CSE-5351 Parallel Processing
Homework 4 Report

Parallel Prime Sieve of Eratosthenes Algorithm

Avinash Shanker
1001668570

Problem Statement:

Write a data-parallel program using distributed non-shared memory model as taught in the class for the sieve of Eratosthenes. Use MPI for message passing and test the program on the workstation cluster.

The program has two inputs: The largest number, n, up to which the prime numbers are to be found, and, p, the number of processors. The program should run for any number of processors ranging from 1 to 32.

Make the rest of the assumptions yourself but your grade will be based on how good your parallelization and communication scheme is, so feel free to make optimizations. Attach a note on your parallelization scheme. The output of your program is the prime numbers found by your program.

Also implement the sequential program and calculate and make the plot for the following:

1. Execution time versus the number of processors for 1, 2, 4, 8, 16, 32
2. Execution speedup versus the number of processors for 1, 2, 4, 8, 16, 32
3. Efficiency versus the number of processors for 1, 2, 4, 8, 16, 32

Solution and Explanation:

Parallel Program

- To compile: *mpicc prime_p.c*
- Run the code in 2 modes:
    - Mode1(Prime count and Time): *ibrun -np 2 ./a.out 1000000*
    - Mode2(Prime count, Prime numbers and Time): *ibrun -np 2 ./a.out 100000 PRIMES*

Sequential Program

- To compile: *mpicc prime_s.c*
- Run the code*: ibrun -np 1 ./a.out 1000000*

Attachments:

- *prime_p.c (parallel program) and prime_s.c (sequential program)*
- *Report.docx*
- *Graph.xls*
- *Output.txt*

***Note**: Both outputs are written to /a.out and ensure to re-compile before running.

Implementation Steps:

1. To divide the workload as evenly as possible among all the processors, first the master node will setup all integers up to √n.

2. Now, divide the left-out integers among the rest of the slave processes. If the work is less for the master node, redistribute among the slave and master node.

3. Each process will calculate its starting and ending integer assigned to it. Previous step ensures there is even distribution of work.

4. Uneven distribution of integers can occur only when the integers are very small for number of processors as master node has to hold √n.

5. Now each process will allocate space for an array equal to the numbers it is was assigned.

6. Every element in the array will be set to Zero denoting that the integer could be a prime number.

7. The Master node will initialize its first element to 1 denoting a non-prime number after which sieve process will start.

8. The master process has the job of broadcasting the next prime number. First value is 2.

9. Calculate the first index of the first multiple found in the array, so its multiples can be marked by jumping indexes time the multiples.

10. For example, if the first prime is 2, jump by 2 indices until the end of array and mark the numbers, this way you don't have to calculate the multiple separately.

11. Every process must go through its respective array and mark all the integers that are the multiples of current prime.

12. Then the current prime is added to added as a counter label.

13. Each time we reach the array bound, next unmarked number is picked and broadcasted among the processors.

14. Once we reach the root n value, all the processes count the number of unmarked elements and via MPI_Reduce and send this data back to master process.

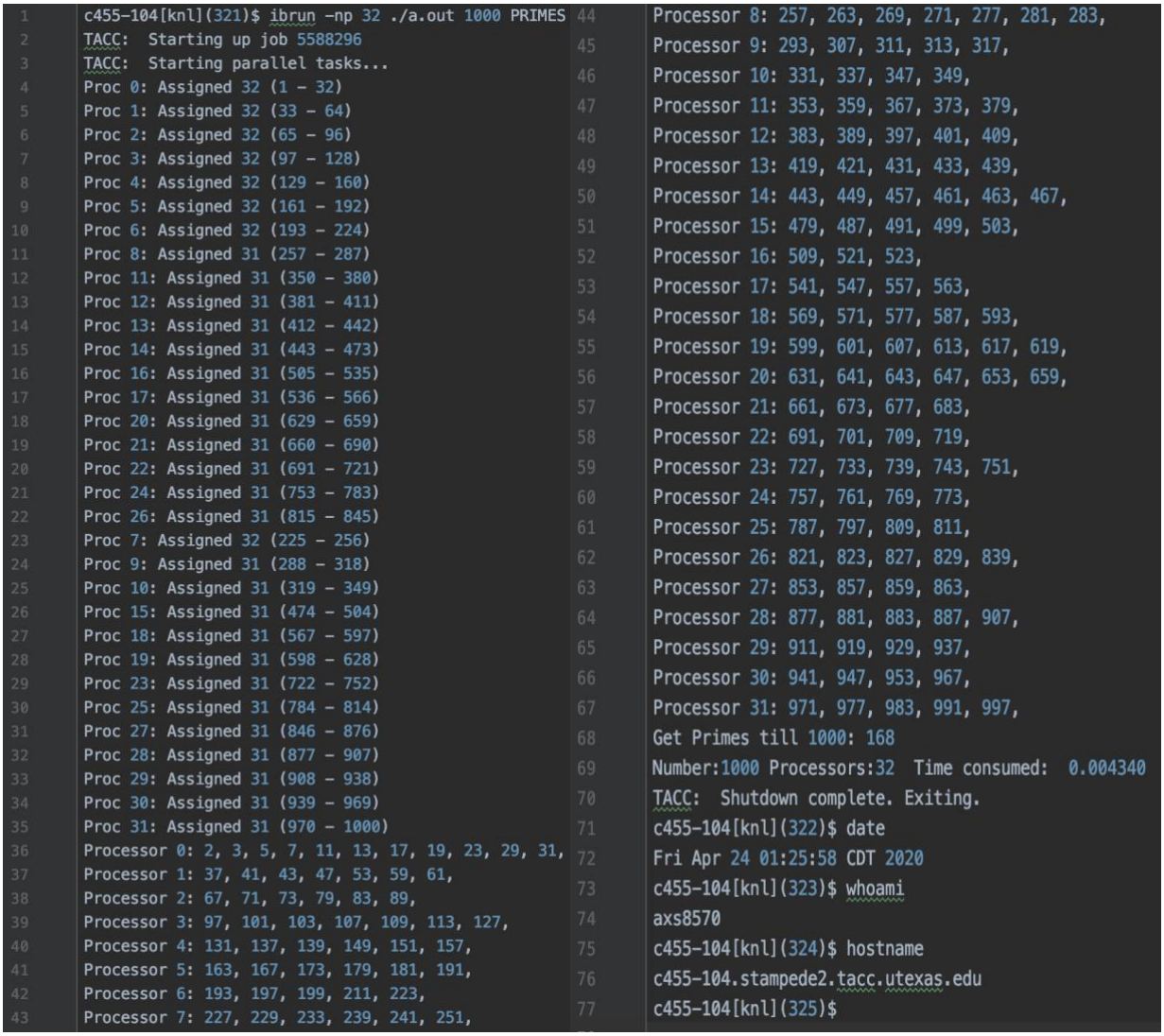
15. The Master node can now return this value to the user along with time taken to execute.

16. Also, in order to obtain all the prime number as well, in command state added parameter PRIMES will print prime numbers till n as well.

17. But, to do this all processes greater than 0 will have to wait in a blocking command MPI_Recv during which the master process prints before invoking MPI_Send to each process.

Output:

- As above mentioned in the implementation steps, I have attached the snapshot of code running on 32 processors for N=1000
- It is printing all the prime numbers between 2 and 1000
- To better understand, I have added print statements to show how the ranges are divided between each processor
- Also, the primes calculated by each processor.
- The code has returned 168 primes between 2-1000



```
c455-104[knl](321)$ ibrun -np 32 ./a.out 1000 PRIMES
TACC:  Starting up job 5588296
TACC:  Starting parallel tasks...
Proc 0: Assigned 32 (1 - 32)
Proc 1: Assigned 32 (33 - 64)
Proc 2: Assigned 32 (65 - 96)
Proc 3: Assigned 32 (97 - 128)
Proc 4: Assigned 32 (129 - 160)
Proc 5: Assigned 32 (161 - 192)
Proc 6: Assigned 32 (193 - 224)
Proc 8: Assigned 31 (257 - 287)
Proc 11: Assigned 31 (350 - 380)
Proc 12: Assigned 31 (381 - 411)
Proc 13: Assigned 31 (412 - 442)
Proc 14: Assigned 31 (443 - 473)
Proc 16: Assigned 31 (505 - 535)
Proc 17: Assigned 31 (536 - 566)
Proc 20: Assigned 31 (629 - 659)
Proc 21: Assigned 31 (660 - 690)
Proc 22: Assigned 31 (691 - 721)
Proc 24: Assigned 31 (753 - 783)
Proc 26: Assigned 31 (815 - 845)
Proc 7: Assigned 32 (225 - 256)
Proc 9: Assigned 31 (288 - 318)
Proc 10: Assigned 31 (319 - 349)
Proc 15: Assigned 31 (474 - 504)
Proc 18: Assigned 31 (567 - 597)
Proc 19: Assigned 31 (598 - 628)
Proc 23: Assigned 31 (722 - 752)
Proc 25: Assigned 31 (784 - 814)
Proc 27: Assigned 31 (846 - 876)
Proc 28: Assigned 31 (877 - 907)
Proc 29: Assigned 31 (908 - 938)
Proc 30: Assigned 31 (939 - 969)
Proc 31: Assigned 31 (970 - 1000)
Processor 0: 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31,
Processor 1: 37, 41, 43, 47, 53, 59, 61,
Processor 2: 67, 71, 73, 79, 83, 89,
Processor 3: 97, 101, 103, 107, 109, 113, 127,
Processor 4: 131, 137, 139, 149, 151, 157,
Processor 5: 163, 167, 173, 179, 181, 191,
Processor 6: 193, 197, 199, 211, 223,
Processor 7: 227, 229, 233, 239, 241, 251,
Processor 8: 257, 263, 269, 271, 277, 281, 283,
Processor 9: 293, 307, 311, 313, 317,
Processor 10: 331, 337, 347, 349,
Processor 11: 353, 359, 367, 373, 379,
Processor 12: 383, 389, 397, 401, 409,
Processor 13: 419, 421, 431, 433, 439,
Processor 14: 443, 449, 457, 461, 463, 467,
Processor 15: 479, 487, 491, 499, 503,
Processor 16: 509, 521, 523,
Processor 17: 541, 547, 557, 563,
Processor 18: 569, 571, 577, 587, 593,
Processor 19: 599, 601, 607, 613, 617, 619,
Processor 20: 631, 641, 643, 647, 653, 659,
Processor 21: 661, 673, 677, 683,
Processor 22: 691, 701, 709, 719,
Processor 23: 727, 733, 739, 743, 751,
Processor 24: 757, 761, 769, 773,
Processor 25: 787, 797, 809, 811,
Processor 26: 821, 823, 827, 829, 839,
Processor 27: 853, 857, 859, 863,
Processor 28: 877, 881, 883, 887, 907,
Processor 29: 911, 919, 929, 937,
Processor 30: 941, 947, 953, 967,
Processor 31: 971, 977, 983, 991, 997,
Get Primes till 1000: 168
Number:1000 Processors:32  Time consumed:  0.004340
TACC:  Shutdown complete. Exiting.
c455-104[knl](322)$ date
Fri Apr 24 01:25:58 CDT 2020
c455-104[knl](323)$ whoami
axs8570
c455-104[knl](324)$ hostname
c455-104.stampede2.tacc.utexas.edu
c455-104[knl](325)$
```

Graphs and Results

1. Execution time vs the number of processors for 1, 2, 4, 8, 16, 32:

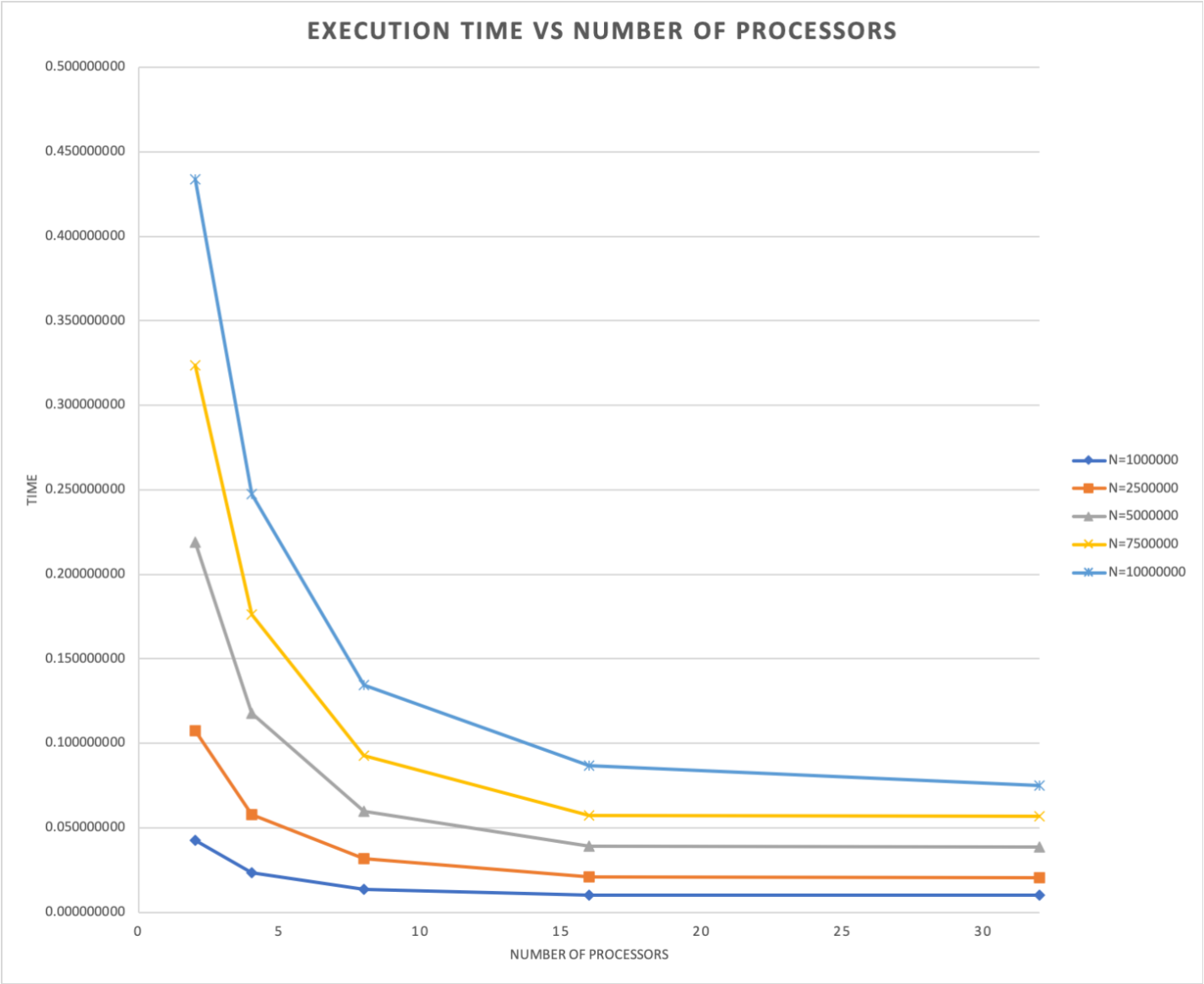


Figure 1

In Figure1:

- N= number in the range which prime numbers were generated. Graph compares the execution time of the program to compute primes till N with the numbers of processors.
- We can see that as the number of processors increases the time taken to compute the primes decreases.

2. Execution speedup versus the number of processors for 1, 2, 4, 8, 16, 32

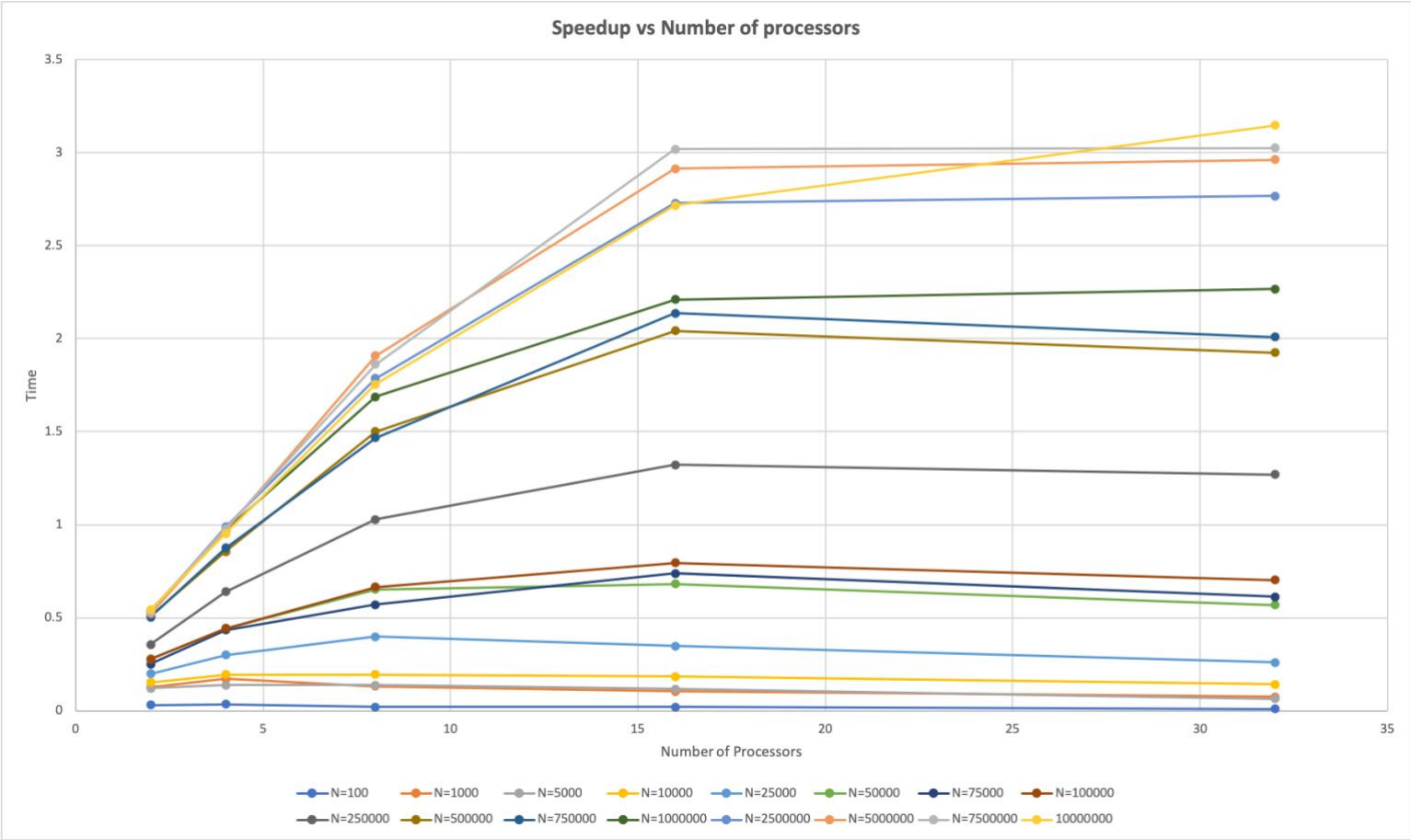


Figure 2

In Figure 2:

- The comparison between speedup and the number of processors for each N, where N represents the range in which prime numbers were calculated.
- I have calculated the speedup for multiple values of N ranging from 100 to 100000000

3. Efficiency versus the number of processors for 1, 2, 4, 8, 16, 32

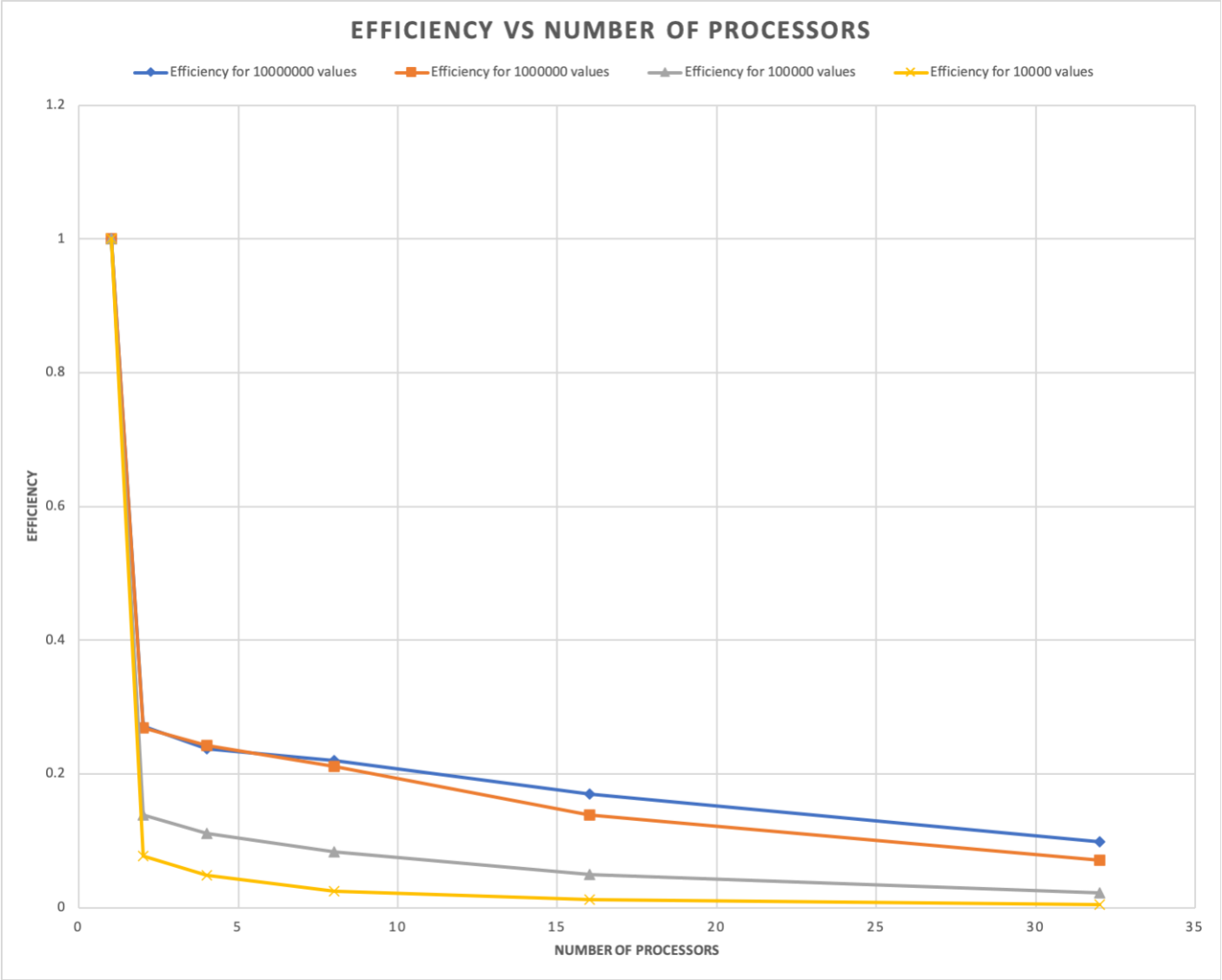


Figure 3

In Figure 3:

- I am comparing the efficiency against the numbers of processors for various values of N ranging from 10000 to 100000000.
- We can observe that the efficiency of the parallel program going down as the number of processors increases.
- This can occur due to multiple reasons, but mainly due to message passing and communication time and storage time between the processors.