Parallel Prime Number Generation.

CSE 633 – PARALLEL ALGORITHMS (SPRING 2020)

PRESENTER: PRIYESH PATEL

INSTRUCTOR: DR. RUSS MILLER, DR. M. D. JONES

Outline

- Why generate Prime numbers?
- Sieve of Eratosthenes algorithm
- Parallel approach
- Data distribution
- Results
- Observations
- Challenges

Why generate prime numbers?

- Several cryptography algorithms like RSA or Diffie Hellman key exchange are based on large prime numbers.
- Computationally large prime number is likely to be a cryptographically strong prime.
- However, as the length of the cryptographic key values increases, this will result in the increased demand of computer processing power to create a new cryptographic key pair.
- Prime numbers also hold an important place in Quantum Mechanics.

Sieve of Eratosthenes algorithm

0	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	37	38	39
40	41	42	43	44	45	46	47	48	49
50	51	52	53	54	55	56	57	58	59
60	61	62	63	64	65	66	67	68	69
70	71	72	73	74	75	76	77	78	79
80	81	82	83	84	85	86	87	88	89
90	91	92	93	94	95	96	97	98	99

Source: https://www.realclearscience.com/

Sieve of Eratosthenes algorithm

```
void sieve(int N) {
        bool isPrime[N+1];
        // initialize the boolean array to true
        for(int i = 0; i <= N;++i) {
            isPrime[i] = true;
        }
        isPrime[0] = false;
        isPrime[1] = false;
        // If N is prime, its factor will not be greater than sqrt(n)
        for(int i = 2; i * i <= N; ++i) {
            if(isPrime[i] == true) {
                //Mark all the multiples of i as composite numbers
                for(int j = i * i; j <= N ;j += i)
                    isPrime[j] = false;
            }
        }
    }
```

Parallel approach (data distribution)

- Block data decomposition
- If *n* is the number of elements and *p* be the number of processing elements,
  - Divide data(range) into *n/p* contiguous blocks of equal size.
  - Let *i* denote the rank of the process
    - Range start = i * (n/p)
    - Range end = (i + 1) * (n/p) -1

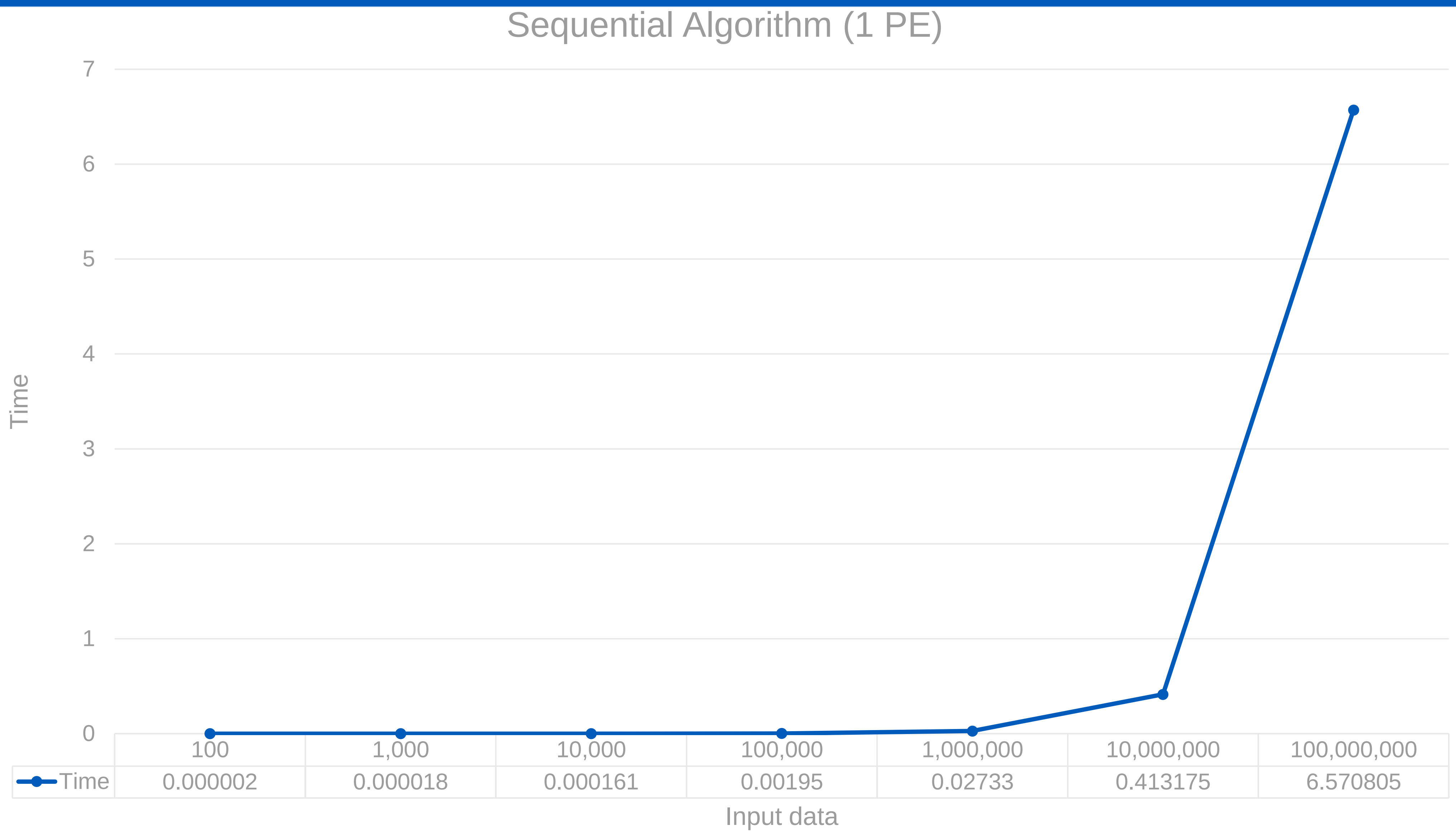
Parallel approach (Algorithm)

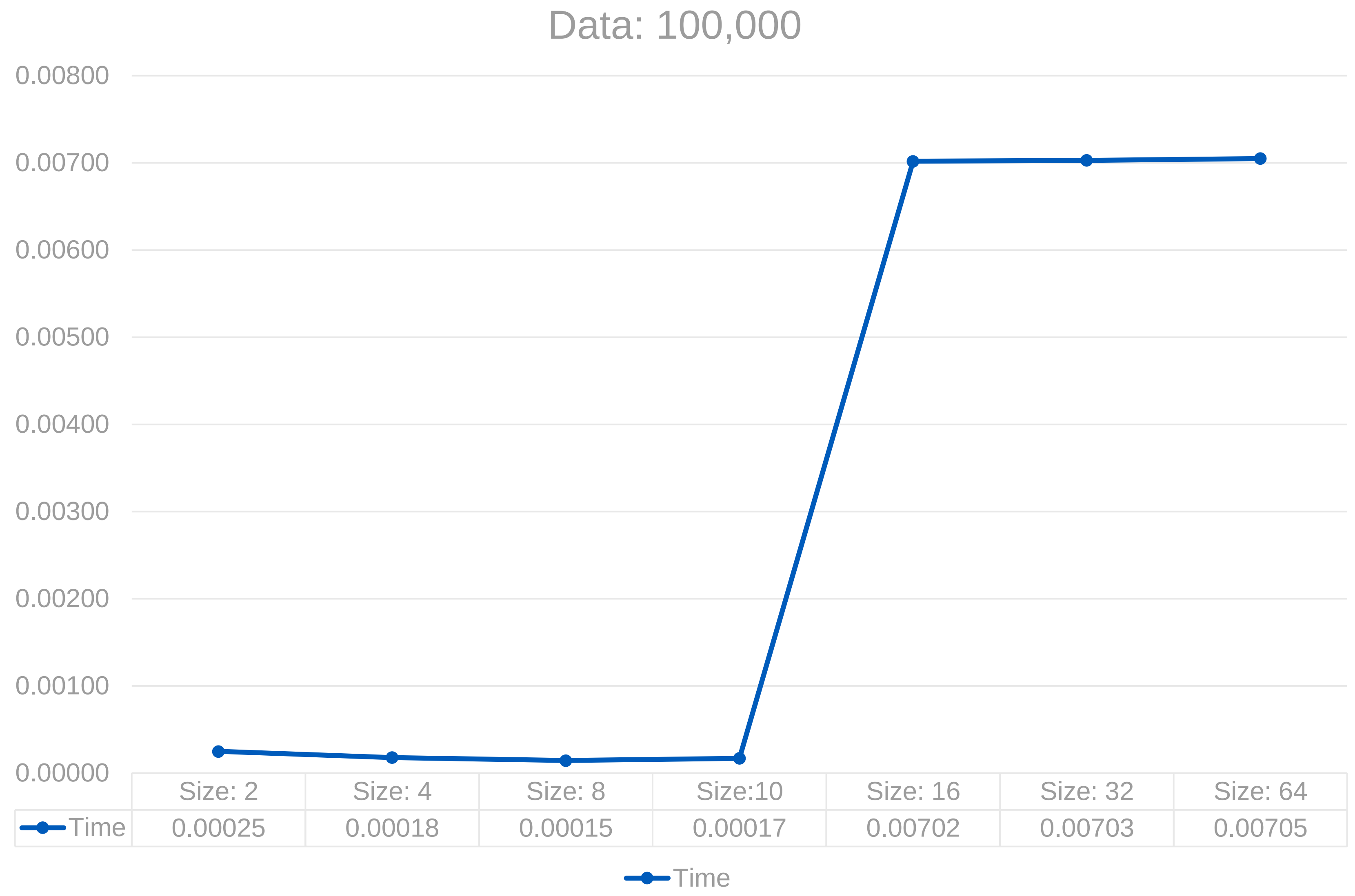
1. low = rank*(N/size)
2. high = (rank + 1)*(N/size) - 1
3. Initialize/mark `isPrime` array in the range low to high with true
4. Find the `next` marked global minimum value.
5. If `next` > sqrt(N), GOTO step (8)
6. Broadcast minimum `next` value to all processors.
7. Unmark all the multiples of `next` in the range low to high.
8. Apply reduce operation on each of the processor's array

Parallel approach (data consolidation)

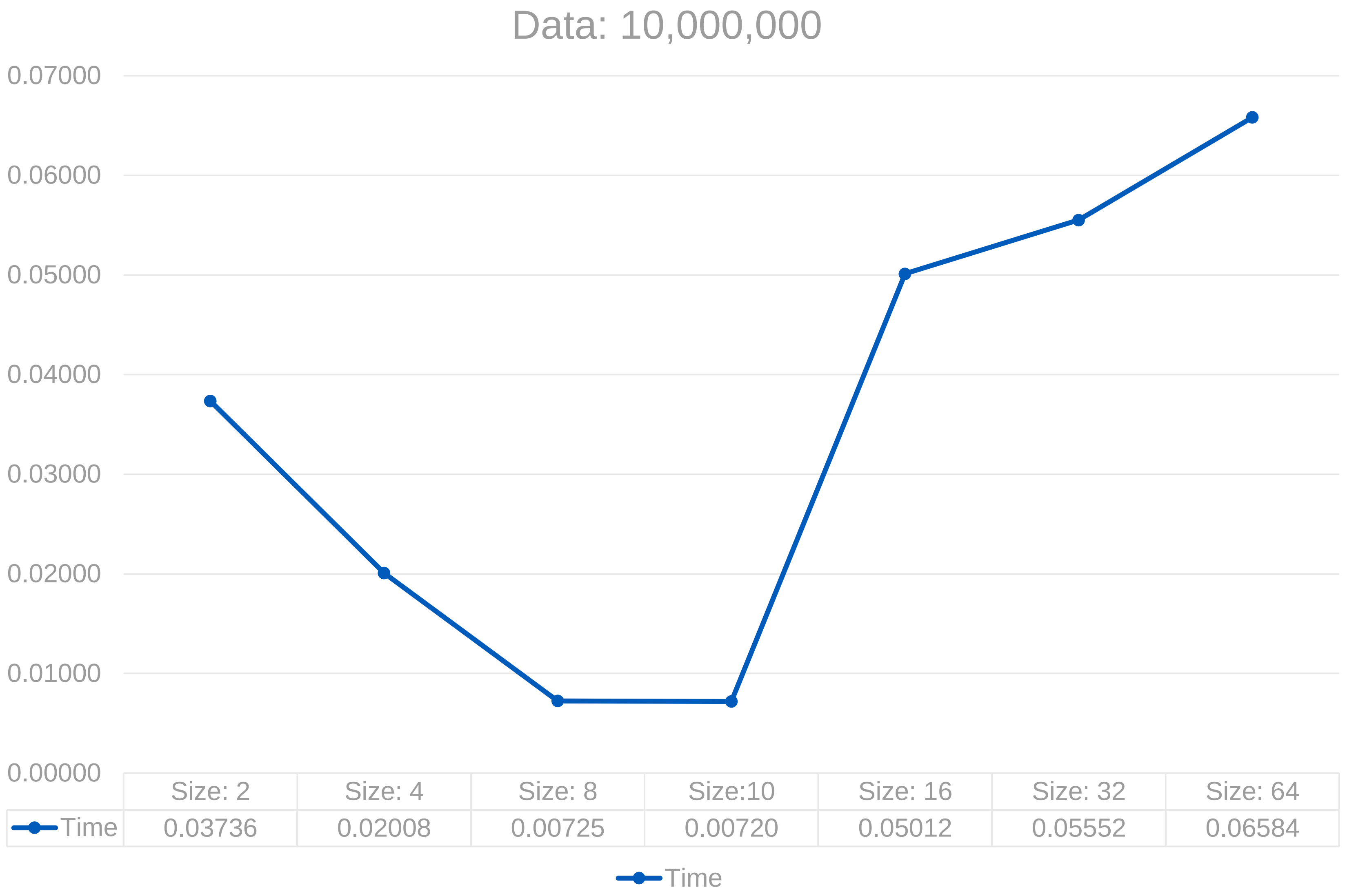
- Use reduction operation on every processor's Boolean array.
- Use MPI_REDUCE(sendbuffer, recvbuffer, count, datatype, op, root, comm)
  - Count = number of total elements
  - Op = MPI_BAND (bitwise AND operation)
  - Root = Processor 0 will receive final array when sieve's algorithm ends in the recvbuffer.

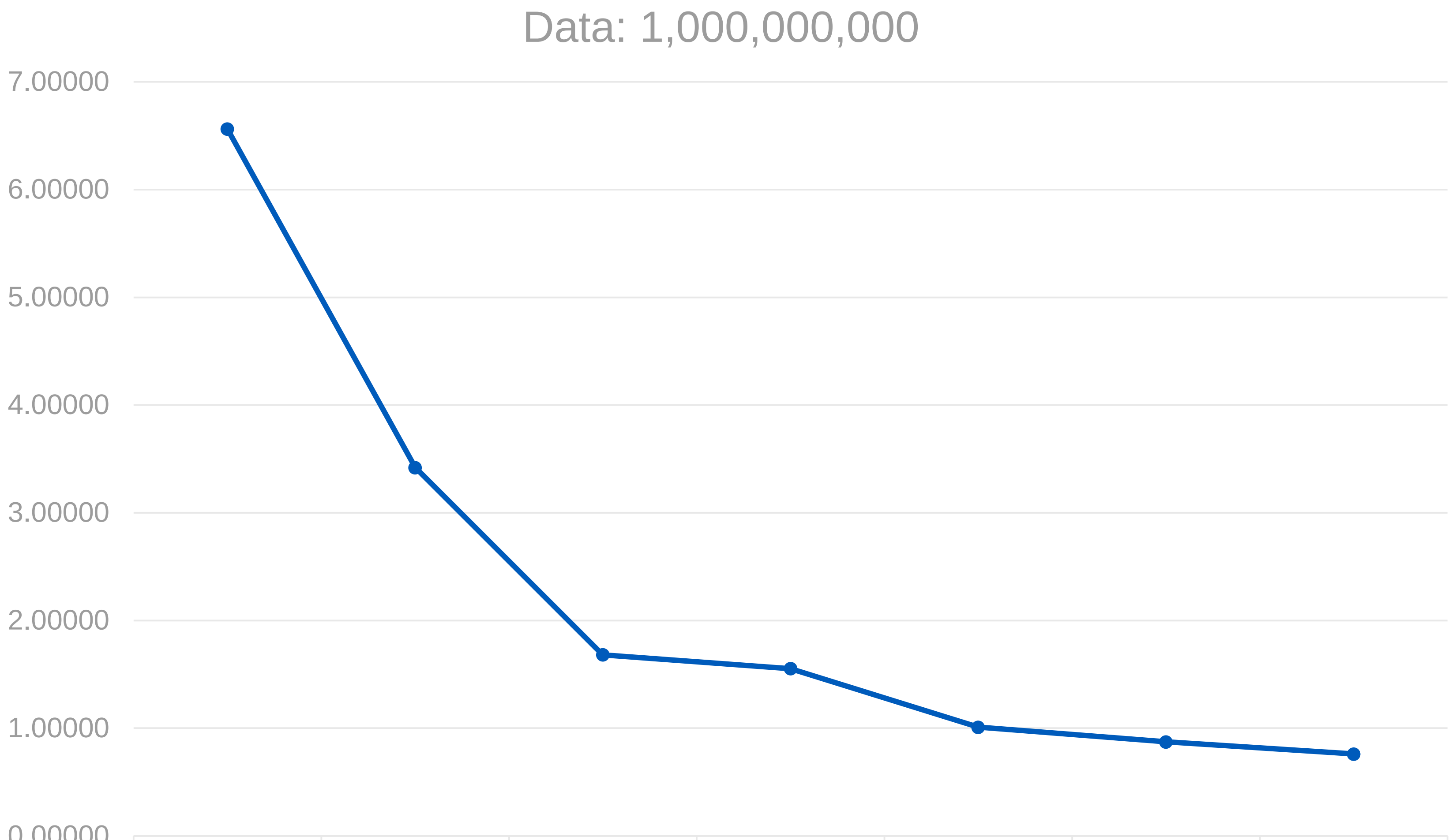
Results


Sequential Algorithm (1 PE)
Time
7
6
5
4
3
2
1
0
100
1,000
10,000
100,000
1,000,000
10,000,000
100,000,000
Time
0.000002
0.000018
0.000161
0.00195
0.02733
0.413175
6.570805
Input data



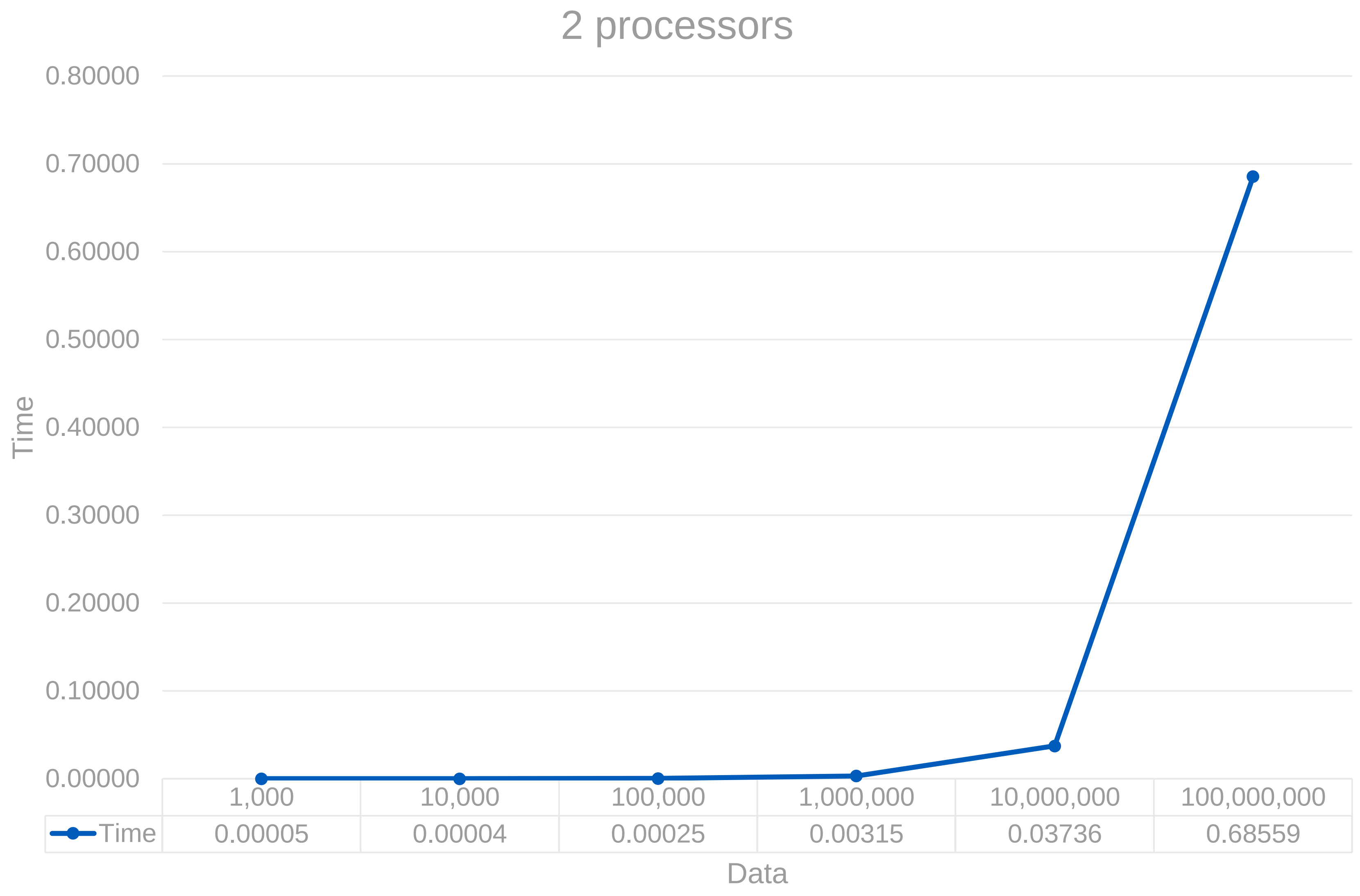
	Size: 2	Size: 4	Size: 8	Size:10	Size: 16	Size: 32	Size: 64
Time	0.00025	0.00018	0.00015	0.00017	0.00702	0.00703	0.00705


Data: 10,000,000
0.07000
0.06000
0.05000
0.04000
0.03000
0.02000
0.01000
0.00000
Size: 2
Size: 4
Size: 8
Size:10
Size: 16
Size: 32
Size: 64
Time
0.03736
0.02008
0.00725
0.00720
0.05012
0.05552
0.06584
Time

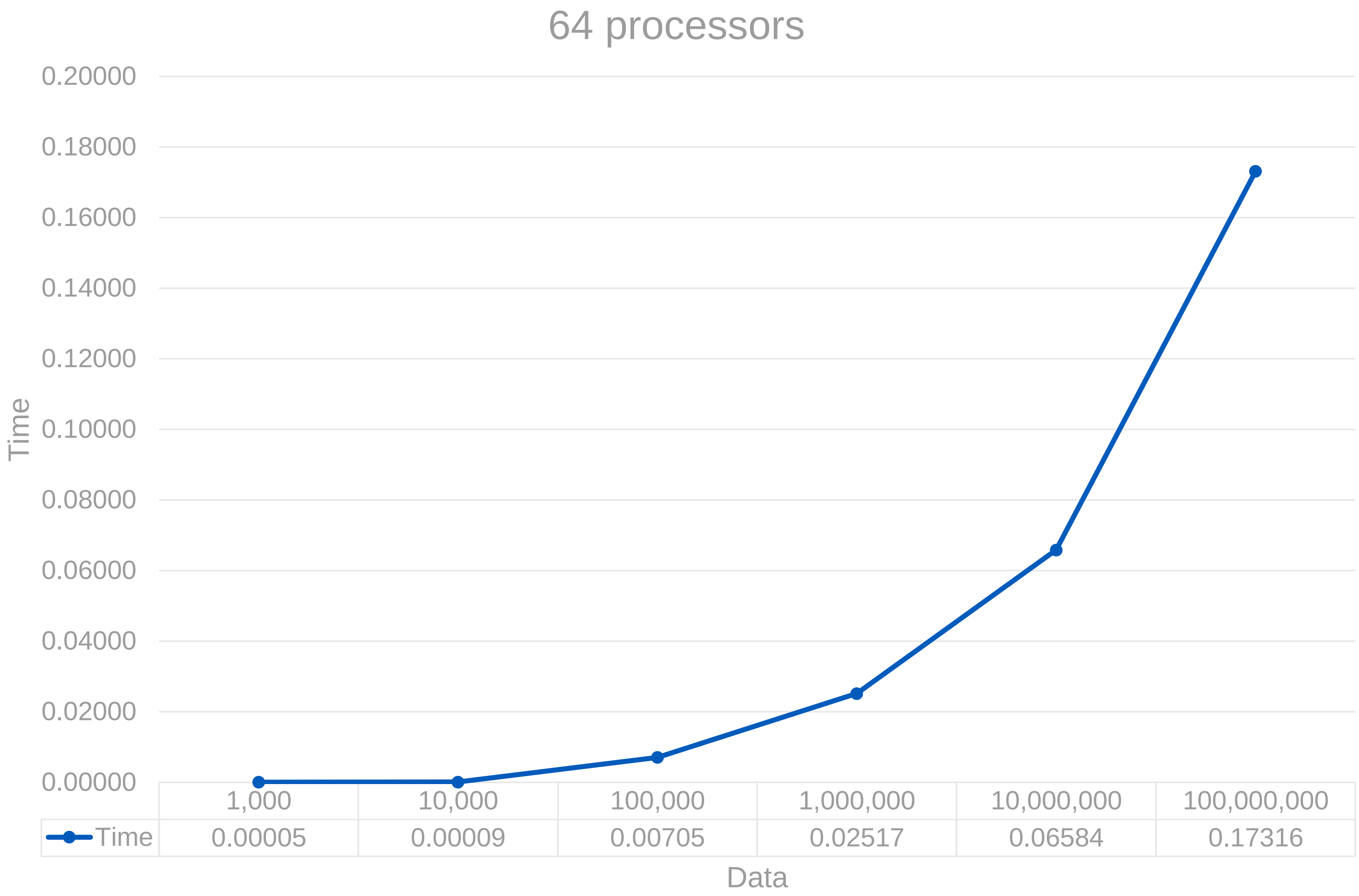


	Size: 2	Size: 4	Size: 8	Size:10	Size: 16	Size: 32	Size: 64
Time	6.56218	3.41948	1.68128	1.55250	1.00891	0.87242	0.76088

Time

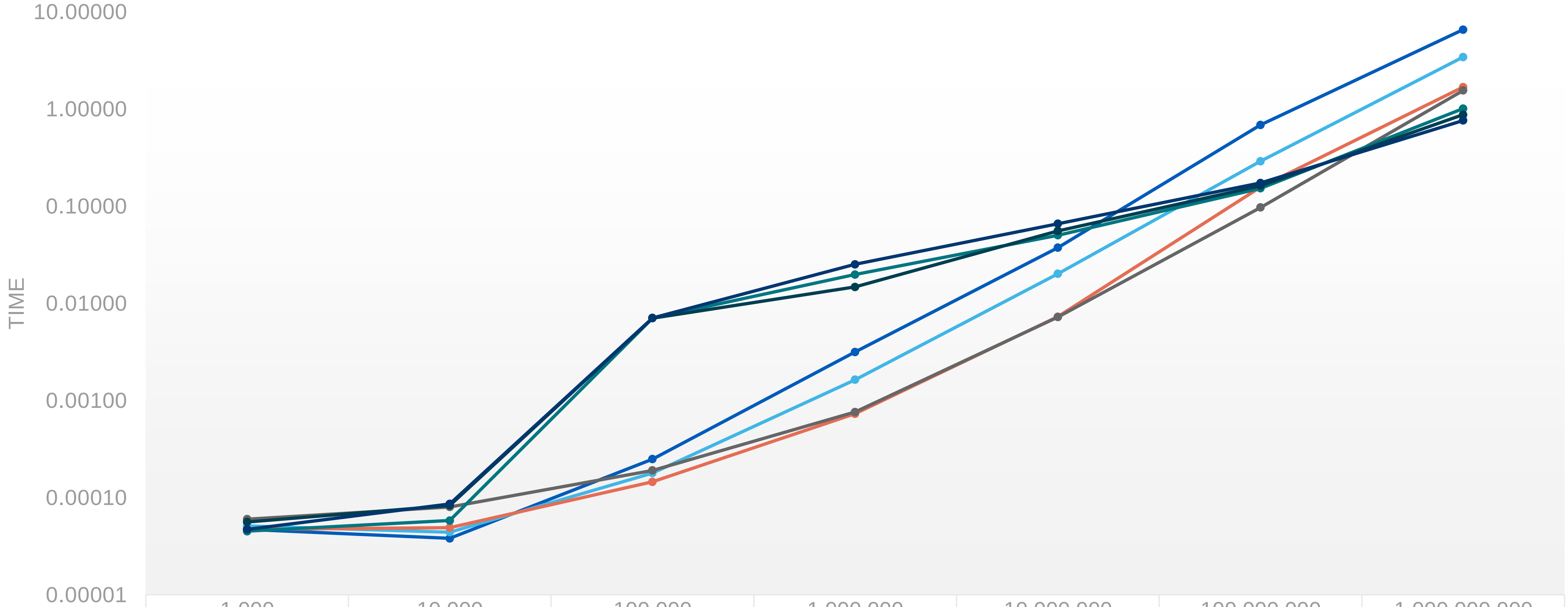

2 processors
0.80000
0.70000
0.60000
0.50000
0.40000
0.30000
0.20000
0.10000
0.00000
Time
1,000
10,000
100,000
1,000,000
10,000,000
100,000,000
Time
0.00005
0.00004
0.00025
0.00315
0.03736
0.68559
Data

	1,000	10,000	100,000	1,000,000	10,000,000	100,000,000
Time	0.00005	0.00005	0.00015	0.00073	0.00725	0.15677



	1,000	10,000	100,000	1,000,000	10,000,000	100,000,000
Time	0.00005	0.00009	0.00705	0.02517	0.06584	0.17316

Parallel Algorithm



	1,000	10,000	100,000	1,000,000	10,000,000	100,000,000	1,000,000,000
Size: 2	0.00005	0.00004	0.00025	0.00315	0.03736	0.68559	6.56218
Size: 4	0.00005	0.00004	0.00018	0.00163	0.02008	0.28967	3.41948
Size: 8	0.00005	0.00005	0.00015	0.00073	0.00725	0.15677	1.68128
Size: 10	0.00006	0.00008	0.00019	0.00076	0.00720	0.09700	1.55250
Size: 16	0.00005	0.00006	0.00702	0.01976	0.05012	0.15244	1.00891
Size: 32	0.00006	0.00008	0.00703	0.01470	0.05552	0.16225	0.87242
Size: 64	0.00005	0.00009	0.00705	0.02517	0.06584	0.17316	0.76088

INPUT DATA

Observations

- Smaller number of nodes couldn't handle more than 1 billion data.
- The run time was 3.2128 for 10 billion data when run on 128 node.
- As the input number gets bigger, the number of prime numbers found decreases and hence the communication between the processors also decreases.
- Most of the numbers are already cancelled in each of the processor ranges.
- The algorithm does scale well in parallel, if the input data increases.

Challenges

- Storing values that exceed the double datatype range.
- Getting the Boolean array space allocated for large numbers.
- Finding the first number in the processor's range that is the multiple of received prime number.

Thank You For Listening.
Questions?

Fun fact:

The largest known prime number is $2^{82,589,933} - 1$, a number which has 24,862,048 digits.