Parallel Implementation of Prime Factorization

Melissa Ivie, Shawn Wagner, Sriya Gorrepati

Parellel Programming Spring 2022

1 Abstract

Prime numbers are a key component in data encryption. This project utilizes multiple approaches to prime factorization and aims to implement them into parallel processing systems. The parallel implementations are contrasted with non-parallel implementations to show the performance improvements achieved.

2 Introduction

Prime Factorization has represented an interesting topic in mathematics for centuries but only in the modern age of security has it gained much traction for practical purposes. The use of prime numbers has become increasingly important in the data encryption/decryption world and working with large primes has become a keystone concept in many security methodologies.

At a simplified level, using parallel programming to find the prime values within a designated range could improve the performance and efficiency of base prime computations and enable better discovery for larger numbers. In this project we will compare a non parallel method for finding all the prime numbers within a range to a parallel implementation with various numbers of processors. Our thesis is that a parallel implementation of prime factorization will demonstrate an improved performance and efficiency when compared to a non parallel execution methods.

3 Background

An Optical Parallel System for Prime Factorization[1]

A system for prime factorization is developed with optical modulo operations in it. A two-dimensional parallel processing is employed to achieve large-scale information processing for modulo multiplication, which is one of the important elements in an algorithm for prime factorization.

This method is used for huge data processing. When huge prime factors are given, it has the capability to provide correct information.

New Prime Factorization Algorithm and Its Parallel Computing Strategy [2]

A new algorithm to factorize numbers and a new parallel strategy for prime factorization is presented. For big numbers, factorization is hard to get. This paper present a novel method which constructs suitable solution tree by modulo and prunes search branches by equations set and other methods such random, and we also present a parallel strategy by using huge number of nodes.

This is one of the parallel computing algorithm which is used in finding the prime factors.

Models And Trends in Parallel Programming[3]

This paper tells about all models and trends of the parallel programming. It is one of major factors behind the motivation in choosing the prime factorization method. we also use some of the tools and methods mentioned in this paper to prove our theory.

Extremality Of Prime Factorization[4]

This paper describes how prime factorization leads to optimization. Some of the algorithms in this paper are useful while computing the factorization methods.

Factorization of Large Integers on Massively Parallel Computers[5]

Though we cannot factorize large integers on a massive parallel computer in our project now, we found this topic very interesting to research on. In this topic we talk about is a 32-bit VLSI processor of the super-minicomputer range (106 integer operations per second). An NCUBE of order k has 2k nodes, k = 0,1,2 ... and one of order k + 1 is formed by connecting two cubes of order k at corresponding nodes. We have got to learn about quadratic sieve method as well. This procedure is still getting adapted to the hardware.

4 Approach

The overall approach for this project began with implementing a set of sequential approaches to prime factorization using several different algorithms to create a basis of comparison for the parallel applications. The simulation will run all implementations finding primes within the ranges 2 - 2^8, 2^8 - 2^{10}, 2^{10} - 2^{12}, 2^{12} - 2^{14}, 2^{14} - 2^{16}, 2^{16} - 2^{18}, and 2^{18} - 2^{20}, recording the execution time required to find the prime values at each tier. The parallel MPI programs will be executed with 2, 4, 8, 16, 32 and 64 processors to explore how the parallel implementation scales with an increasing number of nodes. The final average run time for each program within each integer tier will be recorded and compared to determine the potential effectiveness of the parallel prime decomposition methods.

4.1 Sequential Implementations

The base for this approach requires creating programs that decompose the prime values with different sequential algorithms so that we can determine the relative effectiveness of using parallel processing. The first serial method uses the Rho Algorithm for factorization. In this approach, we take a positive integer n and find divisors for it by testing all values between 2 and the $\sqrt{n}$. The algorithm for this follows the steps below:

1. Begin with a randomly generated x and c value. Find the y equal to x and set f(x) = x^2 + c
2. While the divisor is not found:
   (a) update the x fo f(x) modulo n
   (b) update the y to f(f(y))
   (c) Calculate the greatest common denominator of $|x - y|$ and n
   (d) if the greatest common denominator is not in unity
      i. if the greatest common denominator is n, repeat from step 2 with a different x, y and c
      ii. else the greatest common denominator is the answer

The next sequential approach follows the Wheel Factorization Method. In this algorithm, you begin with a small list of numbers called the basis which is usually the first few prime numbers. A 'wheel' list is then generated of the integers that are co-primes with all of the numbers in this basis. The Wheel method uses the following pseudo code:

1. For the number N to be prime or not:
   (a) bool xisPrime
      i. if x ¡ 2 return false
      ii. for N in 2,3,5, return false
      iii. for p = [0, $\sqrt{n}$] such that p=p+30
         A. for c in [p+7, p+11, p+13, p+17, p + 19, p +23, p+29, p+31]
         B. if c ¿ $\sqrt{n}$, break
         C. else if N % (c+p) = 0, return false

      return true

The final serial method uses Euler's Factorization, which is a sieve factorization like the Sieve of Eratosthenes. This algorithm works based on the principle that all of the numbers N which can be written as the sum of two powers in two different ways can be factored into two numbers. The implementation of this follows the below steps:

1. Find the sum of squares by iterating a loop for 1 to $\sqrt{n}$ because no factor exists between $\sqrt{n}$ and N other than N. They find the two pairs whose sum of squares is equal to N.

2. store the values is x,y,z,d

3. find the values of k, h, l and m using the formula use these values to find the factors.

4. check the pair where both of the numbers are even and divide them in half and find the factors.

4.2 Initial Parallel Implementation

This approach details the first parallel application to identify the primes within each given range. It utilizes the base algorithm for prime factorization as follows:

1. Iterate through a loop of all numbers in the given range, passing each value n to the isPrime function an adding all items to the primes list when isPrime returns true.

2. Determine the primeality for each n.

3. Iterate through a loop from $i = 2$ to $\sqrt{n}$ because no factor exists between $\sqrt{n}$ and N other than N.

4. Attempt to divide every n by each possible divisor i and return false if any divisor is found.

5. Print the list of all prime values within the given range.

This form is also referred to as the naive prime solution. It is parallized using $MPI\_Send()$ and $MPI\_Receive()$ commands. The total number of processors is determined and used to compute a delta that divides the total range of values into segments to be processed by each worker node. The calculated range segment is then sent to each processor where the above formula is utilized to compose a list of all primes within the given set. Once the end of the range is reached, the calculated primes are returned to the master processor at rank 0. After receiving the results from every node, process 0 reports the total number of primes found and calculates the execution duration.

4.3 Sieve of Eratosthenes

The Sieve of Eratosthenes is a method for identifying prime numbers. The method functions by listing the numbers from 2 to the desired range, n. Examining the values in the number list starting from the first value, p, and proceeding sequentially through the numbers until $p >= \sqrt{n}$, all multiples of p are eliminated from the number list. Once p has executed all iterations the number list will be comprised solely of the prime numbers within the range.

The algorithm described in an article by Sorenson[6] initializes an array of length n and marks the composite numbers as 1 and primes as 0. The pseudocode is given as follows:

Initialize:

$s[1] := 0;$

for $i := 2$ to n do $s[i] := 1;$

Main Loop:

$p := 2;$

while $p^2 \leq n;$ do

Remove multiples:

for $f := p$ to $\lfloor n/p \rfloor$ do $s[pf] := 0;$

Find the next prime:

repeat $p := p + 1$ until $s[p] = 1;$

end while;

A parallel implementation of this sieve was incorporated using an approach described in a paper titled "Parallelization of Sieve of Eratosthenes"[7]. The sieve is parallelized through the distribution of the number array between the designated number of processes, x. Each process is assigned $\frac{n}{x}$ numbers to sieve. For the purpose of creating an easily usable simulation, x and n are restricted to powers of 2, where n must be greater than x. Having powers of 2 for both x and n will always allow even shares with an even number of elements and is useful for this method as the initial p is 2. A limitation to this method is that there are memory allocation issues if n is too large. In the process of running simulations and results gathering, we were only able to reach an upper limit of $n = 2^{20}$. Powers larger than 20 caused the simulation to shutdown.

5 Simulations

In order to maintain a consistency across the investigated and implemented methods, our simulations were performed using the following criteria:

- Ranges tested in powers of 2:

  $2^8, 2^{10}, 2^{12}, 2^{14}, 2^{16}, 2^{18}, 2^{20}$

- Number of processors used in powers of 2:

  2, 4, 6, 8, 16, 32, 64

- Simulation results are a measurement of the elapsed execution time in microseconds (1000000 * seconds)

- A baseline result is measured using a singular processor execution
- Each combination of range and number of processors was tested 10 times recording the average duration as the result

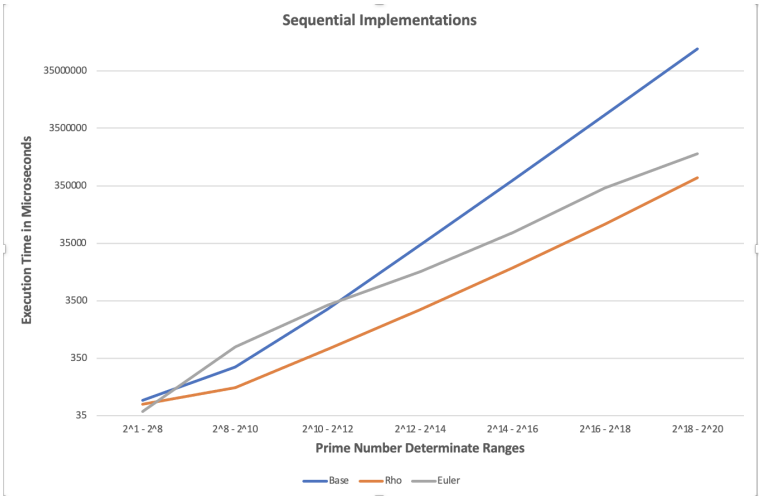
6 Results

The following results encapsulate the outcome from the simulation detailed above for each of the tested approaches. Overall findings indicate that Rho is the most efficient of the serial methodologies and that parallel implementations are more effective for higher integer values but less effective for lower digit ranges.

6.1 Sequential Algorithms

The initial results detailed the outcome for linear implementations. The base prime method, Euler method and Rho method were tested using the simulation conditions to produce the following figure. Note that the result graphs are pictured using a logarithmic vertical access for a clear depiction of the values and tier differences.

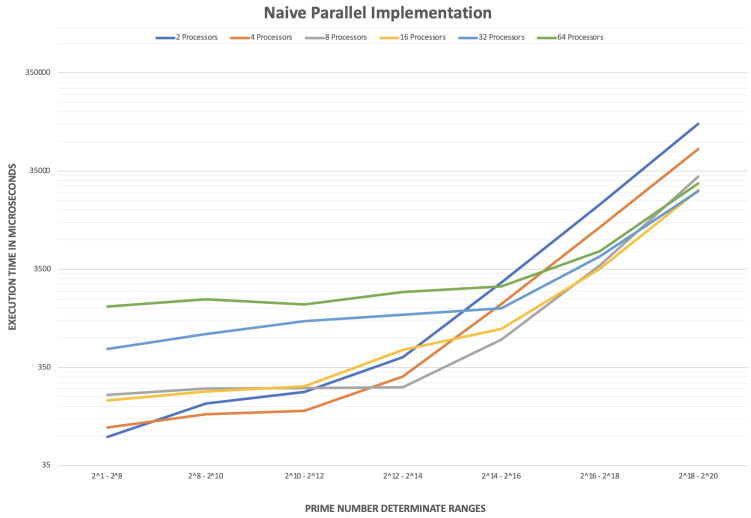
The efficiency of these sequential algorithms can be contrasted to determine relative performance and identify the best serial method basis for parallel comparison. The image below compares each of the three above algorithms and displays their execution duration in microseconds. The Rho method displays the most consistent highest efficiency results.



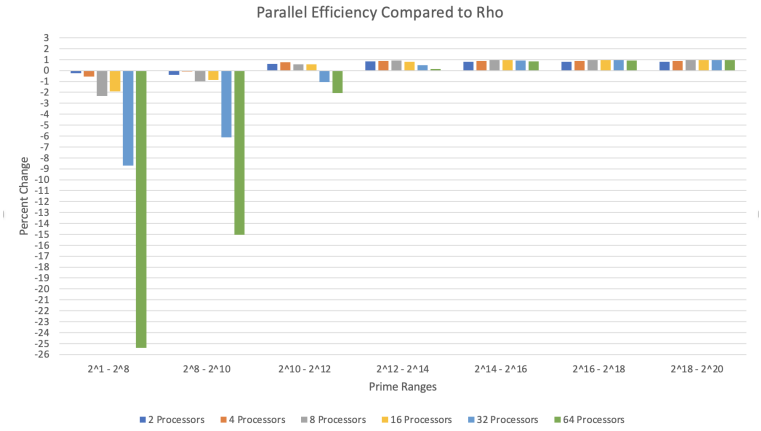
6.2 Parallel Naive Implementation

This Parallel method was tested with 2, 4, 8, 16, 32, and 64 processors. The distribution utilizes a master slave approach with the Naive factorization method,

similar to the base sequential method aside from its parallel execution. Processors communicate with MPI send and receive commands and generate the following execution results.

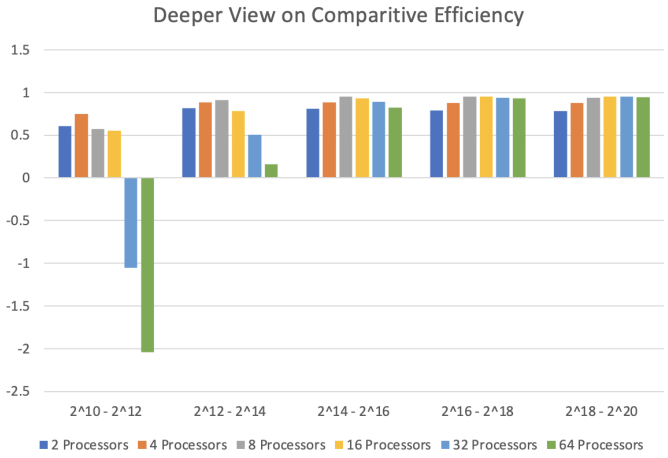


This figure can be compared to the serial simulations to note several key differences. First, the slope of each line is less sharp, indicating a smaller degree of execution change across each range tier. Second, the results display a much lower initial efficiency and a high number of processors do not always result in a better duration time. We can further examine the improvement of this parallel application by measuring these execution times as an percent change of the Rho algorithm results, pictured in the image below.



The zero baseline represents a 0% change in efficiency, meaning that any measurements at this line display the same level of effectiveness as the serial Rho

algorithm, the most effective of the non-parallel methods. All bars below this line are less effective and all above the line demonstrate improved effectiveness. It can be noted that at the $2^1 - 2^8$ and the $2^8 - 2^{10}$ ranges, the parallel implementations with each processor number takes much longer than the serial Rho algorithm. So for these sets of values, the parallel method is much less productive. This is likely because of the overhead required for dividing out range subsets and communication between processors. A closer view of the $2^{10} - 2^{12}$ range and above can been viewed in the following figure.

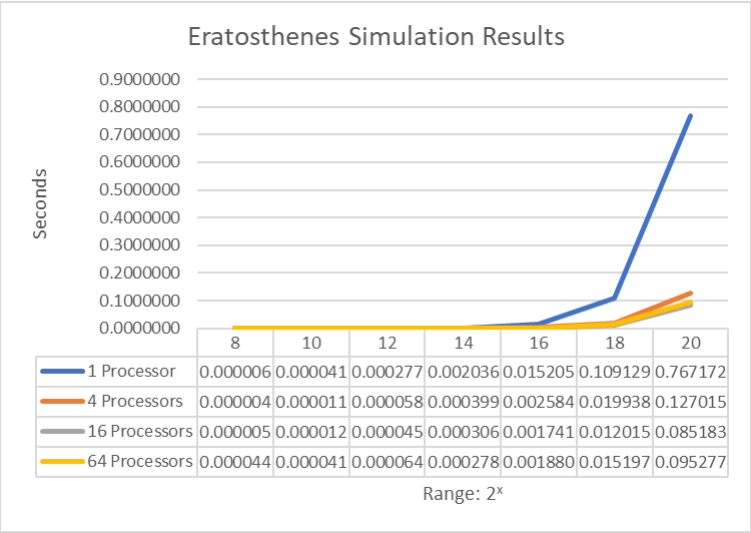


These data ranges show a dramatically increased level of efficiency, indicating that the parallel implementation begins demonstrating better durations when values are above four digits. All values close to the "1" line indicate a near 100% efficiency improvement from the Rho algorithm.

It is also interesting to note that utilizing a higher number of processors, particularly the the 32 and 64 processor levels, does not provide increased effectiveness until the $2^{16} - 2^{18}$ range. This trend can also been seen with the 8 and 16 node levels in the 2^{12} and 2^{14} ranges. The processing time saved by using multiple threads is overcome by the overhead in these groups and does not result in lower execution times until the higher digit tiers. It is possible that further testing at even greater number levels would reveal an even more distinct trend in the differences caused by larger quantities of processors.

6.3 Sieve of Eratosthenes

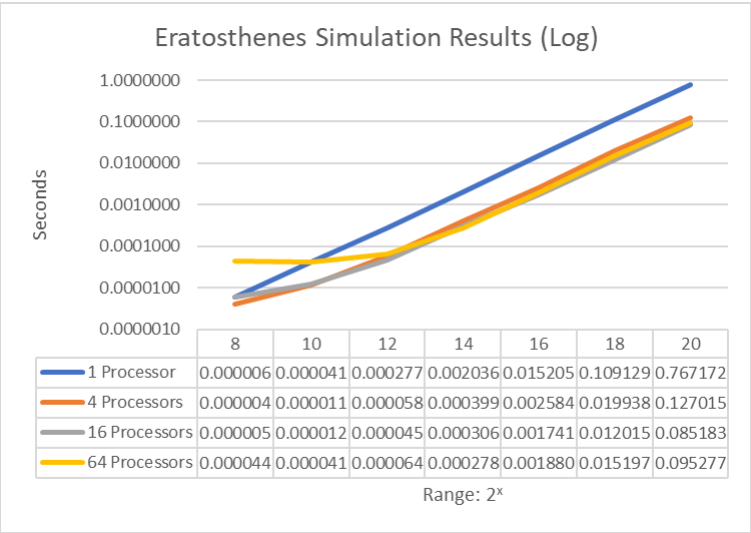
The results for this method are produced from simulations conducted using 1, 4, 16, and 64 processors and represent the execution duration for the specific ranges.



The results show that the parallel implementations at lower ranges do not yield significant improvements to the overall run time. However, at larger ranges there is marked improvement with the largest tested range for the singular process taking approximately 6 times as long as the parallel executions.

Comparing the parallel simulations together, we can see that the results are quite similar. At the lower ranges, the higher number of processes takes more time due to distributing of numbers over a larger number, but once the larger ranges are reached, the execution times are close in efficiency.

The method employed shows that the execution is primarily aided through the parallel execution, but do show that the scaling of the execution is quite consistent regardless to the number of processors.



7 Conclusions

The efficiency and efficacy of parallel solutions depends on the parameters to which they are employed. Given numbers on the lower end of the number line, parallel implementations may yield little to no benefit and may even result in a poorer result. However, as n increases, the results trend in the way of increasing improvements as reflected in both methods. With respect to the functionality of prime numbers in the realm of data encryption, parallel implementations of prime factorization prove to be more productive and time efficient.

8 Future Work

In future, we want to implement an optimal parallel system method. It is a system for prime factorization with optical modulo operations is developed and demonstrated. In the system, two-dimensional parallel processing is employed to achieve large-scale information processing for modulo multiplication, which is one of the important elements in an algorithm for prime factorization. We also want to try to factorize the large integers on massively parallel computers. It uses the quadratic sieve method. This is a long procedure and it requires hardware that satisfies the compatibility as well. Thus it could be one of the most impactful techniques to find the prime factors.

References

[1] K. Nitta, N. Katsuta, and O. Matoba, "An optical parallel system for prime factorization," *Japanese Journal of Applied Physics*, vol. 48, no. 9S2, p. 09LA02, 2009.

[2] L. Huang, "New prime factorization algorithm and its parallel computing strategy," in *2015 1st International Conference on Information Technologies in Education and Learning (icitel-15)*. Atlantis Press, 2016, pp. 118–121.

[3] C. I. Saidu, A. Obiniyi, and P. O. Ogedebe, "Overview of trends leading to parallel computing and parallel programming," *British Journal of Mathematics & Computer Science*, vol. 7, no. 1, p. 40, 2015.

[4] K. Rao, "Extremality of prime factorization," *Resonance*, vol. 26, no. 12, pp. 1643–1648, 2021.

[5] J. A. Davis and D. B. Holdridge, "Factorization of large integers on a massively parallel computer," in *Workshop on the Theory and Application of of Cryptographic Techniques*. Springer, 1988, pp. 235–243.

[6] J. Sorenson, *An introduction to prime number sieves*, 01 1990.

[7] G. Bhat, N. Kini, S. Ray, S. Prabhu, and G. Hegde, "Parallelization of sieve of eratosthenes," *International Journal of Scientific Research in Computer Science Applications and Management Studies*, vol. 3, 01 2014.