1. Intro to Prime Number
2. Sequential Sieve Background
3. Parallel Sieve Implementation
4. Results and Observations
5. Goals
Sequential Algorithm

Definition:

```python
def FindPrime(n):
    prime = [True for i in range(n+1)]
    for i in range(2,n+1):
        for j in range(2,i):
            if i%j==0:
                prime[i]=False
                break
    prime[0] = False
    prime[1] = False
    for i in range(2,n+1):
        print(i) if prime[i] == True else None
```

Time complexity: $O(n^2)$

- The prime number is a positive integer greater than 1 that has exactly two factors, 1 and the number itself.
- First few prime numbers are 2, 3, 5, 7, 11, 13, 17, 19, 23
- Except for 2, which is the smallest prime number and the only even prime number, all prime numbers are odd numbers.
- Every prime number can be represented in form of $6n + 1$ or $6n – 1$ except the prime numbers 2 and 3, where $n$ is any natural number.
The Sieve of Eratosthenes is a method used to find prime numbers. Prime numbers are important in modern encryption algorithms like sha256 that keep our digital transactions safe. Public-key cryptography also uses prime numbers to create specialized keys. The Sieve is also used in mathematics, abstract algebra, and elementary geometry to study shapes that reflect prime numbers. Biologists use the Sieve to model population growth, and composers use prime numbers to create metrical music. Olivier Messiaen, a French composer, used prime numbers to create unique rhythms in his music pieces.

Sieve of Eratosthenes
Sieve Simulation
**Sequential Sieve Algorithm**

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>14</td>
<td>15</td>
<td>16</td>
<td>17</td>
<td>18</td>
<td></td>
</tr>
<tr>
<td>19</td>
<td>20</td>
<td>21</td>
<td>22</td>
<td>23</td>
<td>24</td>
<td></td>
</tr>
<tr>
<td>23</td>
<td>26</td>
<td>27</td>
<td>28</td>
<td>29</td>
<td>30</td>
<td></td>
</tr>
<tr>
<td>31</td>
<td>32</td>
<td>33</td>
<td>34</td>
<td>35</td>
<td>36</td>
<td></td>
</tr>
</tbody>
</table>

Time complexity: $O(n \cdot \log(\log(n)))$

---

Pseudo code

```plaintext
find primes up to N
For all numbers a : from 2 to sqrt(n)
    IF a is unmarked THEN
        a is prime
        For all multiples of a (a < n)
            mark multiples of a as composite
All unmarked numbers are prime!
```
Parallel Sieve Implementation

- Split the array of length n between processors p each of size n/p if extra element is there, adjust in the last processor.

- Mark all even numbers as non-prime in each processor in parallel.

- Broadcast the minimum prime number in process 0 to other processes.

- Cancel out the multiples in process 0 and the other processes in parallel.

- After the primes are found in each process combine the result recursively.
Recursive Halving
Broadcasting at lower level
Broadcasting at terminal nodes

- Process 0 will send all the primes till sqrt(n) to all processes.
- Other processes will receive the prime and cancel the multiples in their range.
- Process 0 will also cancel the multiples.

```c
if (processId == 0)
{
    MPI_Request send_request;
    // Long int broadcast next prime;
    for (c = low; c <= sqrt(n); c++)
    {
        long int prime = -1;
        // If the number is unmarked
        if (marked[c - low] == 1 & c != 2)
        {
            prime = c;
            for (int j = c + 1; j <= high; j++)
            {
                if (j % prime == 0)
                {
                    // printf(" div \ld\n", prime);
                    marked[j] = low = 0;
                    // printf(" for rank 0 j = \ld, low = %ld, prime = %ld\n", j, low, prime);
                }
            }
            for (int i = 1; i < numberOfProcesses; i++)
            {
                // printf(" Broadcasting marked\ld\ld\n", marked[low - low], prime);
                int tag = (int)c;
                // MPI_Send-sendPrime, 1, MPI_COMM_WORLD, &send_request);
                MPI_Send(c, 1, MPI_COMM_WORLD, &send_request);
                // printf(" lol \ld\n", next_prime);
            }
        }
    }
}
```
Initial failed attempt for Broadcasting

- Process 0 will send all the primes till $\sqrt{n}$ to all processes using MPI_Isend.
- Other processes will receive the prime using MPI_Irecv and cancel the multiples in their range.
- The receive buffer is is getting resolved at different times in each processor causing faulty results.

```c
while (counter <= sqrt(N)) {
    MPI_Request recv_request;
    int ifResolved;
    MPI_Status recv_status;
    long int next_prime = -1;
    // printf(" Recv here %ld, process=%d\n", next_prime, processId);
    int tag = counter;
    MPI_Irecv(&next_prime, 1, MPI_LONG, 0, tag, MPI_COMM_WORLD, &recv_request);
    // MPI_RECV(&next_prime, 1, MPI_LONG, 0, tag, MPI_COMM_WORLD, &status);
    // printf(" Recv here after i recv %ld, process=%d\n", next_prime, processId);
    sleep(1);
    // do:
    // MPI_Test(&recv_request, &ifResolved, &recv_status);
    // while(!ifResolved);
    // printf(" Recv 2 %ld, process=%d\n", next_prime, processId);
    if (ifResolved) {
        if (next_prime == -1) {
            // MPI_RECV(&next_prime, 1, MPI_LONG, 0, 0, MPI_COMM_WORLD, MPI_STATUS);
            // printf(" Recv %ld\n", next_prime);
            for (c = low; c <= high; c++) {
                // if the number is unmarked */
                if (c % next_prime == 0) {
                    marked2[c - low] = 0;
                    // printf("Prime finding next_prime marked2 [%d-low] marked2 = %d\n", c - low, marked2[c-low]);
                }
            }
        } else |
        for (int i = 1; i < noOfProcesses; i++) {
            // printf(" Broadcasting marked=%ld prime=%ld\n", marked2[c - low], prime);
            // printf(" %ld %ld, marked2[c-low], c);
            // int tag = (int)c;
            MPI_Isend(&prime, 1, MPI_LONG, i, tag, MPI_COMM_WORLD, &send_request);
            // MPI_Send(&prime, 1, MPI_LONG, i, tag, MPI_COMM_WORLD);
            // printf(" lol %ld\n", next_prime);
        }
    } else {
        // printf(" Process id %ld\n", processId);
        int counter = 1;
        while (counter <= sqrt(N)) {
            MPI_Request recv_request;
            int ifResolved;
            MPI_Status recv_status;
            long int next_prime = -1;
            // printf(" Recv here %ld, process=%d\n", next_prime, processId);
            int tag = counter;
            MPI_Irecv(&next_prime, 1, MPI_LONG, 0, tag, MPI_COMM_WORLD, &recv_request);
            // MPI_RECV(&next_prime, 1, MPI_LONG, 0, tag, MPI_COMM_WORLD, &status);
            // printf(" Recv here after i recv %ld, process=%d\n", next_prime, processId);
            sleep(1);
        }
    }
}
```
Parallel Stitch Step

PE 0

1, 2, 3, 4

PE 1

5, 6, 7, 8

PE 2

9, 10, 11, 12

PE 3

13, 14, 15, 16

PE 3

1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16

PE 3

9, 10, 11, 12, 13, 14, 15, 16

PE 1

1, 2, 3, 4, 5, 6, 7, 8
## Result parallel

1 core per Node

<table>
<thead>
<tr>
<th>Processors</th>
<th>Time in sec</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>733.451</td>
</tr>
<tr>
<td>4</td>
<td>445.246</td>
</tr>
<tr>
<td>8</td>
<td>285.531</td>
</tr>
<tr>
<td>16</td>
<td>129.095</td>
</tr>
<tr>
<td>32</td>
<td>107.378</td>
</tr>
<tr>
<td>64</td>
<td>165.498</td>
</tr>
<tr>
<td>128</td>
<td>385.445</td>
</tr>
</tbody>
</table>

Communication overtakes computation
**IB Vs TCP|IP Network**

1 core per Node

| Processors | IB (Time in sec) | TCP|IP (Time in sec) |
|------------|------------------|-------------------|
| 2          | 54.011           | 66.297            |
| 4          | 38.485           | 72.850            |
| 8          | 36.957           | 61.447            |
| 16         | 26.384           | 104.279           |
| 32         | 23.527           | 127.930           |
| 64         | 33.528           | 197.186           |
| 128        | 49.977           | 329.153           |
Speed-Up

1 core per Node  Input size = $10^8$

<table>
<thead>
<tr>
<th>Processors</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1.926</td>
</tr>
<tr>
<td>4</td>
<td>3.173</td>
</tr>
<tr>
<td>8</td>
<td>4.948</td>
</tr>
<tr>
<td>16</td>
<td>10.944</td>
</tr>
<tr>
<td>32</td>
<td>13.157</td>
</tr>
<tr>
<td>64</td>
<td>8.537</td>
</tr>
<tr>
<td>128</td>
<td>3.66</td>
</tr>
</tbody>
</table>

**Speedup**

$$\text{Speedup} = \frac{T_{\text{Seq}}}{T_{\text{parallel}}} \quad T_{\text{Seq}} = 1412.8$$

**Graph**

![Graph showing speedup vs. number of processors](image-url)
Scaled Result (Gustafson’s law)

<table>
<thead>
<tr>
<th>Processors</th>
<th>Time in sec</th>
<th>Input size</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1376.044</td>
<td>10^4</td>
</tr>
<tr>
<td>2</td>
<td>750.429</td>
<td>2*10^4</td>
</tr>
<tr>
<td>4</td>
<td>401.852</td>
<td>4*10^4</td>
</tr>
<tr>
<td>8</td>
<td>222.785</td>
<td>8*10^4</td>
</tr>
<tr>
<td>16</td>
<td>128.394</td>
<td>16*10^4</td>
</tr>
<tr>
<td>32</td>
<td>133.172</td>
<td>16*10^4</td>
</tr>
<tr>
<td>64</td>
<td>190.983</td>
<td>64*10^4</td>
</tr>
<tr>
<td>128</td>
<td>396.292</td>
<td>128*10^4</td>
</tr>
</tbody>
</table>

1 core per node  Data/PE = 10^4
Efficiency

1 core per Node     Input size $10^8$

<table>
<thead>
<tr>
<th>PE</th>
<th>Time in sec</th>
<th>Cost</th>
<th>Efficiency</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>733.451</td>
<td>1466.9</td>
<td>0.96</td>
</tr>
<tr>
<td>4</td>
<td>445.246</td>
<td>1780.98</td>
<td>0.79</td>
</tr>
<tr>
<td>8</td>
<td>285.531</td>
<td>2284.24</td>
<td>0.61</td>
</tr>
<tr>
<td>16</td>
<td>129.095</td>
<td>2065.44</td>
<td>0.68</td>
</tr>
<tr>
<td>32</td>
<td>107.378</td>
<td>3436.09</td>
<td>0.41</td>
</tr>
<tr>
<td>64</td>
<td>165.498</td>
<td>10591.87</td>
<td>0.13</td>
</tr>
<tr>
<td>128</td>
<td>385.445</td>
<td>49336.96</td>
<td>0.02</td>
</tr>
</tbody>
</table>

Efficiency = $\frac{T_{seq}}{\text{cost}}$

$T_{seq} = 1412.869$ sec

1 core per Node
Input size $10^8$
References

• AMCS Slides By Prof. Russ Miller
• GFG
• https://mpitutorial.com/tutorials
Thank You
Questions ?