1. **Problem Statement**

Using the Sieve of Eratosthenes concept for finding prime numbers, design and implement a program using the C programming language to find all primes numbers to the Nth integer with a max value of N=4,294,967,295. Implement 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.

1. **Approach**

The best way to approach this problem is by utilizing the software development process. This can be broken down into series of steps as follows:

1. Requirements
2. Analysis and Design
3. Implementation and Testing
4. Improvements and Delivery

In the requirements step, all the requirements necessary to this project were defined. This included determining why MPI/OpenMP was required, the upper bound required for evaluating prime numbers, the resources and tools available and the final output desired. In analysis and design, a detailed evaluation of the Sieve of Eratosthenes concept was performed. In this step, the abstract caveats of this concept were broken down. Then, in design, the various tasks were divided among the processes that will be created through MPI. The process flow was determined showing how processes interact and how work will be evenly divided.

Implementation and testing was further broken down into a series of steps. First, a basic MPI program was created testing the output for “Hello world”. Being able to successfully demonstrate that MPI can be implemented in the program code, compiled, and successfully run was critical to the success of this project. After demonstrating that this can be executed, a Makefile was created to streamline the compile and test steps. The implementation was documented through each step of the implementation process prior to a successful output in the attached document: ***Implementation\_Documentation.docx***.

After demonstrating that MPI can be utilized, further implementation of the design was made. First, the debug mode and upper limit for evaluating prime numbers, N, was made accessible to the command line as arguments. This allowed for flexible testing and enabled the Makefile to define a list of test cases for testing performance. Then, all the housekeeping variables, e.g. process rank, total number of processes, etc., must be determined. Keeping in mind that all processes will be getting a copy of the program, process-sensitive operations must be carefully defined, such as those for process 0, the master process.

After housekeeping variables have been defined, the total workload must be evenly distributed. The logic this 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 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

Using this process flow, the work will always been as evenly distributed as possible where, in most cases, all processes will have exactly the same amount of work within one integer. The only extreme cases are were the problem size is very small for the number or processes, where the master process will hold more integers because it MUST hold all integers up to the square root of N to ensure it broadcasts all primes for sieving.

After distributing the workload, each process can now allocate space for a character array comparable to the number of integers that process is responsible for. Each element will be initialized to ‘0’, denoting that integer is potentially a prime number. All processes also check for the success of this memory allocation prior to initializing all elements. In addition, the master process must initialize its first element, 1, to ‘1’, denoting a non-prime number. Once this is completed, the sieving process can begin.

In the sieving process, the master process has a special job which is broadcasting the next prime number. This number is initially assigned the value 2. Every process must go through its respective array, including the master process, and mark all integers that are multiples of the current prime. This is done by first calculating the index of the first multiple found in this process’ array, assuming the multiple is in the array. Then, with that index, it is checked against the bounds of the array, and if within the bounds of array, it is marked in a loop, adding the current prime to this value until the end of the array is met. Each time this is completed for the master process, it checks for the next unmarked integer and broadcasts it. All processes do this until the integer broadcasted is the value of the master set’s last integer signaling the end of the process.

Once this is completed, all processes then count the number of unmarked elements, and through MPI\_Reduce, send this information back to the master process. The master process, at this point, can report the number and successfully end the program. Debug statements also allow printing these values to the console for the user to view. This process involves all processes greater than 0 to wait in a blocking command, MPI\_Recv, while the master process prints to the screen before sending, MPI\_Send to process 1. Then, consecutively, each process follows that routine.

The timer for this process is set prior to MPI housekeeping and work distribution and checked after the total number of primes is determined. To further improve this process, OpenMP was also evaluated in two places: (1) in the for-loop to mark all multiples that are not primes and (2) in the for-loop for summing up all primes found. This can be done by designating a parallel region spanning both areas and simple adding “pragma omp for” directive above each loop.

1. **Solution**

Through this process, a successful implementation of the Sieve of Eratosthenes was created. The source file, ***sieve\_of\_erath.c***, and its corresponding ***Makefile*** can be found in the ***src*** folder attached. All results, including raw output, for this project can also be found in the attached document: ***TestResults.xlsx***. The MPI executable, ***sieve\_of\_erath****,* and the MPI + OMP executable, ***sieve\_of\_erath\_omp***, can be created and tested using the following commands:

**make**  # makes both executables

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

**make test** # runs all tests trials on program with

# n = [100, 100,000,000] and processes = [2, 11]

**mpirun –n 4 ./ *sieve\_of\_erath*\_omp 100 DEBUG** # runs executable with OMP directives in

debug mode with n = 100

Because of connection latency and buffering, print statements in DEBUG MODE may be out of order, but all values can be seen and the total prime count correctly printed. Figure 1 illustrates the speedup using multiple processors at the upper limit (N=4,294,967,295).

**Figure 1.** Processing time as a function of the number of processes

While hundreds of runs were performed, many of the runs were the values set for N were low exhibited more noise in the results. Runs with higher N values exhibited more clear patterns as shown below in Figure 2.

**Figure 2.** Speedup as a function of the number of processors for determining the number of primes between 50,000,000 and 100,000,000.

This pattern was evident across all runs. As the number of processors increase, the performance increased accordingly until roughly 4-5 processors where performance took a dip. This may be attributed to the experiments being performed on 4 nodes. Increasing the number of processes continues the trend upwards in speedup.

Continuous testing was performed throughout the development process to allow continuous improvements to the code. In the final stages, OpenMP was also introduced to the for-loop where each process marks multiples and the final for-loop where primes were counted. Adding the omp pragma to the first loop slowed down the process significantly. Originally, this was attributed to the omp parallel for loop continuously spinning up threads via forking and then joining. Moving the omp parallel directive outside the do-while loop, which prevents the repeated forking-joining solves this. However, even then, there were minimal improvements in speedup. The optimal solution which also utilized OpenMP was keeping it primarily in the final prime counting routine. However, even applying the directive here gave minimal yields in performance.

Overall, the final source code and corresponding executable was impressive allowing the calculation for the number of primes between 1 and 4,294,967,295 in roughly only 33 seconds utilizing 4-8 nodes.