SIEVE PARALLEL ALGORITHM

CSE633 – Shivangi Mishra

University at Buffalo The State University of New York

CONTENT

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

Sequential Algorithm

```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[i] = True
```

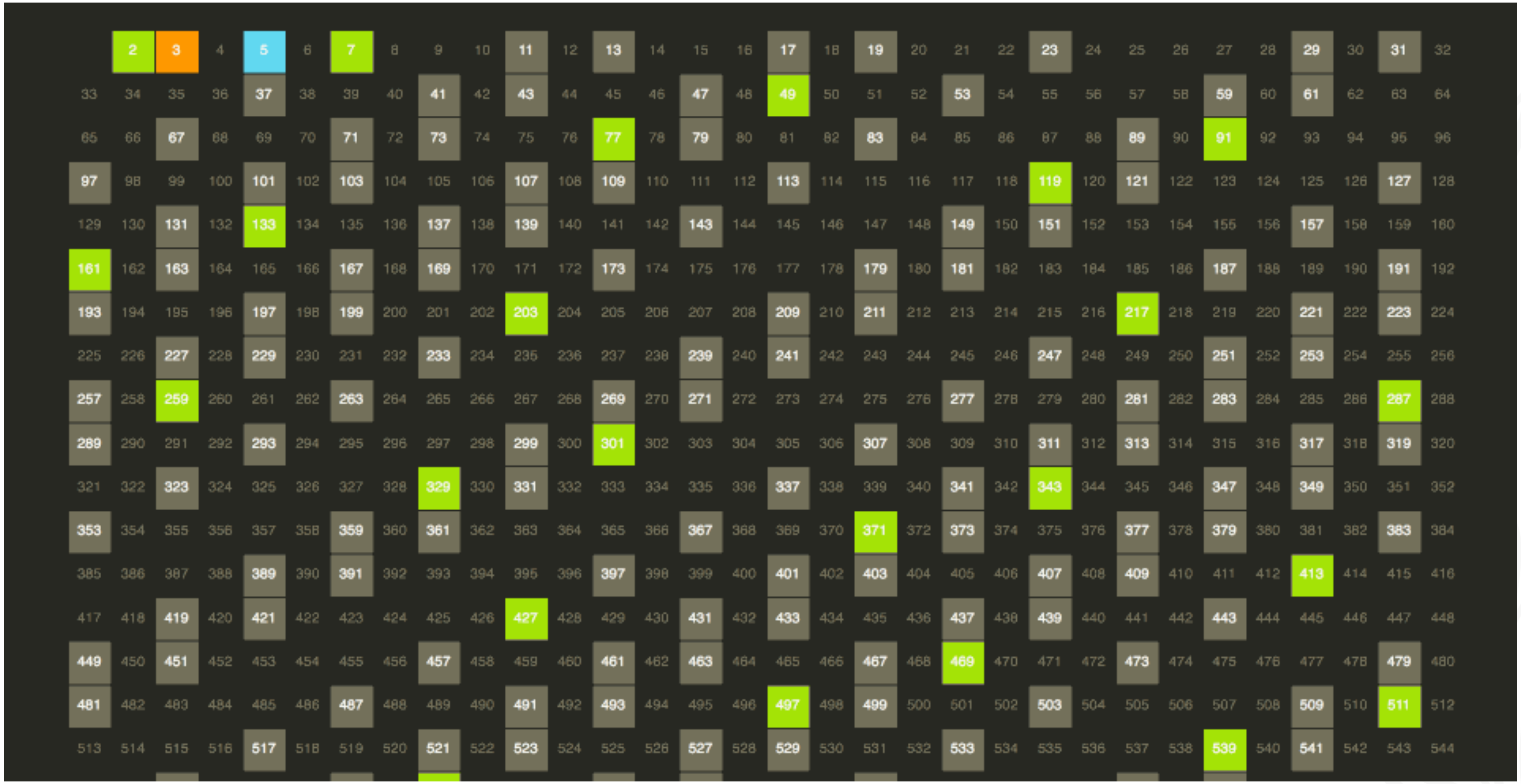
Time complexity: O(n^2)

- 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.

Sieve of Eratosthenes

- 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 Simulation



Sequential Sieve Algorithm

1	2	3	4	5	6
7	8	9	10	11	12
13	14	15	16	17	18
19	20	21	22	23	24
25	26	27	28	29	30
31	32	33	34	35	36

```
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 as composite
All unmarked nummbers are prime!
```

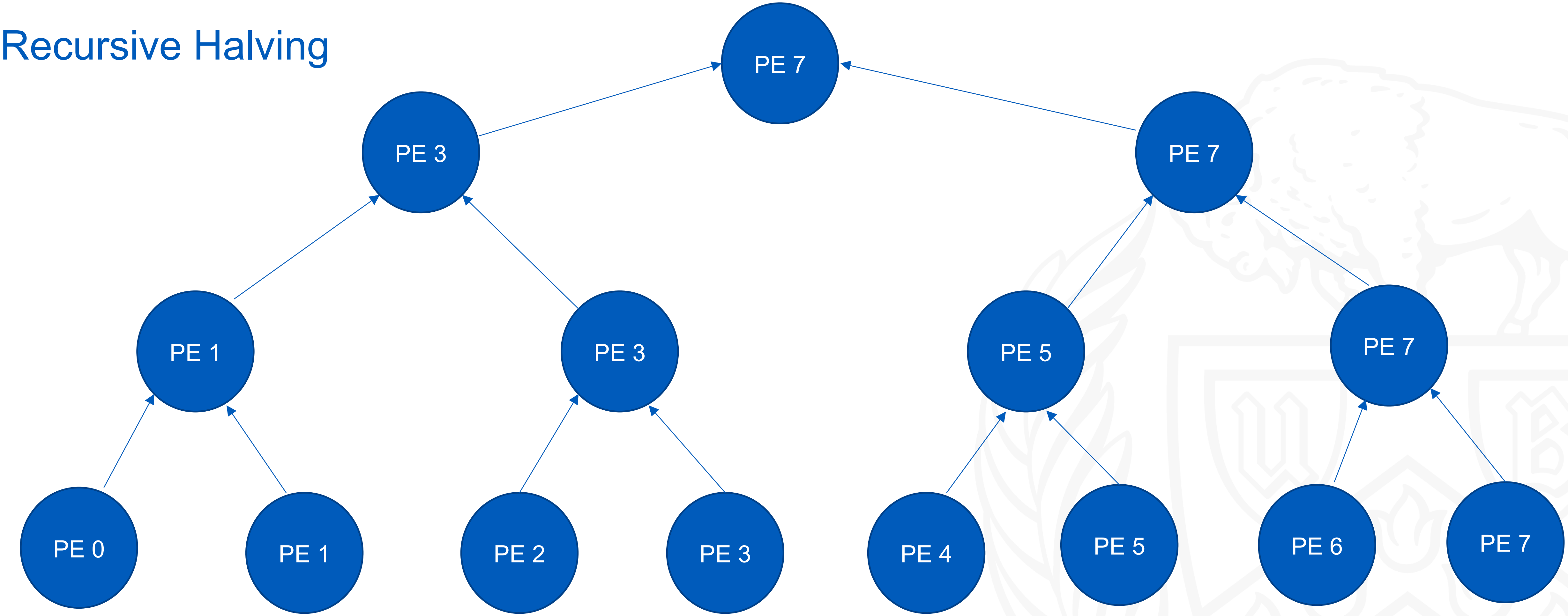
Pseudo code

Time complexity: O(n*log(log(n)))

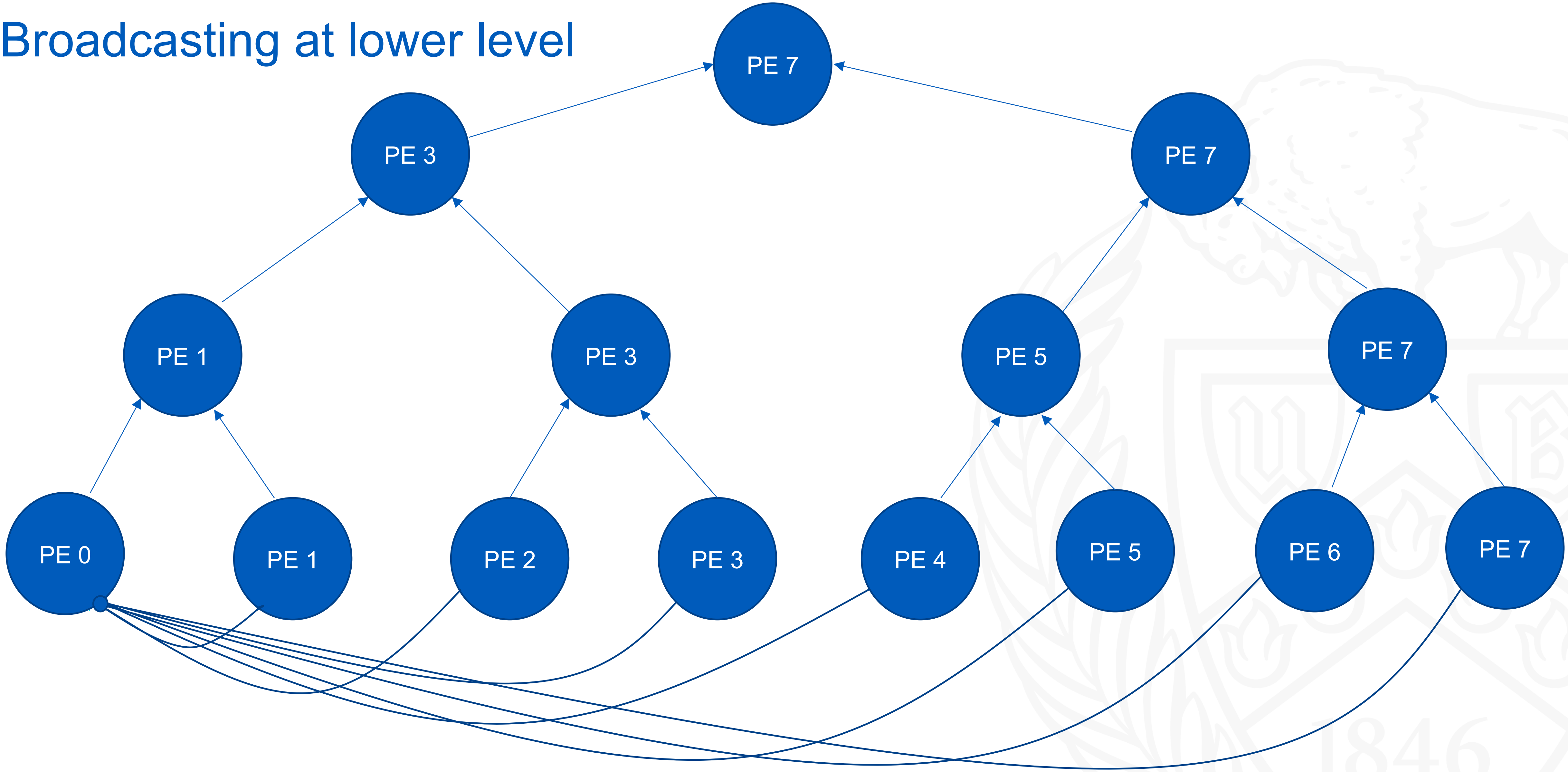
Parallel Sieve Implementation

- 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 bdcast_next_prime;
    for (c = low; c <= sqrtN; c++)
    {
        long int prime = -1;
        /* If the number is unmarked */
        if (marked2[c - low] == 1 & c != 2)
        {
            prime = c;
            for (int j = c + 1; j <= high; j++)
            {
                if (j % prime == 0)
                {
                    // printf(" div %ld\n", prime);
                    marked2[j - low] = 0;
                    // printf(" for rank 0 j= %d, low= %ld,prime= %ld\n", j, low, prime);
                }
            }
        }
        for (int i = 1; i < noOfProcesses; i++)
        {
            // printf(" Broadcasting markes=%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);
        }
    }
}
```

```c
else
{

    printf(" Processid %d\n", processId);
    int counter = 1;
    while (counter <= sqrtN)
    {
        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_IGNORE);
            // 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("after sending next prime  marked=%ld c=%ld next_prime=%ld rank=%d\n", marked2[c-low],c, next_prime,processId);
                }
                if (next_prime == 3 & c % next_prime == 0)
                {
                    // printf(" debug %ld, c=%ld, rank=%d marked=%ld,\n", next_prime, c, processId, marked2[c - low]);
                }
            }
        }
        counter++;
    }
}
```

Initial failed attempt for Broadcasting

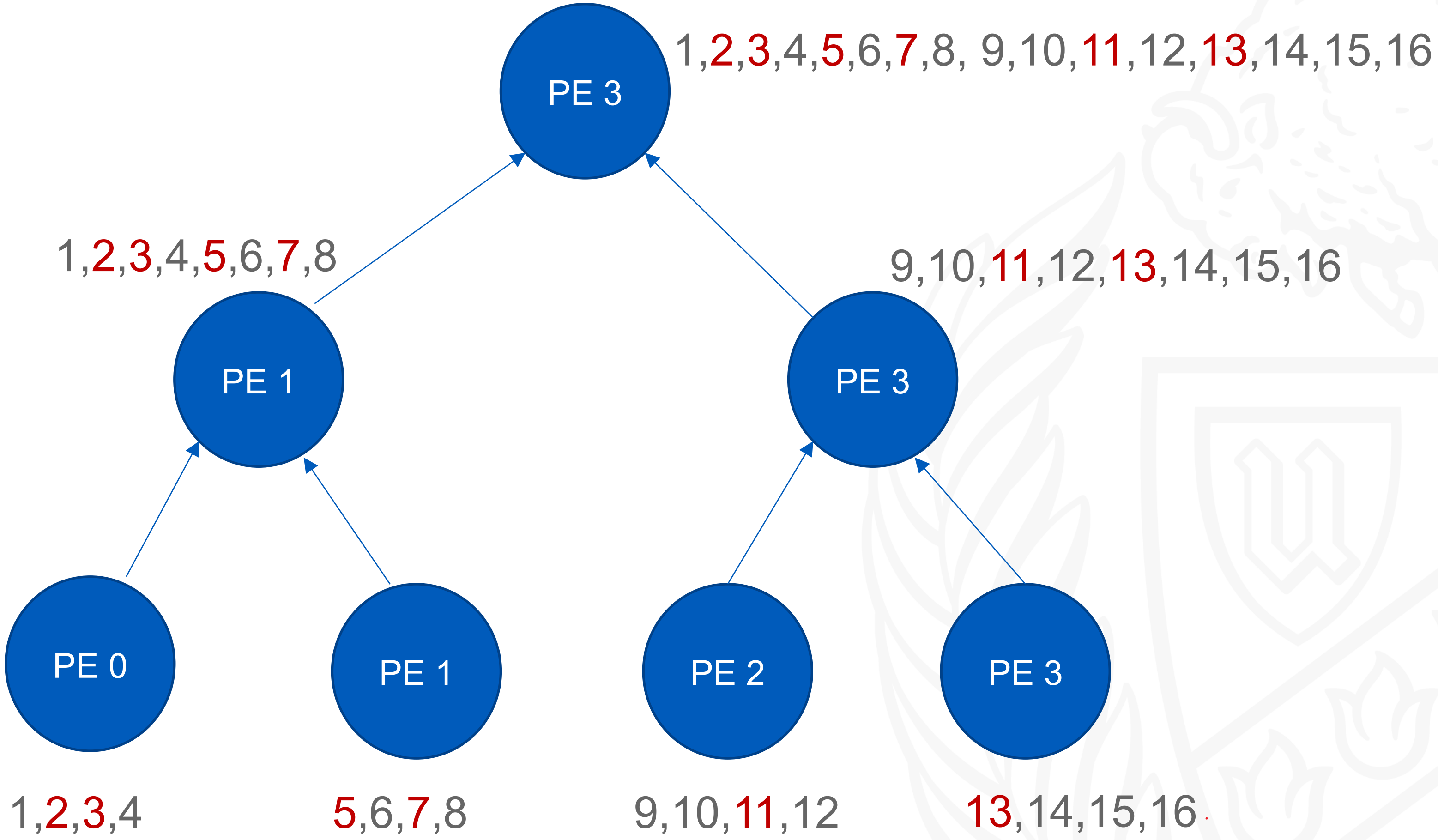
- Process 0 will send all the primes till sqrt(n) to all processes using MPI_Isend
- Other processes will receive the prime using MPI_Irecv and cancel the multiples in their range.
- The receive buffer is is getting resolved at different times in each processor causing faulty results.

```c
while (counter <= sqrtN)
{
    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, processI
    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_STAT
            // 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;
```

```c
            for (int i = 1; i < noOfProcesses; i++)
            {
                // printf(" Broadcasting markes=%ld prime=%ld\n", marked2[c - low], prim
                // 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(" Processid %d\n", processId);
        int counter = 1;
        while (counter <= sqrtN)
        {
            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, processI
            sleep(1);
```

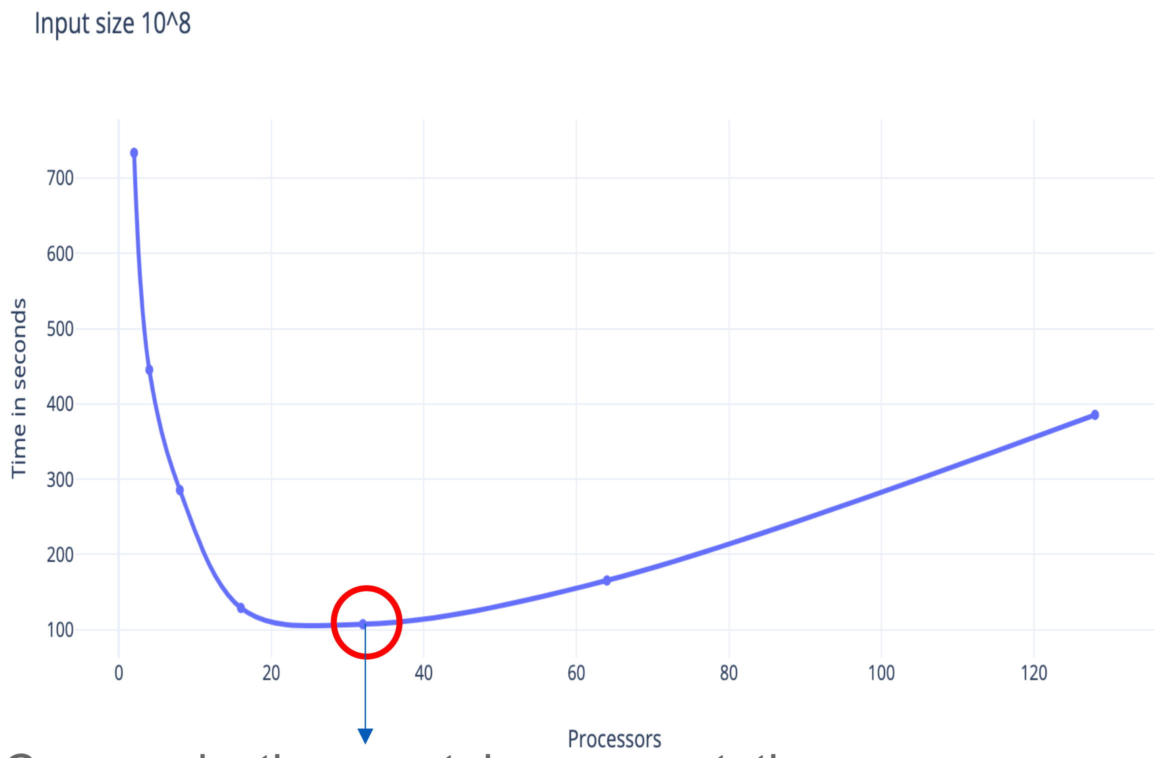
Parallel Stitch Step



Result parallel

1 core per Node

Processors	Time in sec
2	733.451
4	445.246
8	285.531
16	129.095
32	107.378
64	165.498
128	385.445

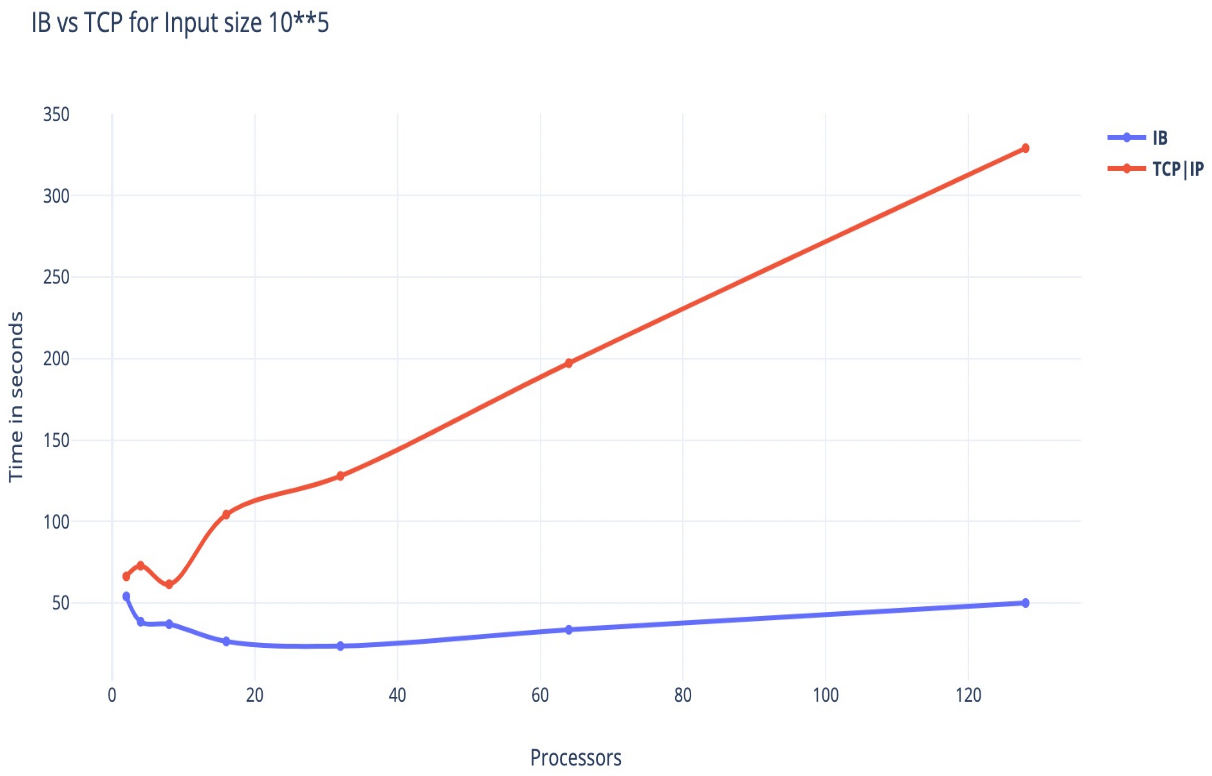


Communication overtakes computation

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

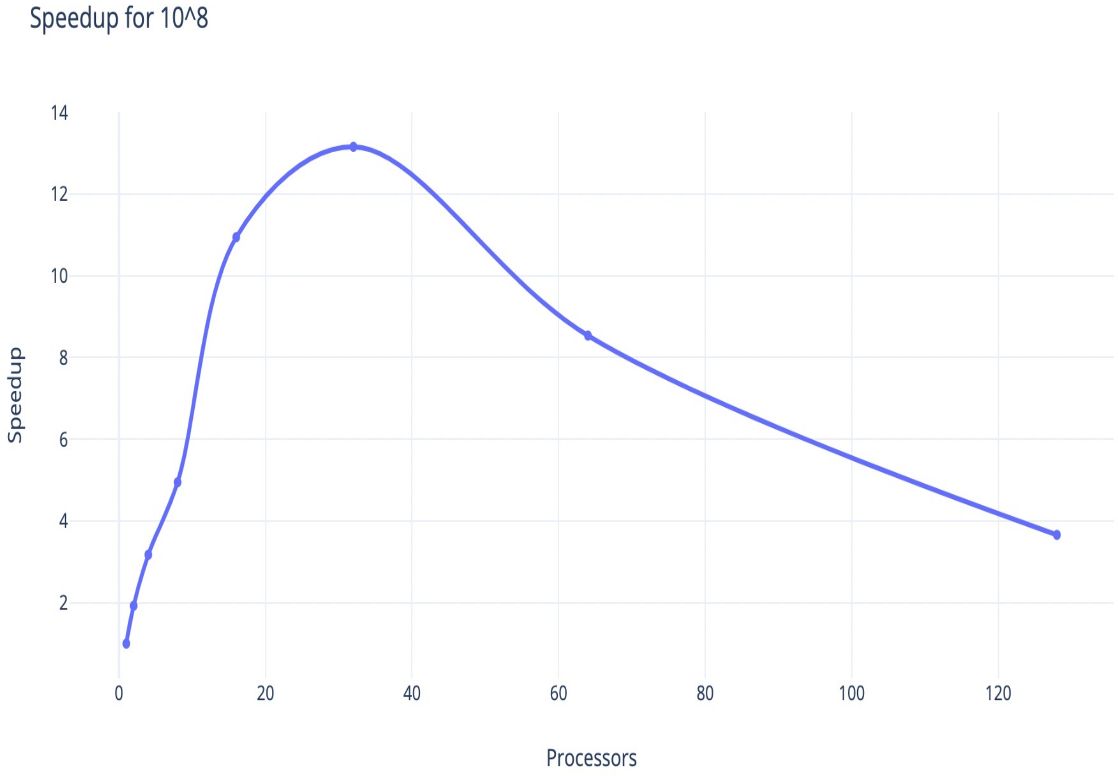


Speed-Up

1 core per Node Input size = 10^8

Processors	Speedup
1	1
2	1.926
4	3.173
8	4.948
16	10.944
32	13.157
64	8.537
128	3.66

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

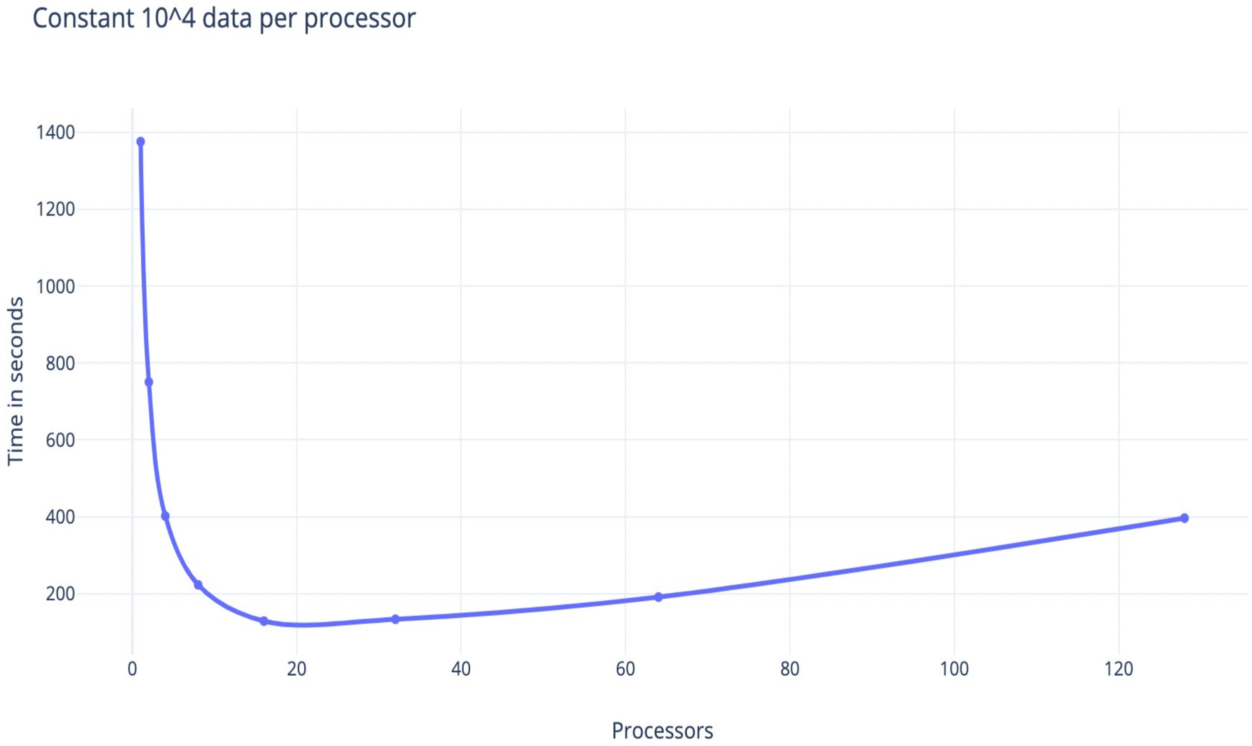


Scaled Result(Gustafson's law)

1 core per node

Data/PE = 10^4

Processors	Time in sec	Input size
1	1376.044	10^4
2	750.429	2*10^4
4	401.852	4*10^4
8	222.785	8*10^4
16	128.394	16*10^4
32	133.172	16*10^4
64	190.983	64*10^4
128	396.292	128*10^4

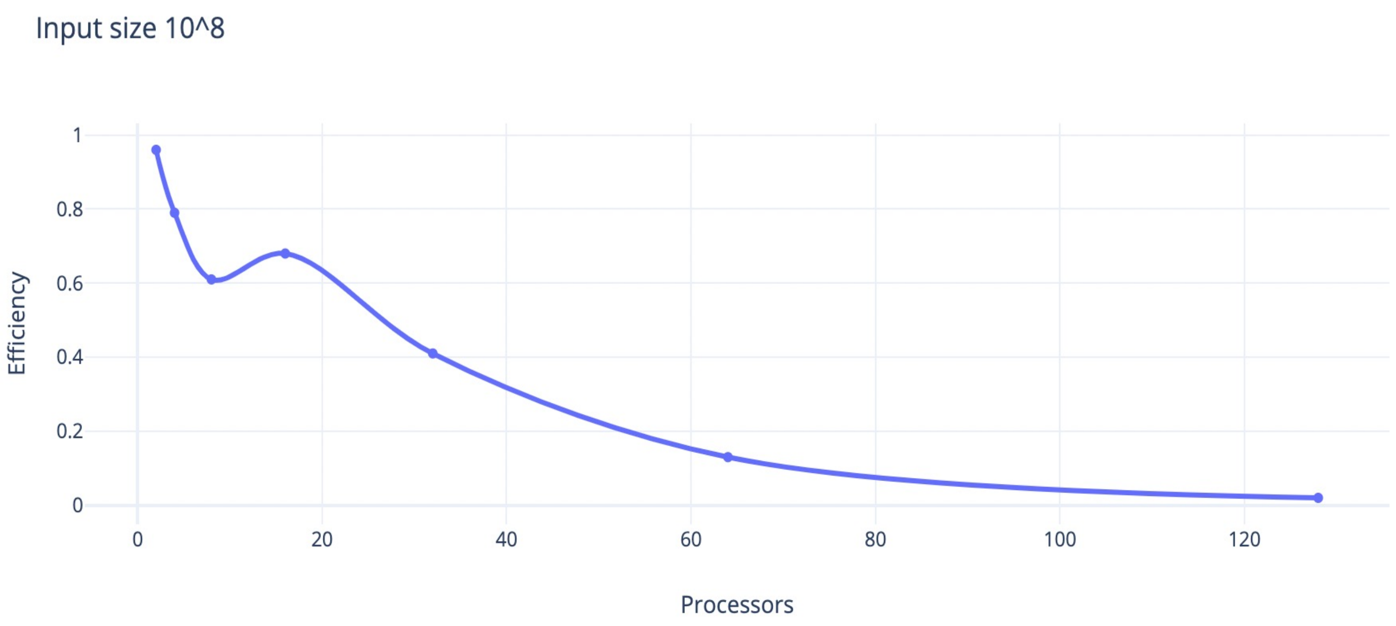


Efficiency

1 core per Node Input size 10^8

Efficiency = $\frac{T_{seq}}{cost}$ T_{seq} = 1412.869 sec

PE	Time in sec	Cost	Efficiency
2	733.451	1466.9	0.96
4	445.246	1780.98	0.79
8	285.531	2284.24	0.61
16	129.095	2065.44	0.68
32	107.378	3436.09	0.41
64	165.498	10591.87	0.13
128	385.445	49336.96	0.02



References

- AMCS Slides By Prof. Russ Miller
- GFG
- https://mpitutorial.com/tutorials

Thank You
Questions ?