**Finding Prime Numbers - Sieve of Eratosthenes**

I used the Sieve of Eratosthenes algorithm which takes ***O(n log log n) and O(√n)*** space to find all the primes from 1 to 10^8. This algorithm works by taking advantage of the CPU’s L1 near perfect hit and miss rate by using the cache register on each core as the sieve array which has a max 64kbs. Now using the sieve start marking all the multiples of each prime, starting from 2, as non-prime. The remaining numbers that are not marked are primes. For example the same algorithm to find primes between 1 and 20 :

* Write out the numbers to 20
* Get ***i*** (smallest number, start), 0 and 1 are not primes so the next smallest number is 2. Now Circle it as it is the smallest.
* For each number bigger than ***i***, if the number is a multiple of ***i***, put a cross on it for its not a prime. In this case ***i***=2
* Get the next ***i*** (the next smallest number in this case is 3) circle it and repeat process C.
* The Algorithm repeats process D until the **end,** thenonly prime numbers will be left. this case 2,3,5,7,11,13,17

For this particular algorithm I divide the range [1, 10^8] into 8 equal subranges, each of size 12.5 million, then assign each subrange to a separate thread. and let each thread run the sieve algorithm on its own subrange. This is basically an equivalent to having 8 mailmen deliver mails to a block with 100 houses which would be both efficient and time effective than leaving the Job to one Mailman. Similarly, The algorithm delivers each thread for processing to a different core, in this case assuming there are 8 cores.

On the main Sieve function I use the segmented sieve method to optimize the memory usage and the performance of the sieve algorithm. This technique works by dividing each subrange into smaller segments, and processing each segment one by one.

After all the threads finish their work, the helper function gets their sum and the largest 10 primes. To find and store the Max 10 primes of, I used a priority queue to push and pop the items without extra memory cost.

The output is written to a file named primes.txt. The output consists of four lines: the execution time in seconds, the total number of primes found, the sum of all primes found, and a list of the largest 10 primes.

**Correctness and Efficiency**

- The correctness of the sieve algorithm is based on the fact that every composite number has a prime factor that is less than or equal to its square root. Therefore, by marking all the multiples of each prime up to the square root of the limit, we can eliminate all the composite numbers and leave only the primes.

- The segmented sieve technique is based on the fact that every segment [low, high] is a part of the original range [1, limit], and the primes in each segment are the same as the primes in the original range. Therefore, by applying the sieve algorithm to each segment, we can find all the primes in the original range.jk

- In multithreading, each thread handles a specific part of the task, dividing the work evenly among them. This is similar to how people divide tasks to work faster and more efficiently. By assigning small portions of work to different CPU cores, the multithreading approach can significantly speed up computations. For example, using 8 threads can make the process 8 times faster compared to using just one thread. However, it's important to note that increasing the number of threads may not always lead to increased efficiency, as it could also raise the overall computation cost.

**Experimental Evaluation**

I implemented my approach in C++ and ran on a machine with 5 cores and 8 GB of RAM, using the g++ compiler with the -O3 optimization flag..

- The output of my program is: primes.txt

—-------

Execution Time: 1129 Milliseconds

Total Primes: 5761455

Sum of Primes: 279212384583680.000000

Top Ten Maximum Primes: 99999787, 99999821, 99999827, 99999839, 99999847, 99999931, 99999941, 99999959, 99999971, 99999989,

—-------

- The Average runtime for the algorithm was 1.12 seconds to find 5,761,455 primes, and the sum of all primes is 279,212,384,583,680

- I also compared my program with a single-threaded version that uses the same algorithm and technique, but without multithreading. The output of the single-threaded version is:

—-------

Execution Time: 3050 Milliseconds

Total Primes: 5761455

Sum of Primes: 279212384583680.000000

Top Ten Maximum Primes: 99999787, 99999821, 99999827, 99999839, 99999847, 99999931, 99999941, 99999959, 99999971, 99999989,

—-------

- This means that the single-threaded version took 3.05 seconds about twice the time to find the same number and sum of primes.. Therefore, my program is about 2 times faster than the single-threaded version, which shows the benefit of multithreading.