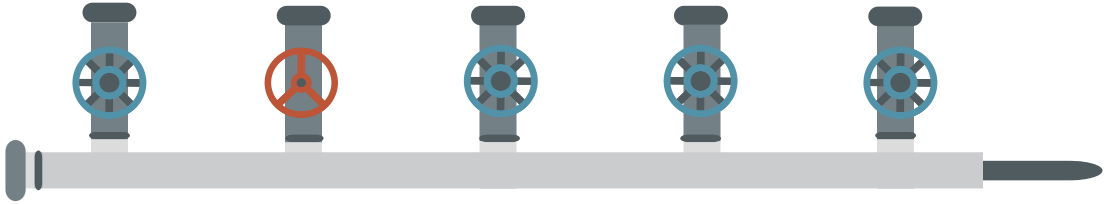
Grand Challenge - 2

CMPE - 275 Enterprise Application Development

Under Professor John Gash

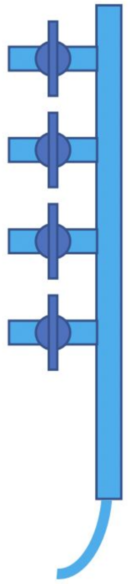


Team - Sprinters

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

Goal

- To implement the grand challenge one pipe architecture in C/C++ without compiling runtime with change in 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 its part and sends back the result to master.
- To achieve this goal, we have used OpenMPI to make parallel computation faster.



OpenMPI

- The Open MPI Project is an open source Message Passing Interface implementation that is developed and maintained by a consortium of academic, research, and industry partners.

- MPI is a standard library for performing parallel processing using a distributed memory model. MPI assigns an integer to each process beginning with 0 for the parent process and incrementing each time a new process is created.

- The message passing interface (MPI) is a standardized means of exchanging messages between multiple computers running a parallel program across distributed memory.

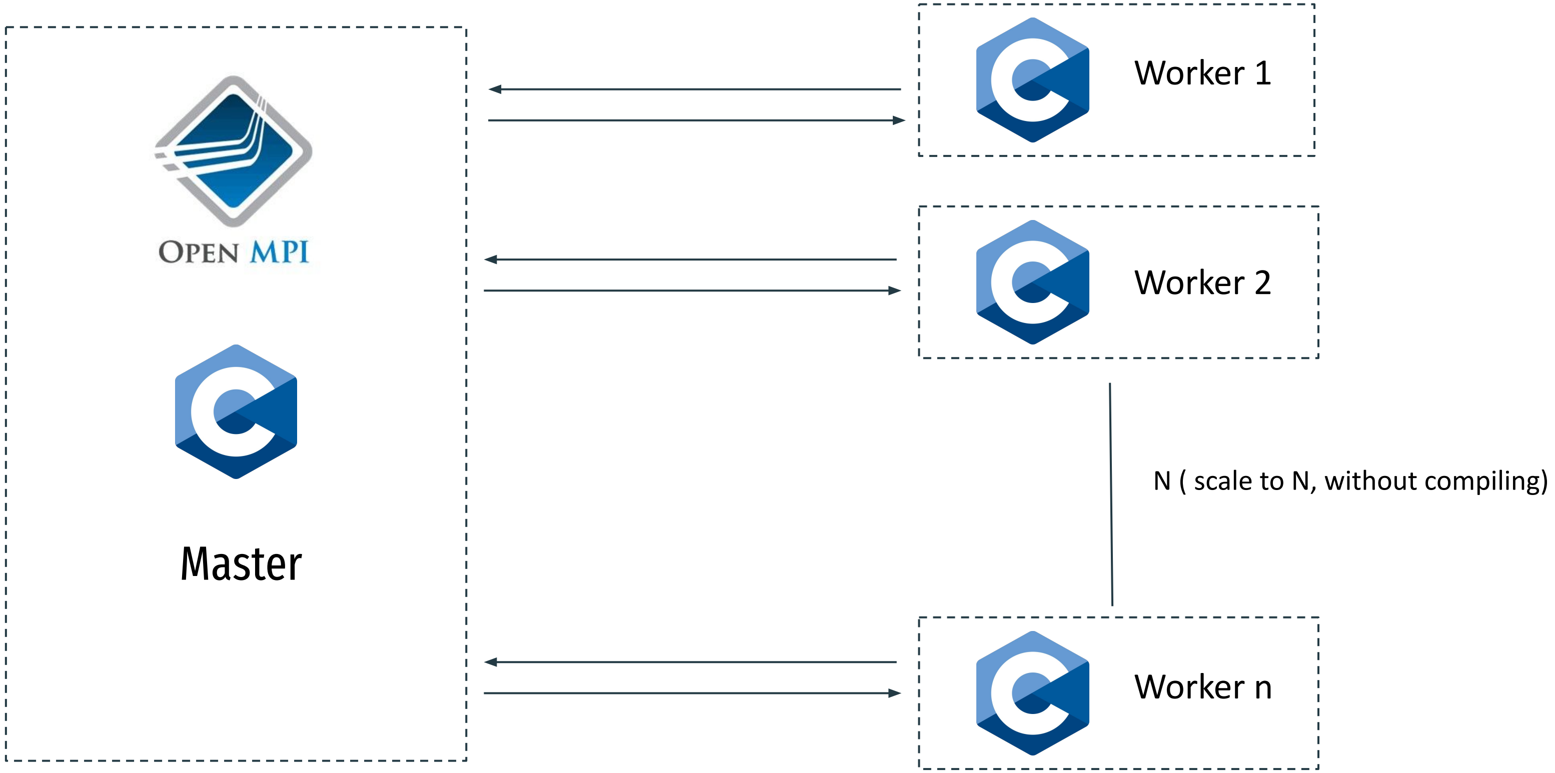
Use cases of MPI

- Parallel Computation: Each process computes its part and sends the work result to the master process. The processes communicate with each other to exchange data using the MPI point-to-point communication method
- Image Processing: Each worker process is responsible for processing its part of the image. The processes use two MPI communication methods to exchange data between them. The master process uses collective communication to send common information to work processes
- Parallel File Systems:The master process is responsible for managing a work queue of unexplored directories, equally distributing works between processes, and collecting work results from worker processes. The worker processes, on the other hand, are responsible for directory traversal.

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 primes numbers to the Nth integer.
- Max value of N = 2^32 (4,294,967,296)
- This task is cumbersome and takes lot of time with single process.
- 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 32 nodes computing parallely.

Architecture Diagram



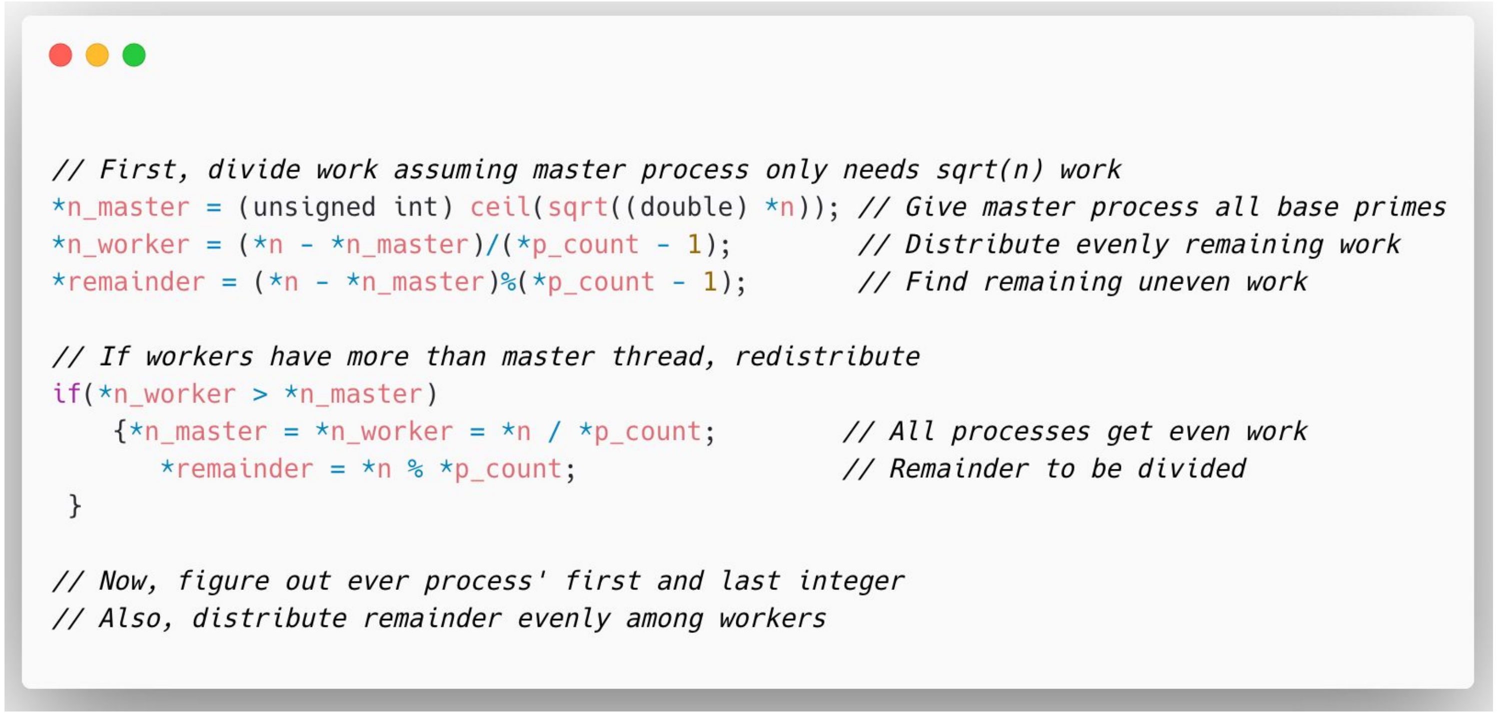
Approach

- First, master princess is given all integers up to sqrt(N).
- Then, the rest of the work is divided among other other processes.
- If other process still have more work than the master process, the work is re-divided to give back more work to master.
- Remainders are evenly distributed among other workers processors at the end.

Implementation

- Each MPI Processor determines which set of integers that process is responsible for, allocates and initializes an array for that set of integers. The master process will always have all integers up to sqrt(N) but can have more work if there are too few processors for a very large N.
- MPI functions:
    - MPI_Init : Starts MPI parallel execution environment.
    - MPI_Comm_rank: Determines the rank of calling process in the communicator.
    - MPI_comm_size: Determines the size of the group associated with the communicator
    - MPI_finalize: Terminates the MPI execution environment.

Implementation

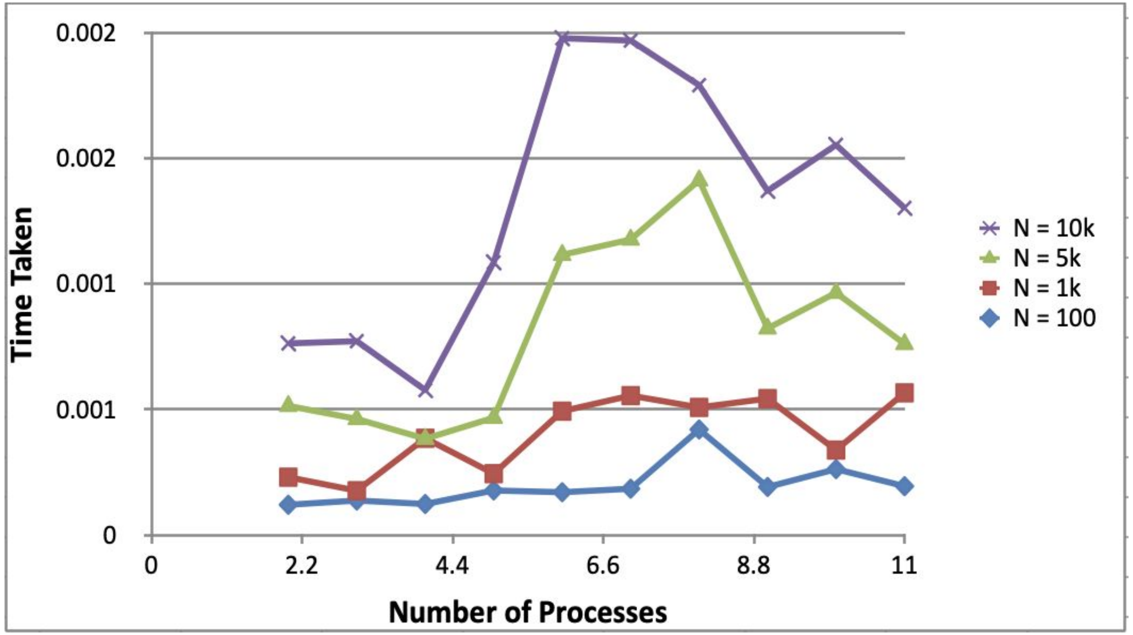


```
// First, divide work assuming master process only needs sqrt(n) work
*n_master = (unsigned int) ceil(sqrt((double) *n)); // Give master process all base primes
*n_worker = (*n - *n_master)/(*p_count - 1);        // Distribute evenly remaining work
*remainder = (*n - *n_master)%(*p_count - 1);  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

Results - Metrics

When N = 100, 1000, 5000, 10000

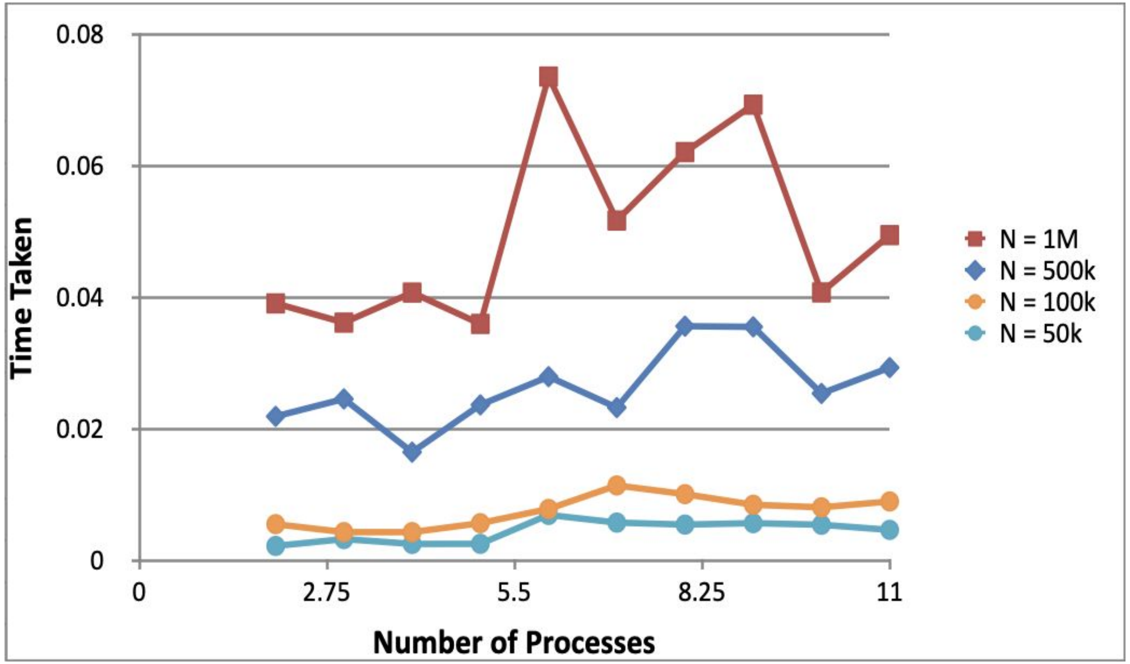
Reason: When master is not able to divide integers evenly among the processes we can see the spikes.



Results - Metrics

When N = 50k, 100k, 500k, 1M.

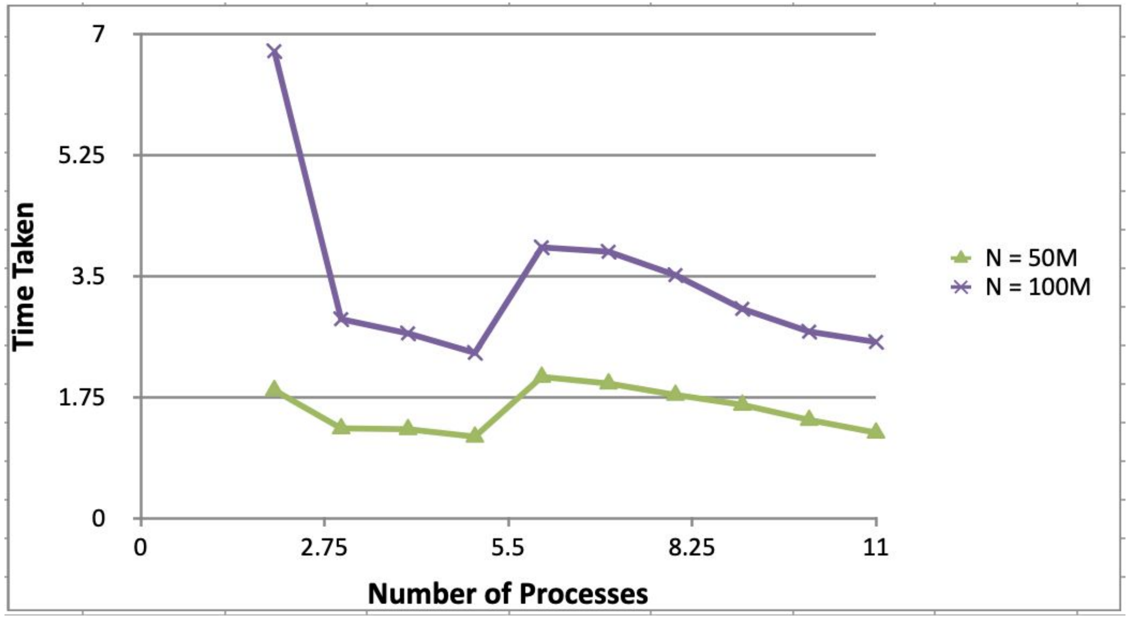
Reason: When master is not able to divide integers evenly among the processes we can see the spikes.



Results - Metrics

When N = 50M, 100M.

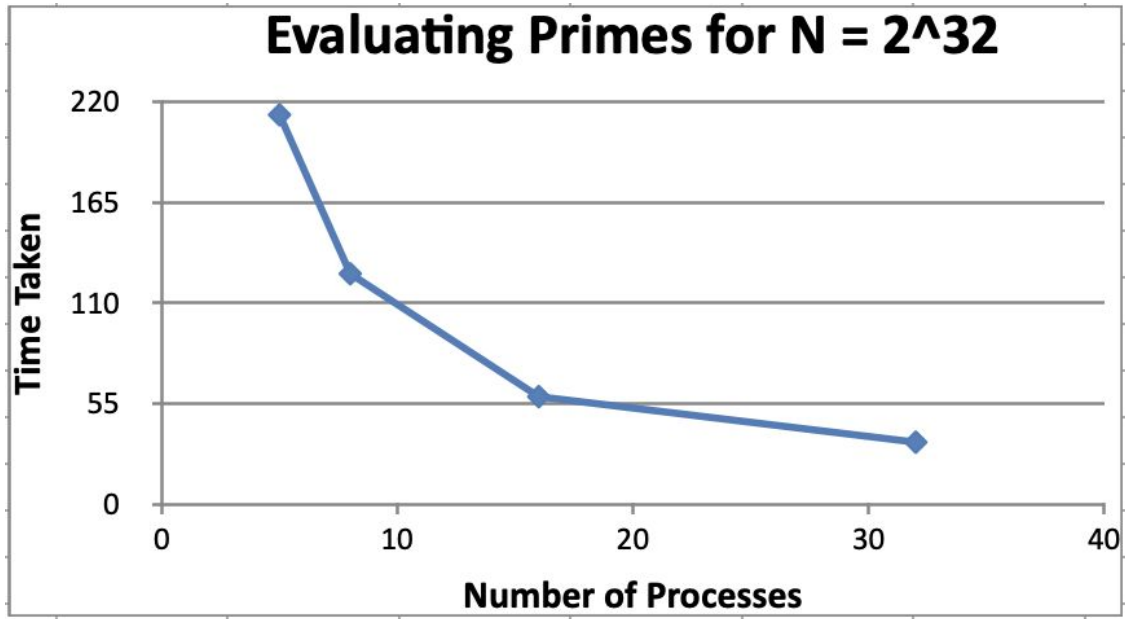
OpenMPI works better with higher workload.



Results - Metrics

When N = 2^32(4294967296).

OpenMPI works better with higher workload.



Our Thoughts

- The processes with higher rank got more work to do, and they ends their computation later than processes with lower ranks. So, when you are using lot processes, the total execution time is defined by the last process with highest rank.
- OpenMPI gives best results for large scale computations. It could literally reduce the time to seconds which might take several minutes with normal computation.

Thank You!

Any Questions?