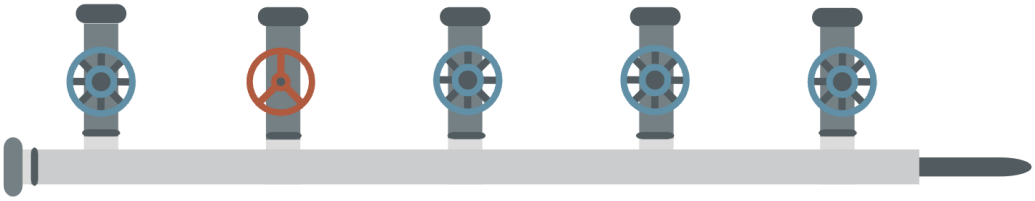
CMPE 275 - Enterprise Application Development

Design and Implementation of Grand Challenge - 02



Under the guidance of
Professor. John Gash,
San Jose State University.

Project submitted by **Team- Sprinters**

Sujan Rao Chikkela	015961062
Akshay Madiwalar	015924922
Sakruthi Avirineni	016009929
Arun Satvik Mallampalli	015999126
Rajashekar Reddy Kommula	016007043

GitHub Link

Powerpoint Presentation Link

Goal

To implement the grand challenge one pipe architecture in C/C++ without compiling the runtime with change in a number of processes and threads, with target CPU. In our approach, we have considered the main pipe as a master, which will distribute the work to workers. Each worker computes their part and sends back the result to the master. To achieve this goal, we have used OpenMPI to make parallel computation faster.

Problem Statement

Using the Sieve of Eratosthenes concept for finding prime numbers, we are designing and implementing a program using the C programming language to find all prime numbers to the Nth integer. We have taken the Max value of N = 2^32 (4,294,967,296)

This task is cumbersome and takes a lot of time with a single process. Therefore, We are Implementing this with MPI commands between multiple processes reinforced with OpenMP, if necessary, to achieve the optimum process to return the total number of primes found.

Target: We try to execute the program, allowing calculation for the number of primes between 1 and 4,294,967,296 (2^32))in roughly less than 35 seconds utilizing 4-8 nodes computing parallelly.

Approach

All of the criteria essential for this project were defined in the requirements process. Understanding the need for MPI/OpenMP, the upper bound necessary for assessing prime numbers, the resources and tools available, and the desired result were all part of this process.

The process of implementation and testing was further divided into a number of parts. First, a simple MPI program was written to test the "Hello world" output. The success of this project depended on our ability to successfully show that MPI can be implemented in the program code, compiled, and successfully run. A Makefile was made to expedite the compile and test procedures when it was shown that this could be run.

The design was further implemented after it was shown that MPI could be used. First, N, the command line's upper limit for evaluating prime numbers, and the debug mode were made available. This made testing more adaptable and made it possible for the Makefile to specify a list of test cases for performance testing. Then, all variables must be established, including

process rank, overall process count, etc. Process-sensitive activities, such as those for process 0, the master process, must be adequately defined because every process will receive a copy of the program.

After variables have been defined, the total workload must be evenly distributed. The logic is as follows:

1. The master process (process 0) set must have all integers up to the square root of N
2. The remaining integers can be divided among the worker processes (process 1+)
3. If the divided work is more than the work given to the master process, re-distribute everything evenly among all processes, giving the process 0 more work.
4. Uneven remaining work, the remainder, will be distributed among the worker processes.
5. Once all work has been divided, each process must calculate its starting integer and ending integer in its set

With this process flow, the workload is allocated as evenly as possible, with most processes having precisely the same amount of work within a single integer. The only extreme circumstances are when there are very few processes, and the issue size is very tiny. In these cases, the master process will keep more numbers since it MUST hold all integers up to the square root of N to guarantee that all primes are broadcast for sieving.

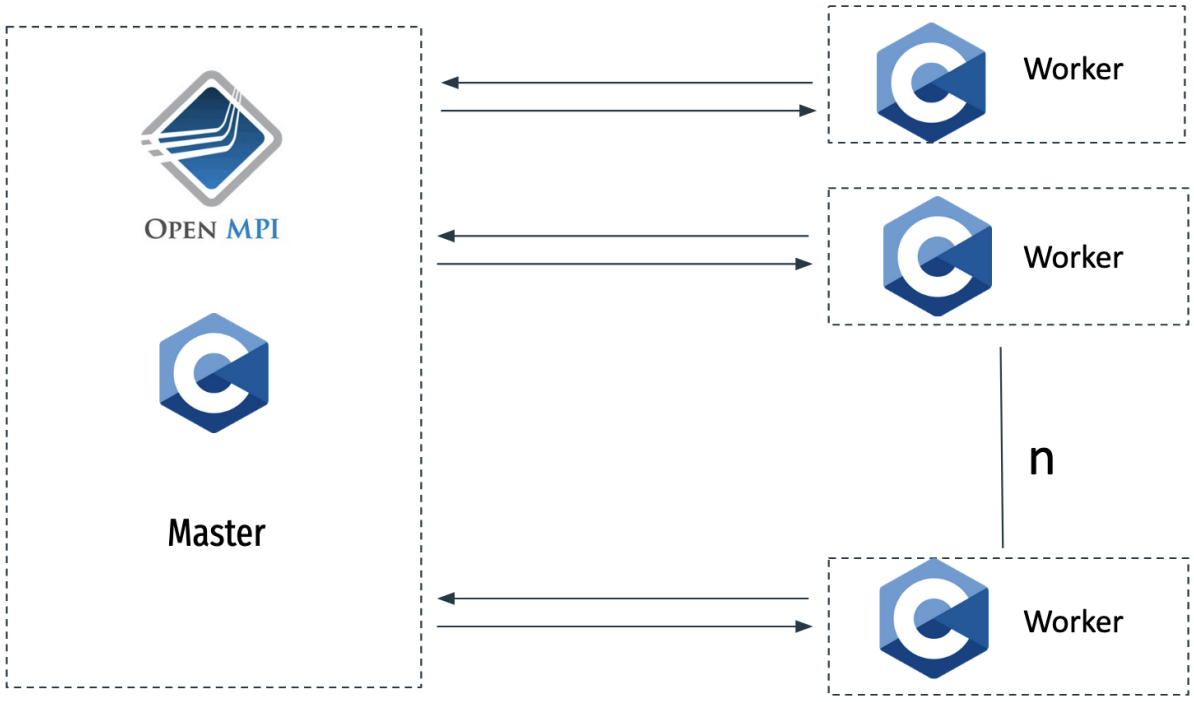
Each process can now allocate space for a character array equal to the number of integers that the process is in charge of once the burden has been divided among them. Each element's initial value will be "0," indicating that it may be a prime number. Prior to initializing all parts, all processes additionally verify that this memory allocation was successful. The master process must also set the value of its first element, 1, which stands for a non-prime number, to "1".

The master process's unique responsibility is broadcasting the following prime number. The value two is first allocated to this integer. The master process must traverse its own array and mark all integers that are multiples of the current prime. Each process must do this as well. This is accomplished by first figuring out the index of the first multiple discovered in the array used by this method, assuming the multiple is there. It is then tested against the array's limits using that index, and if it is, it is tagged in a loop that keeps adding the current prime to this value until the array's end is reached.

Each process counts the number of unmarked items and then uses MPI_Reduce to send this data back to the master process. At this time, the program can be successfully terminated by the master process reporting the number. Debug statements also enable the user to view these values by printing them to the console. The master process displays to the screen before delivering MPI_Send to process 1 in this process, which requires all processes larger than 0 to wait in a blocking instruction called MPI_Recv. Each subsequent step then adheres to this pattern.

Prior to MPI housekeeping and task distribution, the timer for this procedure is set, and it is checked when the total number of primes is established. OpenMP was also tested in two areas in order to further enhance this procedure: (1) in the for-loop used to label all multiples that are not primed, and (2) in the for-loop used to add up all primes discovered. This can be achieved by simply adding the "pragma omp for" command above each loop and designating a parallel region that spans both sections.

Architecture



In order to divide significant computing problems into smaller, independent tasks that may be addressed in parallel, the Master-workers architecture is a general parallel design. The master serves as the main machine in charge and assigns tasks to the workers, who then do them on their own and return the results. Because more workers may be added to the system as needed, the centralized approach of this architecture allows for easy system design and scalability.

In this centralized system, one node is identified as the master, and the others serve as workers. All decisions are made by the master, who also distributes information to the workers for processing. The workers then return the processed information to the master. The master then compiles all the data from the workers and creates the finished product. Before the desired outcome is attained, this process may involve multiple repetitions.

Implementation

The Sieve of Eratosthenes was successfully implemented as a result of this process. The src folder in the attachment contains the code file sieve of erath.c and its related Makefile. The document linked contains all of the project's outcomes, including the raw data: TestResults.xlsx.

The following commands can be used to generate and test the sieve_of_erath MPI executable and the sieve_of_erath_omp and MPI + OMP executable

make # make both the files executable.

make debug # program runs in the debug mode with n = 100

make test # program is tested with multiple inputs from n = [100, 100,000,000] and processes = [2,11]

Print statements in DEBUG MODE could be out of sequence due to connection delay and buffering, but all values can be seen, and the overall prime count is written correctly.

Code: https://github.com/sujanchikkela/CMPE275_GrandChallenge2

```
// First, divide work assuming master process only needs sqrt(n) work
*n_master = (unsigned int) ceil(sqrt((double) *n)); // Give master process all base primes
*n_worker = (*n - *n_master)/(*p_count - 1);        // Distribute evenly remaining work
*remainder = (*n - *n_master)%(*p_count - 1);       // Find remaining uneven work

// If workers have more than master thread, redistribute
if(*n_worker > *n_master)
    {*n_master = *n_worker = *n / *p_count;        // All processes get even work
       *remainder = *n % *p_count;                 // Remainder to be divided
 }

// Now, figure out ever process' first and last integer
// Also, distribute remainder evenly among workers
```

Result Metrics :

Figure 1 shows the speedup at the maximum number of processors (N=4,294,967,295).

We can see that the time taken to run the program decreases with the increase in the number of processors.

When **N = 2^32(4294967296)**.

OpenMPI works better with a higher workload.

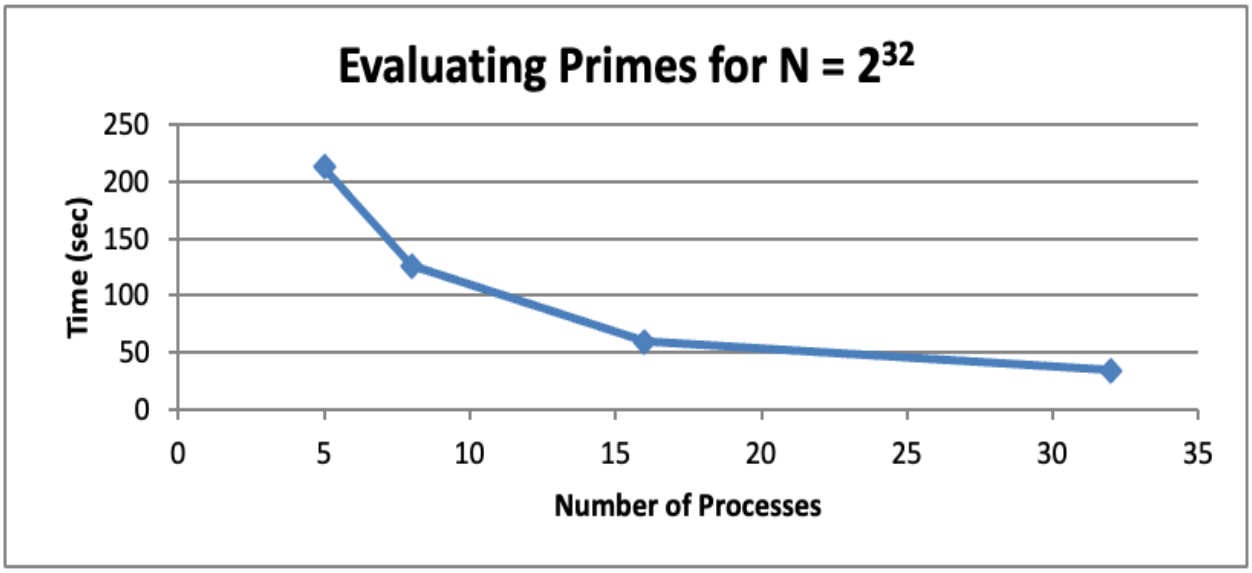


Figure1. *Processing time vs Number of Processes.*

Figure 2: Despite performing hundreds of tests, many of them showed greater noise in the results since the values for N were low. Figure 2 shows that runs with higher N values displayed more distinct patterns.

When N = 50M, 100M.

OpenMPI works better with a higher workload.

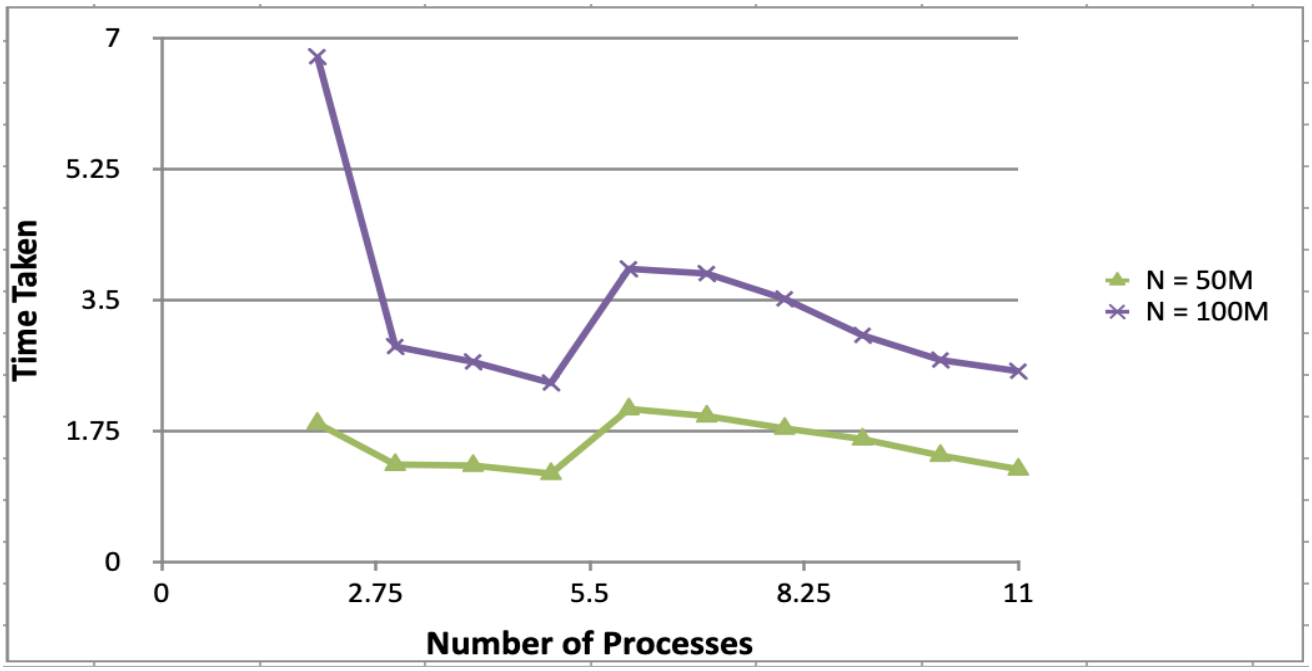


Figure 2. *Speedup for counting the number of primes between 50M and 100M as a function of the processors.*

Figure 3. When N = 100, 1000, 5000, 10000

Reason: When master is not able to divide integers evenly among the processes, we can see the spikes.

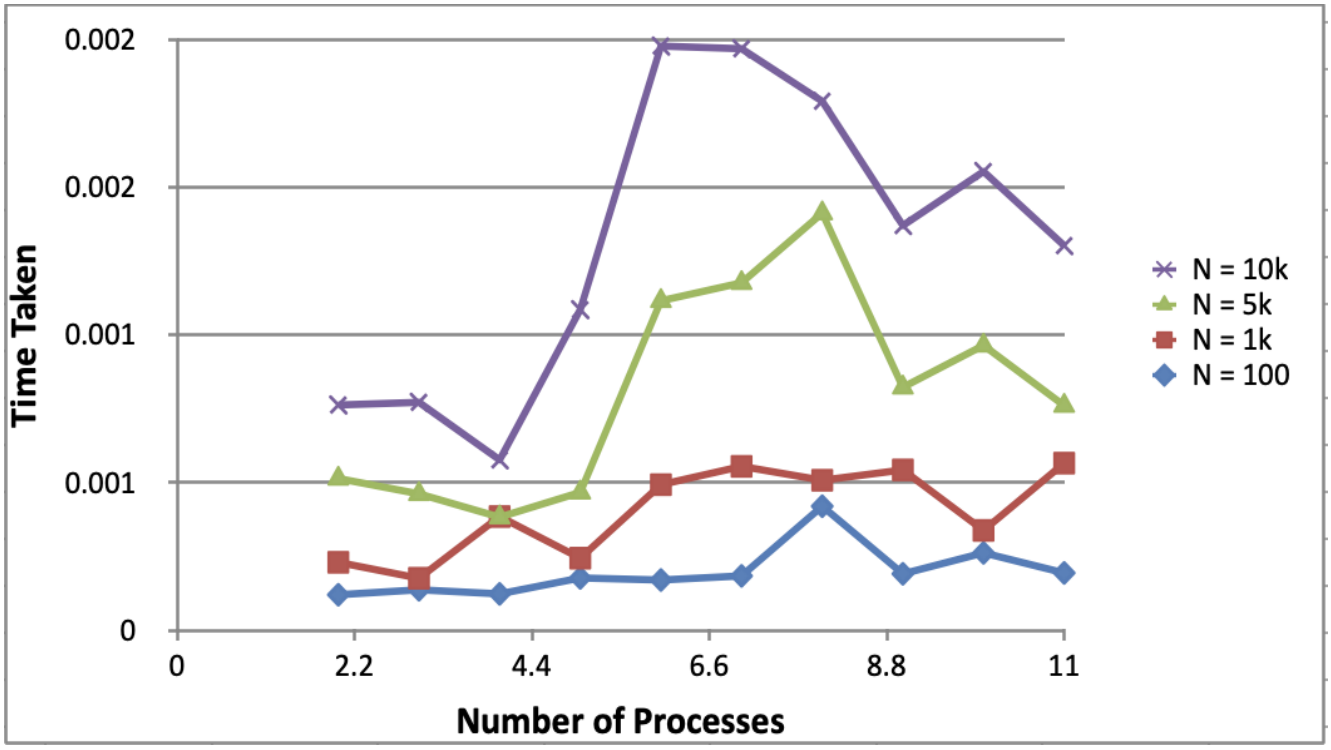


Figure 3. *Illustrates the time taken to run the number of processes(100 to 10k)*

Figure 4: When N = 50k, 100k, 500k, 1M.

Reason: When master is not able to divide integers evenly among the processes, we can see the spikes.

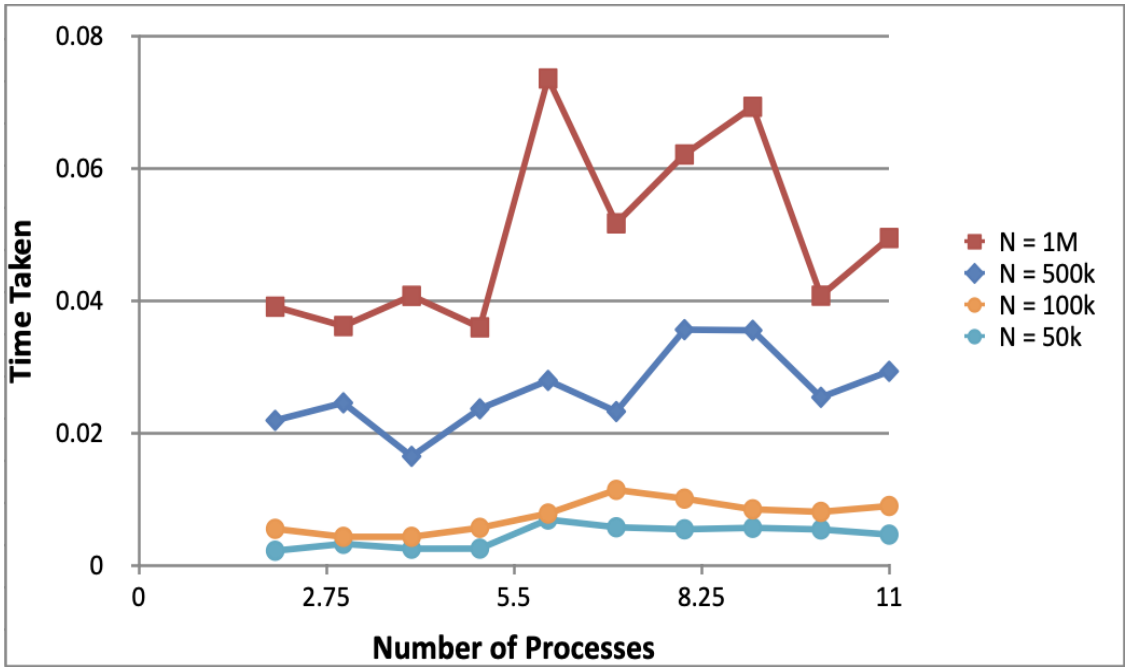
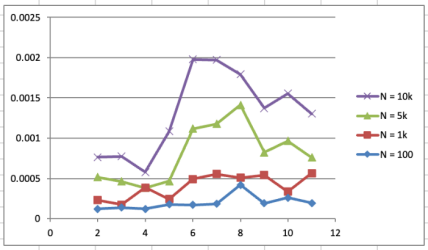
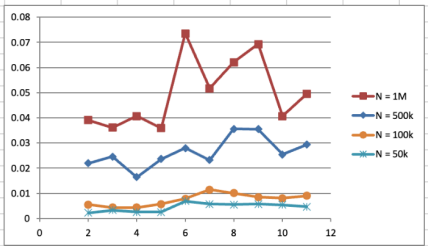


Figure 4. *Illustrates the time taken to run the number of processes(from 50k to 1M)*

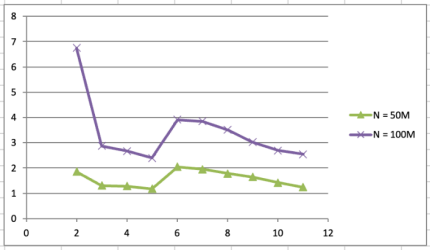
Performance Metrics:

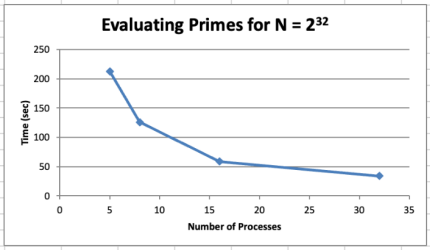
RESULTS		**Experimentally determined Processing Time (sec)
Processes	Last Integer	Proc Time
2	100	0.00012
3	100	0.000137
4	100	0.000123
5	100	0.000177
6	100	0.00017
7	100	0.000184
8	100	0.00042
9	100	0.000191
10	100	0.000262
11	100	0.000194
2	1000	0.00023
3	1000	0.000176
4	1000	0.000385
5	1000	0.000244
6	1000	0.000493
7	1000	0.000555
8	1000	0.000508
9	1000	0.000543
10	1000	0.000338
11	1000	0.000566
2	5000	0.000514
3	5000	0.000462
4	5000	0.000383
5	5000	0.000467
6	5000	0.001116
7	5000	0.001177
8	5000	0.001413
9	5000	0.000823
10	5000	0.000965
11	5000	0.000761
2	10000	0.000763
3	10000	0.000772



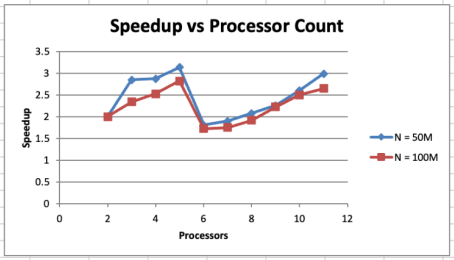


4	10000	0.000577
5	10000	0.001086
6	10000	0.001979
7	10000	0.00197
8	10000	0.001792
9	10000	0.001371
10	10000	0.001554
11	10000	0.001303
2	50000	0.002253
3	50000	0.003297
4	50000	0.002561
5	50000	0.002574
6	50000	0.006962
7	50000	0.005798
8	50000	0.005492
9	50000	0.005714
10	50000	0.00547
11	50000	0.004691
2	100000	0.00556
3	100000	0.00435
4	100000	0.004354
5	100000	0.005714
6	100000	0.00787
7	100000	0.01144
8	100000	0.010118
9	100000	0.008505
10	100000	0.008116
11	100000	0.009008
2	500000	0.021928
3	500000	0.024577
4	500000	0.016497
5	500000	0.023674
6	500000	0.027958
7	500000	0.023264
8	500000	0.035623
9	500000	0.035522
10	500000	0.025411
11	500000	0.029335
2	1000000	0.039084
3	1000000	0.036167
4	1000000	0.040726
5	1000000	0.035969





7	500000	0.023264	
8	500000	0.035623	
9	500000	0.035522	
10	500000	0.025411	
11	500000	0.029335	
2	1000000	0.039084	
3	1000000	0.036167	
4	1000000	0.040726	
5	1000000	0.035969	
6	1000000	0.073528	
7	1000000	0.051668	
8	1000000	0.062059	
9	1000000	0.069256	
10	1000000	0.040748	
11	1000000	0.04945	Speedup
2	50000000	1.852356	2
3	50000000	1.301563	2.846356
4	50000000	1.288671	2.874832
5	50000000	1.180848	3.137332
6	50000000	2.044529	1.812012
7	50000000	1.948326	1.901485
8	50000000	1.784538	2.076006
9	50000000	1.64123	2.257278
10	50000000	1.422718	2.603968
11	50000000	1.239454	2.988987
2	100000000	6.745209	2
3	100000000	2.877385	2.344215
4	100000000	2.672197	2.524218
5	100000000	2.391441	2.820563
6	100000000	3.913776	1.723453
7	100000000	3.851065	1.751518
8	100000000	3.513179	1.919973
9	100000000	3.026485	2.228727
10	100000000	2.69718	2.500838
11	100000000	2.548038	2.647217
5	**4294967296**	**212.999797**	
8	**4294967296**	**126.072248**	
16	**4294967296**	**58.842141**	
32	**4294967296**	**33.915731**	



Contributions:

First of all, we would all like to thank Professor John Gash for giving us the opportunity to use this challenge as an opportunity to showcase our skills. Each of us had the impression that we were working in an office setting. It was not an easy road for us. We ran across a lot of issues when constructing, integrating, and testing. We frequently met in person and over Zoom to discuss our issues.

Sujan Rao Chikkela :

- Implemented OpenMPI Master-worker architecture along with C client.
- Led the team in assisting them whenever they encountered blockers and architecting the application end to end.
- Actively participated in team meetings, resolving issues, and designing system architecture.
- Assisted in the development of project documents and PowerPoint presentations.

Akshay Madiwalar:

- Assisted in understanding and developing Master-worker architecture.
- Initiative to develop the OpenMPI service with a basic setup.
- Actively participated in team meetings and contributed to documentation.
- Actively participated in testing, integration, and resolving issues of the team.

Sakruthi Avirineni:

- Implemented OpenMPI and played a significant role in designing the architecture.
- Assisted in developing the Master-worker architecture for this project.
- Actively participated in designing PowerPoint presentations, testing, and resolving issues of the team.
- Took the initiative to derive the results in a graph and critical metrics for messages and time taken to broadcast to workers.

Arun Satvik Mallampalli:

- Spearheaded the team by setting up regular meetings and assigning tasks to team members.
- Handled integration for the entire project and developed a simplified adapter function.
- Assisted in OpenMPI Master-worker architecture and communication model.
- Actively participated in team meetings to resolve issues accordingly and co-participated in implementing C client.

Rajashekar Reddy Kommula:

- Implemented C client, which accepts messages and broadcast messages.
- Implemented C client for this project and actively participated in team meetings.
- Assisted in designing and implementing the architecture of this project.
- Assisted in OpenMPI Master-worker architecture and communication model.