Assignment 1: Report

Name: **Lukas Hinterleitner**
ID: **lukash24**
Matriculation number: **51912219**

Based on your solution and understanding of the assignment and lecture slides, fill out the questions below:

1. How are master and worker threads spawned? Briefly explain if they work together, how they synchronize, and what C++ features did you use to make this work. How many CPU cores are available on an Alma cluster node? How many threads can you use, and how many are used for the performance measurements?

   - The master thread is the main thread executing the `main` function. It spawns the producer as an asynchronous task using `std::async`.
   - Worker threads are spawned using a `std::vector<std::thread>` called `workers`. The `worker` function is passed as an argument to the `std::thread` constructor, and each thread is stored in the `workers` vector.
   - They work together by sharing a thread-safe queue `SafeQ<int> q`. The producer reads numbers from a file and pushes them into the queue, while worker threads pop numbers from the queue and perform processing.
   - Synchronization is achieved using mutexes, condition variables, and futures. Mutexes are used to protect shared resources, condition variables are used to make threads wait for specific conditions, and futures are used to retrieve the result of the producer's work.
   - The alma cluster of the University of Vienna has six computing nodes. Each of them has two 8-core Intel Xeon CPU’s built in. Hence, each node has 16 physical cores and therefore, we can use 32 threads per node.

2. How is the work distributed among the threads? Did you use static or dynamic work distribution? Is one performing better than the other and why?

   Work is distributed among the threads dynamically. The producer reads numbers from the file and adds them to the queue. Worker threads fetch numbers from the queue and process them independently. Dynamic work distribution is more suitable in this

case since we don’t know how much data we have to process. However, I think static work allocation here will be faster since we don’t need blocking mutexes used in the queue. However, this only holds when we assume that producer also performs well sequentially or is parallelized as well.

3. What accesses needed to be protected with locks to ensure safe accesses?

   - Accesses to the shared queue `SafeQ<int> q` in the `push`, `pop`, `wait_and_pop`, `size`, and `empty` methods.
   - Accesses to shared variables `primes`, `nonprimes`, `sum`, `consumed_count`, and `number_counts` in the `worker` function.

4. In the version where you needed to use atomic variables, which variables needed to be used as atomic variables? Are there any variables where atomic types were not suitable?

   In the atomic version, I used atomic variables for `primes`, `nonprimes`, `sum`, and `consumed_count`. Atomic was not suitable for `number_counts` because it's a vector, and atomic operations on vectors are not supported in C++ standard libraries. Therefore, I created a vector of integer atomics.

5. Briefly explain how you implemented the SafeQ so that multiple threads can access it safely? Which accesses needed to be protected for this to work and why? In which methods, and why not in others? How did you make threads wait for more items?

   - Mutexes are used to protect access to the underlying `std::queue` in the `push`, `pop`, `wait_and_pop`, `size`, and `empty` methods.
   - The `std::condition_variable` is used in the `wait_and_pop` method to make threads wait for new items to be added to the queue and is invoked in the push method.
   - Other methods don't require condition variables as they don't need to wait for specific conditions to be met.
   - Additionally, I added the function try_pop which additionally returns true/false on top of the already implemented pop function when dealing atomics.

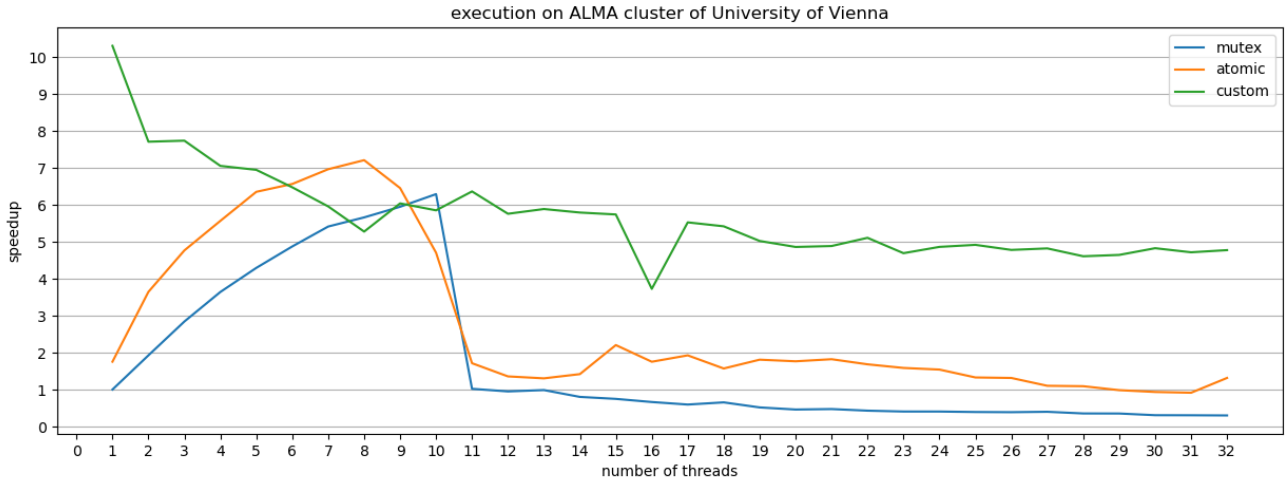
6. Where do threads need to synchronize in the code so that results are correct?

- When accessing the shared queue in the `SafeQ` class methods.
- When updating shared variables in the worker function.

7. Are there any bottlenecks/performance issues in your code? Which synchronization parts/mechanisms are causing the most overheads and why? What was the best-performing version? Did you overcome the performance issues and how? Is there any relation to the memory and caches?

   - Synchronization overhead from using mutexes and condition variables.
   - Contention when accessing shared resources (queue and shared variables).
   - The best-performing version depends on the specific hardware and workload. Optimizations can include reducing contention, using atomics where possible, and optimizing memory access patterns.

8. Include a speedup graph (as shown in the slides) showing the speedup on ALMA for all provided versions. Use horizontal axis for the number of threads (1-32) and the vertical axis for speedup (not the execution time!).



9. Include a table with execution times and speedup of the parallel code. Speedup is measured compared to the sequential version (~10 seconds on ALMA nodes). If you have these in an Excel sheet, you can just attach it with your submission.

   These are stored in CSV-files and have the following naming convention:

   speedup-execution-time-*xxxxxxx*.csv

10. Additionally, I added a third custom solution that uses a thread safe unordered map to cache all the numbers and if it is a prime number or not. It uses atomics for synchronization between workers.