Parallelization of Sieve of Eratosthenes

CS-301
High Performance Computing

Submitted by :

Husain Hirani(201701448)

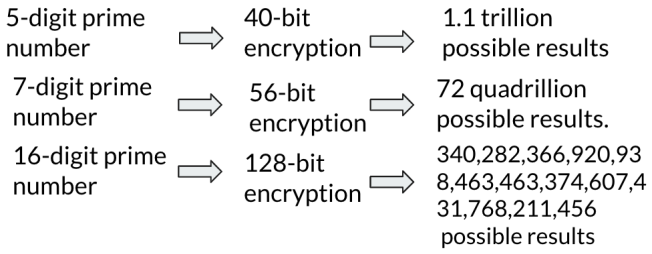
Kishan Kavathiya(201701155)

DA-IICT
GANDHINAGAR

Contents

1 Introduction

Prime Calculation plays nowadays a crucial role on secure encryption. So finding prime numbers efficiently is so important for security and to improve efficiency of various encryption/decryption algorithm.



So larger the prime numbers, better the encryption.

In mathematics no formula has been devised to generate a sequence of prime numbers. Sieve of Eratosthenes is a mathematical algorithm to generate the prime numbers in given range. The way this method works is to find the lowest prime numbers and then strike out all the multiples of it in the remainder of the list. The next lowest prime is then picked up and striking out of all its multiples is continued as previous.

To improve the speed of generating primes, we have used three different approaches, including using redundant computations to reduce process computation time and rearranging the order of computations to increase the cache hit rate.

2 Serial Implementation

Sieve of Eratosthenes is one of the most efficient ways to find all prime numbers smaller than N when N $\leq 10^{10}$ or so.

2.1 Base Algorithm

So in serial implementation, we first initialize a boolean array of size (n+1) with value **true**. Now iterate through all numbers from 1 to $\sqrt{n}$. And for each of these numbers, if the value of boolean array is **true** then we change the value of boolean array to **false** for all the multiples of this number (as these are composite numbers). After performing the above operation, a number is prime only if the value of boolean array for that number is **true**.

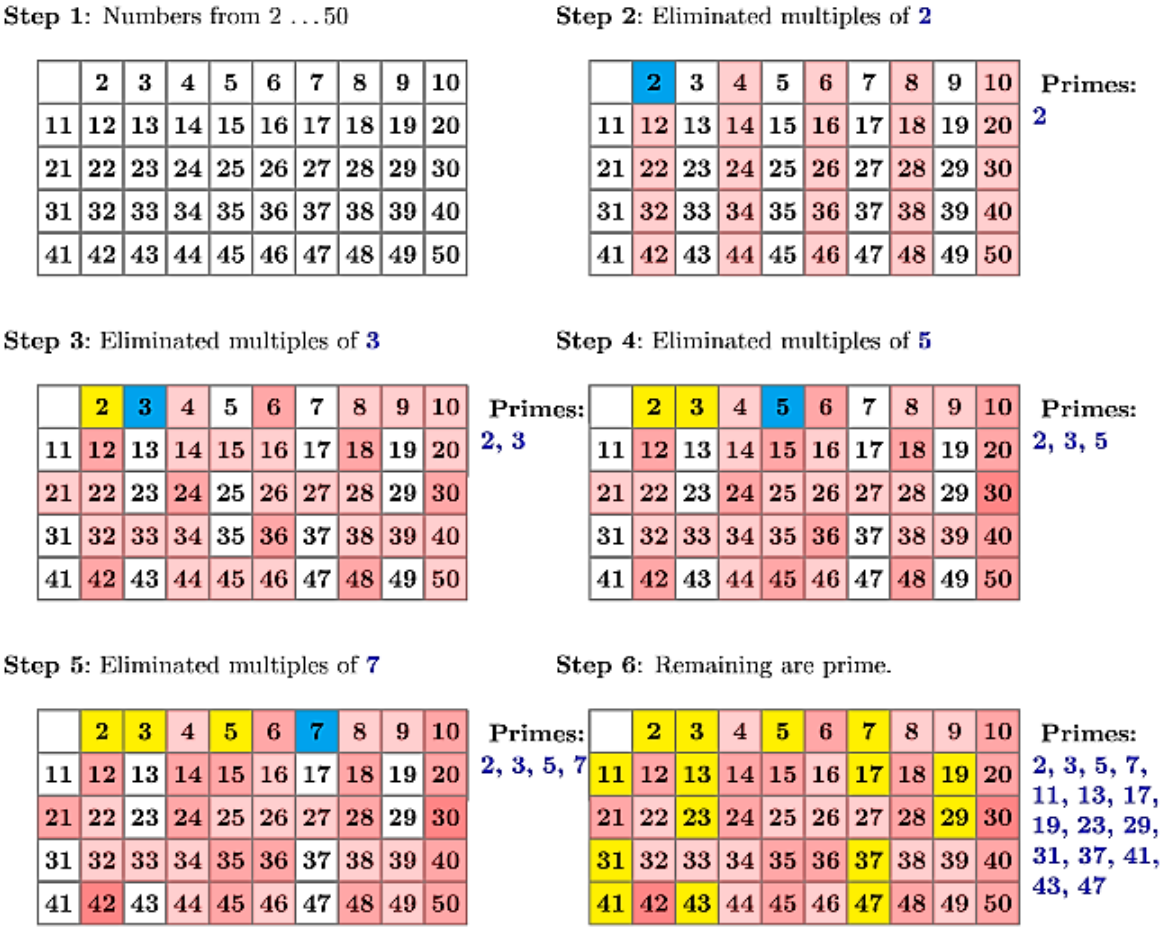
Pseudo-code for sieve of Eratosthenes' serial implementation:

1. Create a boolean array of consecutive integers from 2 to n.
2. Initially, let k equal to 2, which is the first prime number
3. Starting from k^2, count up in increments of k and mark each of this number with value **false**. These numbers will be k(k+1),k(k+2),k(k+3),etc...
4. Find the first number greater than k in list that is **true** and this number should be less than $\sqrt{N}$. If no such number exists, stop. Otherwise, let k now equal this number (which is the next prime), and repeat from step 3.

After using the above algorithm, all the numbers which has the value true are prime numbers.

We have used different optimization techniques to further optimize this algorithm and then parallelized the algorithm with the use of Open MP. And for the efficient parallelization of the algorithm we have used a whole new approach.

Example of serial implementation for N=50:



2.2 Optimization strategy-1

We know that the only even prime number is 2. Other than that all the prime numbers are odd. So we can **save 50**% of computation just by not traversing all of the even numbers. So just by doing this minor modification we can achieve a **speedup** of around 2.

2.3 Optimization strategy-2

Prime sieve uses lot of space. In case of 1 bit per number, we'd need 1GB to store sieve for N=10^9. So what we can do is, since we are not traversing the even numbers, we don't need to store their values in boolean array. So we can even save **50**% of memory by not storing even numbers.

To do that we have to map odd numbers to integers. i.e map k $\longrightarrow \frac{k}{2}$, where k =3,5,7,....

2.4 Time Complexity Analysis

- If we assume that the time taken to mark a composite number is constant, then the number of times the loop runs is equal to $\frac{N}{2} + \frac{N}{3} + \frac{N}{5} + ... + t$, where t is the highest prime number less than N.
- So the above equation can be written as $N * (\frac{1}{2} + \frac{1}{3} + \frac{1}{5} + \frac{1}{7} + ... + t)$
- Harmonic progression of the sum of prime is $\approx \ln(\ln(N))$. This can be proved using harmonic series and Taylor series expansion.

Time Complexity :- $O(N * \ln(\ln(N)))$
Space Complexity :- $O(N)$

- After using optimization strategy 1-2, we have to only look for the odd numbers. So, the computation is reduced to half.
  So, Time Complexity(after using optimization strategy 1 and 2) :- $O((N/2)*\ln(\ln(N/2)))$
- To further reduce the time complexity, we have parallelized the algorithm with the use of open MP.

3 Scope Of Parallelism

In the sieve of Eratosthenes method to find the prime numbers table, once we find the the prime number k, we set k*n as non-prime for each n. So if we know that two elements(i,j) are prime, then we can mark the table in parallel as i does not require any information from j and vice-versa. The same thing we can do with n prime numbers.

However, finding if a number is prime depends on last calculations! Thus, a **barrier** is needed between marking numbers as non-prime and finding next prime numbers to work with.

repeat until finished filing the boolean array

1. find k prime numbers (serially)
2. fill the boolean array for theses k numbers (parallely)
3. after all threads finishes above step (barrier) return to step 1.

As we have discussed above, memory usage is also a main problem.

After all even faster RAM is hundreds of times slower than L! cache. So it's better if we can fit the data to 32KB or 256KB instead of Giga Bytes.

That is why we have used a whole new approach called Blockwise data decomposition to parallelize the algorithm.

4 Parallelization strategies

Sieve is accessed pretty randomly and it is not automatically loaded to cache so we would get continuous cache misses. So we use **blockwise strategy** to overcome all of these difficulties discussed above.

In this approach we divide the range 1 to N into smaller blocks. And only for this blocks we perform sieve of Eratosthenes. Now to parallelize this, we use data parallel approach that is different processor elements perform the same operation on different data sets.

Each processor will be responsible for a segment of the array (whose size will be equal to block size). All the processors perform the same operation(i.e. strikes off multiples of some prime) on its own segment of data.

```
#pragma omp parallel for reduction(+:num_of_primes2)
for (i = 2; i <= n; i+=blocksize)
{
    long long int last = i + blocksize;
    if (last > n)
    {
        last=n;
    }
    num_of_primes2 += eratosthenesOddSingleBlock(i,last);
}
```

We have used reduction to count the total number of primes in given range. Parallelization of the serial approach(pseudo code on page 2) using above method:

Step 1 is simple to translate. Instead of single process creating the entire list of natural numbers, each process in the parallel program will create a it's portion of the list.

Now, we can't parallelize the step 2 of serial implementation because every process need to know the value of k to mark the multiples of k in it's region.

To parallelize step 3, each process is responsible for marking the multiples of k in between k^2 and n. So we have to do a little bit of calculation to determine the first location of k but after that it becomes easy as we only have to mark every kth element.

To further reduce the time we have also combined optimization 1 and 2 with this approach. We have also used divisibility tests, skipping all the numbers which are divisible by (3,5,7,11 and 13).

4.1 Time Complexity

Suppose block size is b, number of processors are p and problem size is N then $b * ln(ln(b))$ time is required to compute for each block. And there are $\approx \frac{N}{b}$ blocks and we distribute it among p processors. So,

$$\textbf{Time Complexity} = \left\lceil \frac{n}{b} \right\rceil * \frac{b}{p} * ln(ln(N))$$

5 Results & Quantitative Analysis

5.1 Hardware Details

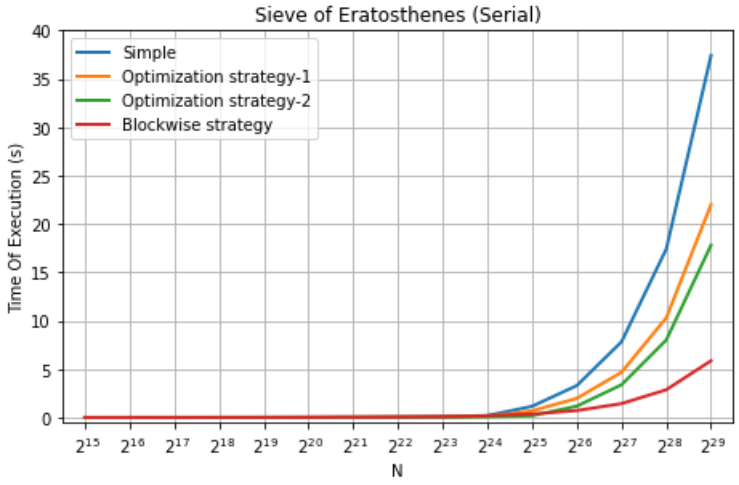
Architecture	x86_64
CPU op-mode(s)	32-bit, 64-bit
Byte Order	Little Endian
CPU	16
On-line CPU(s) list	0-15
Thread(s) per core	1
Core(s) per socket	8
Socket(s)	2
Model Name	Intel(R) Xeon(R) CPU E-52640 v3 @ 2.60GHz
L1d cache	32K
L1i cache	32K
L2 cache	256K
L3 cache	20480K

5.2 Curve Based Analysis

1. For the serial implementation of the sieve of Eratosthenes, as the N increases time of execution will increase exponentially for all the strategies (i.e. Simple,Optimization 1 and 2, Block wise).

   Optimization strategy 1 and 2 are nearly 2 times faster than base algorithm because of lesser computation. Optimization strategy-2 is faster compare to 1 because of lesser memory access.

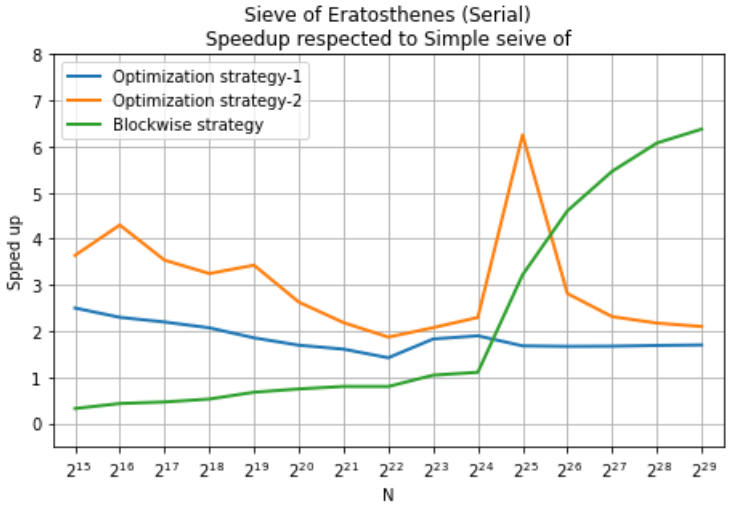
   Block wise serial approach is serial version of the strategy discussed in point-4. We observed that even serial version of block wise approach reduces so much execution time. This is because of lesser cache misses as we only have to worry about small blocks which can be stored in L2 or L3 cache. And that would result in lesser cache misses. We have used block size=$\sqrt{N}$.



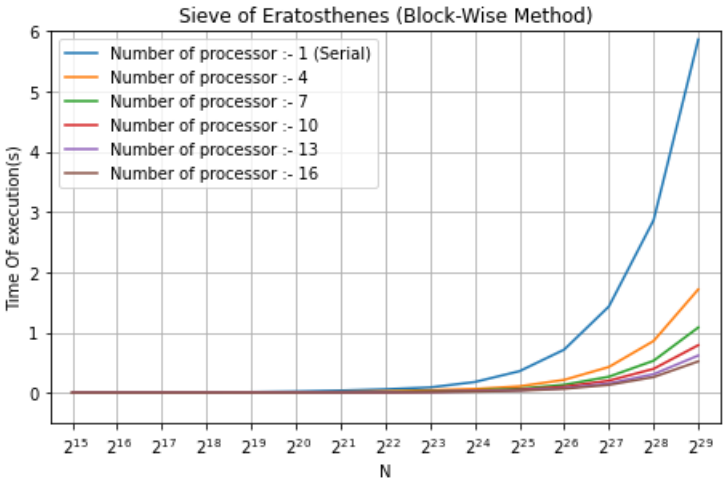
2. Figure below is a graph of speedup vs N for different serial optimization strategies, We can see that block wise strategy has less speedup than other two strategies for N<2^{25}. This is because of more computation is needed in block wise strategy. But after that we observed tremendous amount of increase in speedup of block wise strategy.

   We observed that optimization strategy-2 has speedup ≈ 6 around $N = 2^{25}$. This is because of lesser cache misses. So we can say that cache line size is more than the required memory in that case.

   We also observed sudden increase in speedup of block wise serial strategy after $N = 2^{24}$. That is because of bigger L3 cache size(i.e. 20480 K). That results in lesser memory access compare to other strategies.



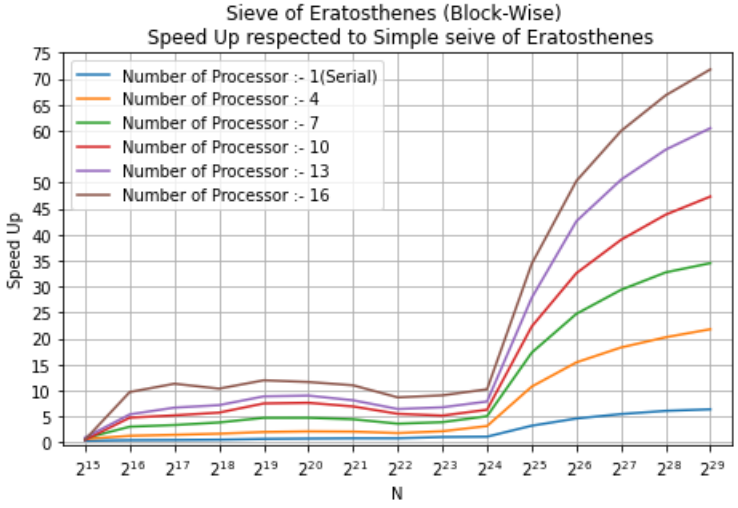
3. The graph of execution time of **parallel** block wise implementation of sieve of Eratosthenes vs N is shown below. The execution time is decreased compare to the graph shown in point 1 which shows that we have efficiently parallelized the algorithm. We observed that as number of processor increases the execution time decreases for same problem size N. So we were able to find prime numbers less than 10^9 in less than **1 second** by using 16 processors.
   We have used blocksize = $\sqrt{N}$.

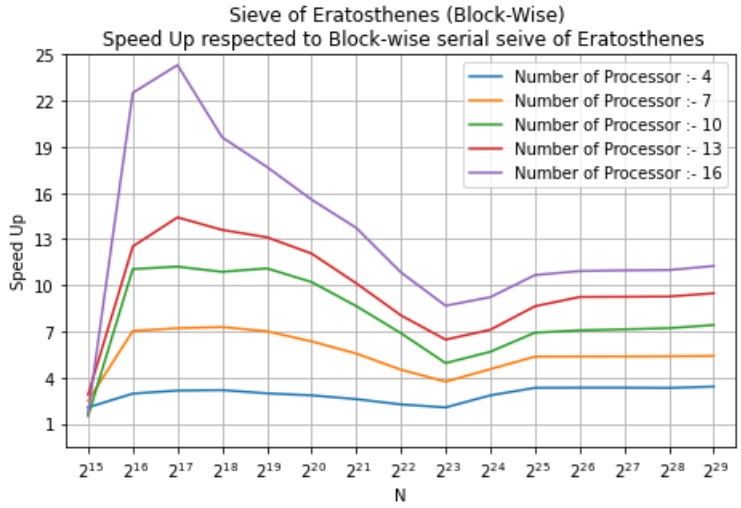


4. Following are two different graphs for speedup. The first graph is of speedup with respect to base implementation of algorithm and the second graph is of speedup with respect to block wise serial implementation of the algorithm.

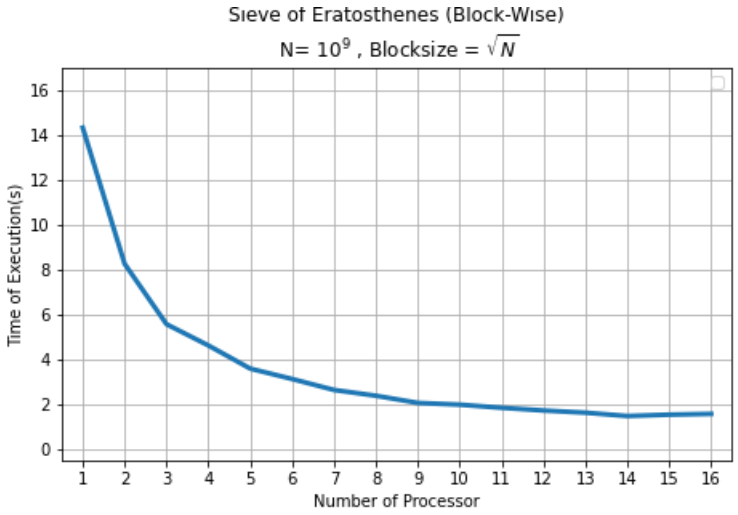
   From the first graph of speedup, we observed that speedup increases by very large amount when problem size is $> 2^{24}$. We achieved highest speedup of around **70** by using 16 processors.

   From the second graph of speedup, we can observe that speedup increases for smaller problem sizes, but when problem size becomes greater than 2^{17} it starts decreasing. So in this case we achieve the maximum speedup of 25. We were still able to achieve the speedup of around **11** for larger problem sizes. The increase in speedup for smaller N is because, size of L2 cache is 256K. So it can hold numbers up to 2^{16}. That results in faster memory access hence, increase in speedup for smaller N.





5. Figure below shows decrease in execution time as we increase the number of processors for constant problem size of 10^9 and block size of $\sqrt{N}$.



6. Useful Results:- This table displays number of primes in given range ,the maximum prime in the given range and execution time to calculate it.

Range(N)	Number Of primes	Max Prime	Execution time(s)
1000	168	997	0.00008
10000	1229	9973	0.0001
100000	9592	99991	0.0011
1000000	78498	999983	0.0023
10000000	664579	9999991	0.024
100000000	5761455	99999989	0.11
1000000000	50847534	999999937	0.8
2000000000	98222287	1999999973	1.81
3000000000	144449537	2999999929	2.82
4000000000	189966588	3999999979	3.83
5000000000	234954223	4999999937	5.32
10000000000	455052511	9999999967	9.43
20000000000	882206716	19999999967	19.33
50000000000	2119654578	49999999967	57.47
100000000000	4118054813	99999999977	122.51
500000000000	19308136142	499999999979	1195.9
1000000000000	37607912018	999999999989	1391.67

Providing a good parallel algorithm for sieve of Eratosthenes is tricky as output might differ is sufficient blocksize is not used. Our implementation provides acceptable results, and reaches maximum efficiency with a small number of processors and a sufficiently large N.

6 Further Improvements

Though the code here is parallelized using openMP (for shared memory architecture). Same can be done using MPI for distributed memory architecture.In MPI more broadcast operations is necessary. So we also have to compute communication time. MPI will be trade-off between communication overhead (Time Complexity) vs memory consumption.

7 References

- https://www.geeksforgeeks.org/sieve-of-eratosthenes/
- https://cs.ipb.ac.id/~auriza/parallel/book/Quinn.%202003.%20Parallel%20Programming%20in%20C%20with%20MPI%20and%20OpenMP.pdf