square root of the global upper bound to the global upper bound. There is a master process that searches in the first interval and broadcasts the least number found unmarked to all processes. All processes calculate the first multiple of that number in their interval and proceed to mark out every multiple of that number in their interval. This is iterated until the master process reaches the square root of the global upper bound and the Sieve completes.

This report discusses the speedup, efficiency, and runtime length of varying numbers of processes and sizes of search spaces.

The Sieve was implemented using the Message Passing Interface (MPI) open-source libraries for C++.

# Method

## Implementation Specifics

The Sieve is an implementation of the Sieve of Eratosthenes instead of the faster, more efficient Sieve of Atkin because the Sieve of Eratosthenes was covered in class. To obtain accurate results and to visualize efficiency and speedup of the Sieve, it was run with the following dynamic values:

* N, a varying maximum searchable range
* P, a varying number of processes
* P, a varying number of processors such that each process ran on its own processor.

Each run was completed on the Missouri University of Science and Technology cluster. The cluster is a crossbar network connected by a gigabit intranet.

Two specific optimizations were included in this implementation:

1. The master process only searches to because all numbers greater than have already had their multiples marked non-prime by previous iterations of the algorithm.
2. The beginning value for each iteration starts at n=i2 for all processes because of the same principle in point 1.

The language specific details can be viewed in the source code, but a summary of the more interesting design choices are included here:

* C++
* MPI functions for parallel processing
* A C++ Standard Library vector<bool> to hold markings for prime or non-prime for each number.
  + This data structure was chosen due to its specialization to store a bool as a bit instead of a byte automatically and thus take 1/8th of the memory that would have been taken by an array of char data types. This reduction in memory usage is extremely important given the large number of objects being stored in memory at any one time. Given more time to implement the solution, a faster, custom implementation would have been generated using a combination of bitmasking, malloc(), and the char data type.

## Data and Task Partitioning Among Processes

Due to the nature of such large N values and C++’s standard implementation of a 32-bit unsigned long integral value having a maximum value of 4294967295 (approximately 4.3 billion) the floating-point double type was used to track interval search ranges for each process. This use of double types (and subsequent casting to integral types) caused a slowdown in the run-time speed of the implementation. The following algorithms to partition tasks and data were used in an attempt to semi-uniformly distribute the tasks and data among processes, resulting in a semi-uniform task granularity.

Data was partitioned algorithmically each run:

* The master process (always process 0) received the search range and marked 0 and 1 as non-prime.
* Every other process received a an interval of width whose actual lower and upper bound values varied based on the process’ identification number returned by MPI\_Comm\_rank().

Tasks were partitioned algorithmically each run:

* The master process began searching from i=2 and broadcasted the current i to all processes including itself.
* Every other process received the number i from the master process.
* All processes worked from the least multiple of i within their interval and marked out all multiples of i.

## Data Collection and Reports

### Calculation of Average Timings

As P increased, the variance in the resulting run-time increased. Therefore, a greater number of runs with P=8 and P=16 were necessary to receive accurate results. Each (N,P) pair had its results recorded as follows:

Where minSum is a function that returns the sum of the minimum X numbers in the range R1..RM. For all cases in this data collection, the minimum 3 timings were averaged and recorded for each (N,P) pairing.

N varied as follows:

1. 32 million
2. 64 million
3. 100 million
4. 128 million
5. 500 million
6. 1 billion
7. 2 billion
8. 4 billion
9. 10 billion

P varied as follows:

1. 1
2. 2
3. 4
4. 8
5. 16

Only the portion of the program that looped iteratively to broadcast, receive, and mark out multiples of primes was timed. Instantiation, population, and output of prime numbers was not timed.

### Calculation of Speedup and Efficiency

The Speedup of a parallel program is the ratio of the time the program would take to complete in a serial implementation versus the time taken to complete in its parallel implementation. All figures reported as “Speedup” figures are calculated as such:

Where normally, is the amount of time in seconds that the fastest sequential implementation of the algorithm can produce. For this project, is equal to the time with P=1. is then the amount of time taken for each run with P > 1 (ie, P=2, 4, 8, or 16).

The efficiency of a parallel program indicates the percentage of time every process is active. Ideally this figure would be 1, but this is almost never the case. All figures reported as “Efficiency” figures are calculated as such:

Where is the same as described in the calculation of Speedup and P is the number of processes used.

### Use of Optimizations

Except where noted in the Data section, all timings collected were using a version of the Sieve that does not implement optimization 2 as described in the Implementation Specifics section. This is due to the instability of results when using the optimization for N >= 1 billion. A problem occurred where the runs would not complete in the allotted time with the optimization in effect.

# Data

## Table of Average Run-Times

The data in the following table shows the average amount of time, in seconds, taken to complete the Sieve for each (N,P) pair.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Average Run-Time | | | | | | |
|  | | P | | | | |
| **1** | **2** | **4** | **8** | **16** |
| N | **1.00E+08** | 32.09627 | 32.22537 | 10.4832 | 4.67103 | 1.913528 |
| **5.00E+08** | 130.2857 | 115.251 | 52.8774 | 15.9885 | 6.010497 |
| **1.00E+09** | 333.178 | 337.541 | 110.7139 | 32.1342 | 16.02173 |
| **2.00E+09** | 477.2133 | 520.2735 | 124.1547 | 63.7672 | 37.10343 |
| **4.00E+09** | 1148.888 | 1442.933 | 252.634 | 101.5771 | 70.2729 |
| **1.00E+10** |  |  |  |  | 170.601 |

These charts visualize the data in the Average Run-Time table. The vertical axis is the number of seconds for each run; the horizontal axis is the number of processes.

## Table of Speedup Ratios

Required for Part 6(a) and 7(a)

The data in the following table shows the speedup ratios for each (N,P) pair. See Calculation of Speedup and Efficiency in the Method section for calculation of these values. The bottom row shows the average of each column, that is, the average speedup across all tested N for each P.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Speedup Ratios | | | | | | |
|  | | **P** | | | | |
| **1** | **2** | **4** | **8** | **16** |
| N | **1.00E+08** | 1 | 0.995994 | 3.061686 | 6.871347 | 16.77334 |
| **5.00E+08** | 1 | 1.130451 | 2.46392 | 8.148711 | 21.67636 |
| **1.00E+09** | 1 | 0.987074 | 3.00936 | 10.36833 | 20.79538 |
| **2.00E+09** | 1 | 0.917236 | 3.8437 | 7.48368 | 12.8617 |
| **4.00E+09** | 1 | 0.796217 | 4.547638 | 11.3105 | 16.34895 |
|  | **1** | **0.965394** | **3.385261** | **8.836513** | **17.69115** |

## Table of Efficiency Ratios

Required for Part 6(b) and 7(b)

The data in the following table shows the efficiency ratios for each (N,P) pair. See calculation of Speedup and Efficiency in the Method section for calculation of these values. The bottom row shows the average of each column, that is, the average efficiency across all tested N for each P.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Efficiency Ratios | | | | | | |
|  | | **P** | | | | |
| **1** | **2** | **4** | **8** | **16** |
| N | **1.00E+08** | 1 | 0.497997 | 0.765421 | 0.858918 | 1.048334 |
| **5.00E+08** | 1 | 0.565226 | 0.61598 | 1.018589 | 1.354772 |
| **1.00E+09** | 1 | 0.493537 | 0.75234 | 1.296041 | 1.299711 |
| **2.00E+09** | 1 | 0.458618 | 0.960925 | 0.93546 | 0.803856 |
| **4.00E+09** | 1 | 0.398108 | 1.13691 | 1.413812 | 1.021809 |
|  | **1** | **0.482697** | **0.846315** | **1.104564** | **1.105697** |

## Completely optimized vs Non-Optimized Timings

Required for Part 5

The data in the following table show the amount of time, in seconds, taken to complete the Sieve with N=n and P=1 for both the optimized and non-optimized solutions. “Optimized” here is intended to reflect implementation of the second optimization listed in the Implementation Specifics section.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Optimized vs Non-Optimized | | | | |
|  |  | **Non-Optimized** | **Optimized** | **Speedup** |
| N | **32E+06** | 9.0763867 | 8.665873 | 1.047371 |
| **64E+06** | 19.358267 | 18.11533 | 1.068612 |
| **128E+06** | 39.879067 | 37.2522 | 1.070516 |

## Number of Primes in [0, 1 billion]

Required for Part 3(a)

As computed by the Sieve multiple times, the number of primes between 0 and 1,000,000,000 not including 0 or 1 is:

50847534 prime numbers

## First 20 Primes after 1 million

Required for Part 3(b)

As computed by the Sieve multiple times, the first 20 primes after 1,000,000 are:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| First 20 Primes after One Million | | | | |
| 1000003 | 1000033 | 1000037 | 1000039 | 1000081 |
| 1000099 | 1000117 | 1000121 | 1000133 | 1000151 |
| 1000159 | 1000171 | 1000183 | 1000187 | 1000193 |
| 1000199 | 1000211 | 1000213 | 1000231 | 1000249 |

## Best Run-Time for P=16 and N=10 billion

Required for Bonus 1

The best run-time achieved by this implementation of the Sieve for P=16 and N=1E10 was:

167.216 seconds

## Highest Prime Number Found

Required for Bonus 2

The highest prime number found is the largest prime number preceding 10 billion:

9999999967

# Analysis of Data

## Analysis of Average Run-Times

The run-times presented follow a clear pattern correlated to the increase of P from 1 to 2 to 4, etc. versus a fixed N. The graphs in the Average Run-Times section of the Data visualize this correlation. Generally speaking, as the number of processes increased, the run-time to completion decreased proportionally. Each doubling P almost halved the run-time for the Sieve.

The run-times were not exactly halved due to a number of factors:

* Doubled communication overhead for a doubled number of processes
* Consistent data partitioning as mentioned in the Method section
* Varied task granularity due to data and task partitioning as mentioned in the Method section

As stated, due to the above factors, the average runtimes for P=4, 8, and 16 vs P=1 decreased almost by a factor of 2 each doubling of P. However, for P=2 vs P=1, it is seen that for almost all N tested, the average run time for P=2 is *greater* than P=1. The explanation for this is theorized thus:

Due to the fashion the data is partitioned, the master process takes an extremely small set of the search space compared to the second process. This means that the second process is searching almost exactly the same space as the space the only process in P=1 would be searching. The task granularity is far from uniform. Normally this would not be a problem, but the added communication overhead of having to communicate with the master process slows the completion of the Sieve dramatically.

It can thus be seen that even though there is a doubling of the processes from P=1 to P=2, unlike with P=4, 8, or 16, the added complexity does not result in faster run-times.

## Analysis of Speedup Ratios

The speedup ratios reflect the timing ratios discussed in the Analysis of Average Run-Times section. They reflect the decrease in speed from P=1 to P=2, showing approximately a 4.5% speed decrease on average. When P>=4 however, the ratios reflect an almost linear speedup. For P=8 and P=16, the ratios show a superlinear speedup.

It is hypothesized that the reason for superlinear speedup in these ratios is inconsistency of the MST cluster and resulting P=1 runtimes were “slow” while resulting P=8 and P=16 runtimes were “fast” and that the combination of these two events caused an inflated ratio for P=8 and P=16.

The speedup ratios reflect almost exactly an ideal increase in speedup per the number of processes used for the algorithm.

## Analysis of Efficiency Ratios

The efficiency ratios reflect the variance in the speedup ratios. The superlinear speedup of the P=8 and P=16 runs shows a more than 100% efficiency ratio for P8 and P=16, which is theoretically not possible. The efficiency ratios for P=2 reflect that the master process for those runs is idle most of the time, since almost the entire search space is allocated to the slave process. These ratios show that the task and data partitioning for P=2 is extremely inefficient, but that the data and task partitioning for P > 2 is much more efficient.

# Appendix

<https://docs.google.com/leaf?id=0B_s5yMVHMC16YzI0ZDE4NjEtYzgwYy00NGZmLTkxMmYtMWIxOTVhZmU4YjUw&hl=en>

The above page contains:

* Project instruction document and scoring guide
* Source code
* Raw output files
* Excel tables organizing raw output
* This report