COP 6616 **Fall 2025**
Programming Assignment #1 Ryan Harding

1. **Summary of Approach:**
The way my program is structured is with two functions, a main function and a function titled `par_prime_finder`. The multithreaded approach to finding primes is done within the latter, while the former handles the execution time, calling `par_prime_finder`, and the output. Execution time is calculated by a start time before the call to `par_prime_finder` and then the elapsed time after `par_prime_finder` returns. I used Rust's standard library `std::time::Instant` struct for the execution time, and the `std::fs::File` struct and `std::io::Write` traits for file I/O. For smoother error-handling and safety, I also used the `std::io::Result` and `std::io::Error` structs. I also kept around a legacy `seq_prime_finder` function from a sequential prime algorithm. This algorithm provided the `sequential.txt` output file, which the parallel output `output.txt` was compared to in testing for correctness.

   The way the `par_prime_finder` function works is a multi-threaded modification of the traditional Sieve of Eratosthenes. That algorithm finds all the primes up to a certain value by finding primes up to the square root of that value and marking all of their multiples. I will explain the correctness of this algorithm in section 2. While this algorithm is extremely simple, it can be considered inefficient at large values because every additional **base prime** (prime below the square root of the limit) creates multiple multiples that must be addressed The traditional algorithm is as follows:

   Listing 1: Traditional Sieve Algorithm

```
function sieve(some limit n > 1):
    let A be a boolean array 0..=n, initial values set to true
    set A[0] and A[1] := false
    for i = 2..=sqrt{n} do:
        if A[i] == true then
            for j=i^2..=n incremented by i do
                set A[j] := false

    collect and return all i such that A[i] == true

```

   The most taxing part of the algorithm is the exponential growth of the amount of primes that must be dealt with after the square root of the limit. So, for my parallel approach, that was the part I decided to parallelize. While a traditional sieve is still used to find and mark all primes up to the square root, I used Rust's standard `std::thread` library to parallelize the approach to marking the rest of the range by splitting the rest of the range into 8 equal chunks, and then spawning eight threads to each deal with marking all multiples of the base primes within each chunk as non-prime. Every base prime should have a roughly equal amount of multiples within each chunk, so the workload should be equal across all threads. At the end of the program I combine the initial base sieve with all the chunks returned by each thread, and then collect the marked primes. The modified algorithm is as follows:

Listing 2: Parallel Sieve Algorithm

```
function parallel_sieve(some limit n > 1, some thread count t):
    let A be a boolean array 0..=sqrt(n), initial values set to true
    set A[0] and A[1] := false
    for i = 2..=sqrt{n} do:
        if A[i] == true then
            for j=i^2..=sqrt(n) incremented by i do
                set A[j] := false

    collect all i such that A[i] == true and store in array B
    set a chunk size to n - sqrt(n) ceiling divided into t equal chunks

    for i = 0..t do:
        set start value to sqrt(n) + 1 + i multiplied by the chunk size
        spawn thread i to:
            create true boolean array A_i 0..chunk size
            for every base prime p in B do:
                set multiple to p^2
                while multiple is within the chunk do:
                    A_i[multiple - start value] := false
                    multiple increment by p

            return A_i

    for i=0..t do:
        join thread i
        append A_i to the end of A

    collect and return all i such that A[i] == true
```

2. **Correctness and Efficiency:**

The design's correctness can be assured by the base assumptions behind the Sieve of Eratosthenes algorithm: That no primes greater than the square root of n will have multiples less than or equal to n, ergo all numbers between the square root of n and n will either be prime numbers or multiples of prime numbers less than or equal to the square root n. To prove this, observe the following:

$$\text{Assume the lowest possible prime ¿ } \sqrt{n}\text{: } \sqrt{n}+1 \tag{1}$$

$$\text{Every multiple } x*(\sqrt{n}+1) \text{ for } x=0..\sqrt{n} \;\; \textbf{Will have already been counted by multiples of primes ¡ } \sqrt{n} \tag{2}$$

$$\text{Thus the first multiple to check would be: } \sqrt{n}*(\sqrt{n}+1)=n+\sqrt{n} \tag{3}$$

$$\textbf{By Definition: } n+\sqrt{n} \textbf{ Exceeds n and therefore it and all larger multiples are not within the sieve.} \tag{4}$$

Thus, all numbers in the set up to any limit will be either prime numbers, or multiples of some prime number less than or equal to the square root of n, or 1 or 0. The Sieve of Eratosthenes marks 0, 1, and all multiples of prime numbers less than or equal to the square root of n as non-prime, and by definition all remaining numbers are prime. As my parallel algorithm does not stray from this fundamental logic, correctness can be expanded to it.

Redundant or unnecessary computations are avoided by the sieve due to starting multiple checks always from the square of the prime being multiplied. Multiples prior would be that prime multiplied by some lower value, which would already have had its multiples checked. Therefore only multiples starting from the square of the prime must be checked. Furthermore, redundancy is avoided in the parallel design by starting the parallel design from the square root of n, as all primes before then have already been checked, and by only dealing with multiples within each chunk per thread.

Redundancy was also avoided with an optimization I implemented on top of the base algorithm from Section 1 to always initialize multiples from the first multiple within the chunk, rather than the square and then iterating up to multiples in the chunk. I will describe this optimization in Section 5.

In terms of balancing workload, I chose the simple approach of 8 equal-sized chunks of the range from the square root of n up to n. The base primes should have a roughly number equal of multiples to mark off per chunk, therefore there should be a roughly equal amount of calculations overall per chunk. Each chunk and thread is also completely independent and has no locking or waiting to contend with. The theoretical maximum reduction in execution time therefore would be down to $\frac{1}{8}$th the time that the execution of that portion of the program took sequentially. However, the first $\sqrt{n}$ values of the range are still calculated sequentially, therefore the overall execution time will end up being roughly the square root of the original execution added by $\frac{1}{8}*(time(n)-time(\sqrt{n}))$. Future experimentation will revolve around attempting to parallelize that first sieve up to the square root.

3. **Experimental Evaluation (On ARM MacBook Pro):**

   - **Total Execution Time:**
     - Sequential: 3.6669 seconds best trial, always within 3.6-3.7 second range.
     - Parallel: 1.6121 seconds best trial, always in the 1.6 second range.
   - **Number of Primes Found:** 5761455
   - **Sum of Primes Found:** 279209790387276
   - **Observations about Thread Performance:** I observed no obvious case wherein threads would have unequal workloads or unnecessary bottlenecks against each other. Each thread uses its own space due to Rust's ownership system, and every base prime should have roughly the same amount of multiples per chunk. The only case that could differ from that would be a case where a large base prime might not have any multiples in a lower chunk, which would be a risk if thread count increased and there were more chunks. However, at the values we have, that isn't the case. The highest base prime, as in highest prime below the square root of 10^8, is 9973, whose lowest multiple in any chunk would be 19946. The chunk size is the range between 10^8 and its square root divided into 8 equal chunks, which leaves a chunk size of 12498750. Thus we can prove the first multiple of the largest base prime IS within the first chunk:

$$\begin{aligned}\sqrt{10^8} &<= 2(9973) <= \sqrt{10^8} + 12498750\\ 10000 &<= 19946 <= 10000 + 12498750\\ 10000 &<= 19946 <= 12508750\end{aligned}$$

     For the first chunk to become imbalanced by having no multiples of the largest base prime, there would need to be around 10055 threads to push 19946 out of the first chunk. And for reasons I'll discuss soon, that is already an impractical design even if it were possible. Beyond that, the only true threats to thread performance and efficiency are the space complexity (Due to Rust's thread ownership system, every thread MUST have its own clone of the base primes) and the fact that the initial square root sieve is still sequential.
   - **Comparison of Performance:**
     - Speedup: Using the best trials I observed, the parallel algorithm was roughly $\frac{3.6669}{1.6121} = 2.2746x$ faster than the sequential algorithm. For the values we have, that doesn't seem like much, but as N increases, that speedup would become far more impactful.
     - There doesn't seem to be any diminishing returns or overhead caused by the approach.
   - **Scalability:**
     - The factors that affect scalability would be the thread count and resulting chunk size. As I've already discussed, thread count can eventually reach a point where not only would workload become imbalanced, but also to a point where it may create diminishing returns as a result. Especially if the chunk size becomes lower than the square root of N, at which point the sequential sieve dominates the runtime.
     - At sufficiently large N, despite a speedup, the algorithm may still not be perfect as the sequential sieve at the start will begin to take significantly more time. At low N, the overhead of starting and joining threads may be costlier than running a simple sieve.
     - Less threads would create less speedup, more would create more but as the count increased the degree of speedup added would lessen further and further until flattening out as chunk size became lower than the initial sieve.

4. **Thread Design and Load Balancing:**

    - Technically speaking, the input range (0 to 10^8) is partitioned into nine sections. The first is the base sieve, 0 to $\sqrt{10^8} = 10^4$, which is used during the initial sequential part of the program. The next eight are equal-sized chunks of the remaining $10^8 - 10^4$ range.
    - As I used Rust's standard thread libraries, scheduling is handled 1:1 with the OS, which is Dynamic for most modern OS'.
    - I didn't struggle in balancing execution time across threads since I could assume each chunk would have a roughly equal amount of multiples of the base primes as any other chunk. The program also runs so fast that attempting to micro-manage thread timing would've been impractical and difficult.

5. **Prime Finding Algorithm:**

    - I chose the Sieve of Eratosthenes due to familiarity and simplicity. My focus was purely on creating a workable parallel solution, not reinventing the wheel or building a new solution from the ground-up. The Sieve of Eratosthenes is an algorithm I'm very familiar with and know how to code effectively, so it's what I stuck with.
    - The time complexity for the sequential Sieve of Eratosthenes is $O(n \log \log n)$. The parallel version, despite running fast, has the same time complexity. Fundamentally the algorithm is still the same, and both the sequential and each prime part run in separate variations of $O(n \log \log n)$ time. Something closer to $\sqrt{n} \log \log \sqrt{n} + \frac{n-\sqrt{n}}{8} \log \log \frac{n-\sqrt{n}}{8}$ time, but still roughly some sort of $O(n \log \log n)$ time.
    - Outside of the traditional sieve optimizations, I did implement an additional optimization, which I mentioned in Section 2. That was ensuring that I started checking multiples in the chunk rather than working up to them from the square of each prime. I did this by checking if the square of the prime was after the start of the chunk, and if not, I instead used ceiling division of the chunk start by the prime to find the first multiple after the start point. The formula is as follows:

$$ceil(\frac{start}{prime}) = \frac{start + prime - 1}{prime}$$

6. **Reflection:**

    - Working on this, I learned how to work with concurrency in Rust, which I have never done. I also learned how to code a concurrent algorithm without shared variables while still combining outputs together.
    - My biggest goal for a future version would be to somehow code a completely parallel version of this, without the initial sequential sieve. In that case, I wouldn't be able to use the fundamental assumption of the Sieve of Eratosthenes (No base primes over the square root of N), because I'd be working in that range in parallel, without a pre-existing base primes set. An idea I had is to parallelize the marking of multiples within the sieve itself (E.g. each thread takes a base prime and marks off all their multiples and then takes the next base prime when they're done). But doing so, especially in Rust, would be a challenge due to both ownership and potential thread conflicts.
    - I did use ChatGPT towards the very beginning of the assignment as a brainstorming tool to figure out how I would structure this. It provided a (not very well-written) algorithm using the Rayon library to automate the handling of threads. As that would be incongruent with the project as well as have its own challenges due to how automated it was, I coded the thread logic in Rust's libraries by hand without AI. In the process I also realized several flaws with the algorithm ChatGPT output to begin with (E.g. their algorithm ended up redundantly checking the first $\sqrt{N}$ elements in the parallel stage despite having already done that) and kept those issues in mind. The pieces I ended up taking from the GPT algorithm were some semantics like the filters to collect the primes, and some basic ideas for how to implement the algorithm.