Assignment 1: Report

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 not explicitly spawned. Instead, it is the default thread of execution that starts when the program begins running. When the program is launched, the master thread executes the **main()** function.

   The master thread acts as a producer, while the worker threads act as consumers.

   The master thread starts by creating a **SafeQ** object to store the numbers from the file. It spawns a thread for the producer function. The producer reads the numbers and pushes them into the **SafeQ**. Once all the numbers are read, the producer pushes sentinel values into the queue to signal that there are no more numbers to process.

   After the producer, the master thread spawns worker threads and adds them to the **workers'** vector. Each worker thread runs the **worker()** function, which retrieves numbers from the **SafeQ** and processes them. The worker threads use **wait_and_pop()** method to wait for new numbers retrieve. When a sentinel value (-1) is pulled from the queue, it stops processing and exits its loop.

   The **SafeQ** object uses a mutex and condition variable to ensure that the worker threads can safely access and modify the queue without any data race. The workers also use a separate mutex to synchronize updates to shared variables like **primes**, **nonprimes**, **sum**, **consumed_count,** and **number_counts**.

   Once all workers are spawned, the master thread waits for them to complete their work using **join()**. This ensures that the main thread does not proceed until all workers have finished processing.

   C++ features:

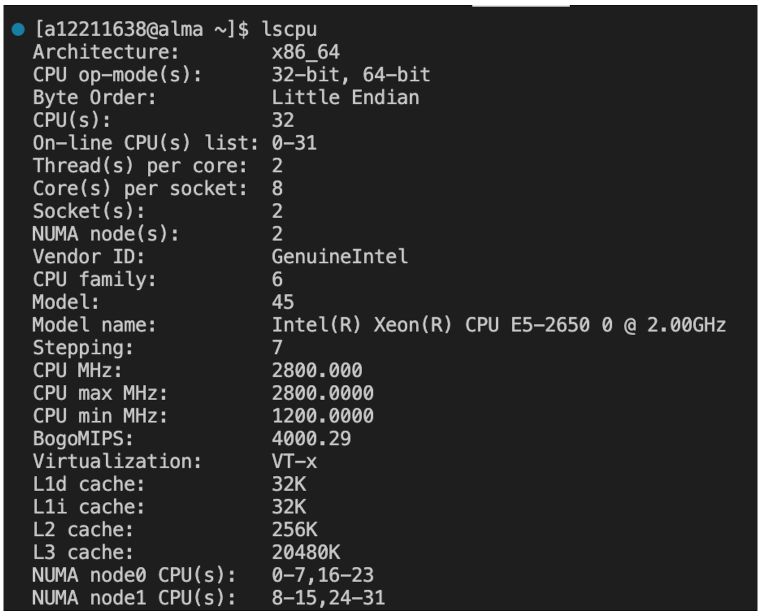
   - **Mutexes:** used to protect shared resources from data races and ensure exclusive access. In this assignment, mutex was used to modify the queue without data race and synchronize updates to shared variables.
   - **Condition Variables:** used for thread synchronization. **cv** was used for coordination between the producer and workers. This allows workers to wait for new data to be available in the **SafeQ**.

- **Smart Pointers: shared_ptr** was used to return a smart pointer to the retrieved number, that provides shared ownership of the number, from the **SafeQ** in **wait_and_pop()**.
- **Futures and Async:** used for asynchronous execution. In this code, **async** was used to execute the producer function concurrently with the main thread. f**uture** was used to retrieve produced_count when the producer is finished.
- **ref:** used to pass references to functions like producer and worker. This helps to ensure that the functions receive the actual shared data.

Used in atomic version:

- **atomic and atomic_flag:** used for synchronizing access to the shared data instead of mutexes. **atomic** was used to create atomic variables for some shared variables. **atomic_flag** was used to implement a spinlock for accessing the queue.
- **this_thread::yield:** used to make the thread relinquish the processor to allow other threads to run.
- **fetch_add:** used to do atomic addition.
- **memory_order_relaxed:** was used to indicate that the operation does not need to enforce ordering constraints.

With **lscpu,** we can access the ALMA CPU information.



```
[a12211638@alma ~]$ lscpu
Architecture:        x86_64
CPU op-mode(s):      32-bit, 64-bit
Byte Order:          Little Endian
CPU(s):              32
On-line CPU(s) list: 0-31
Thread(s) per core:  2
Core(s) per socket:  8
Socket(s):           2
NUMA node(s):        2
Vendor ID:           GenuineIntel
CPU family:          6
Model:               45
Model name:          Intel(R) Xeon(R) CPU E5-2650 0 @ 2.00GHz
Stepping:            7
CPU MHz:             2800.000
CPU max MHz:         2800.0000
CPU min MHz:         1200.0000
BogoMIPS:            4000.29
Virtualization:      VT-x
L1d cache:           32K
L1i cache:           32K
L2 cache:            256K
L3 cache:            20480K
NUMA node0 CPU(s):   0-7,16-23
NUMA node1 CPU(s):   8-15,24-31
```

There are 32 CPU cores available on ALMA with 2 threads per core. However, since there are two sockets with 8 cores, that means it supports up to 32 (2 per core) threads. For this assignment, performance was measured with different configurations, 1 to 32 threads.

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?

   The work is distributed among the threads using a static work distribution method. The integers read from the file are distributed to the queues, one queue per worker thread. This means that the work is divided equally among the threads in advance.

   Static work distribution assigns tasks to threads before the threads start executing while dynamic work distribution assigns work to the available thread at runtime. This causes a bottleneck in the queue when multiple threads try to get access.

   In this case, the static work distribution works better because the work is divided evenly among the threads beforehand, and there is less overhead in managing the distribution during runtime. Also, it should be mentioned that this assumes that the processing time for each integer is relatively similar.

   Dynamic work distribution might perform better where the workload varies significantly for each work object, as it allows for better load balancing.

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

   - In the **pop** method of the **SafeQ** class, access to queue **q** must be protected with a lock. This method dequeues an item from the queue. **mtx** was locked to only allow one thread to modify the queue at a time.
   - In the **wait_and_pop** method of the **SafeQ** class, access to **q** must be protected with a lock. **mtx** was locked to only allow one thread to modify the queue at a time.
   - In the **worker** function, updating the shared variables must be protected with a lock. These variables are updated with local results from each thread. **mtx** was locked to only allow one thread to modify the queue at a time.
   - In the atomic version,
   - **pop, push,** and **wait_and_pop** access is protected with **atomic_flag. test_and_set** and **clear** were used for locking and releasing.
   - Accesses to shared variables were protected by making them atomic to stop data races.

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?

   The following variables were used as atomic:

   - **atomic<bool> producer_done**
   - **atomic<int> primes**
   - **atomic<double> sum**
   - **atomic<int> consumed_count**
   - **Vector<std::atomic<int>> number_counts**

   Using atomic for these variables ensures that multiple threads can update and access them without data races.

   However, for **queue<T> q** it is not suited to use atomic variables. Using atomic types for **size()**, **empty()**, **push()**, and **pop()** may lead to incorrect results or data races. Instead, an atomic flag **locked** was used to protect the queue and ensure safe access.

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?

   The **SafeQ** is implemented by a combination of a mutex and a condition variable. Here is a breakdown of how different methods handle thread safety:

   - **push:** In this method, a lock with mutex was used to ensure that the queue is accessed safely by only one thread at a time.
   - **pop:** a **unique_lock** was used with a mutex to protect the access to the shared queue. The lock ensures that only one thread can access the queue at a time. This prevents data races. **cv.wait(lock)** was used to make the thread wait if the queue is empty.
   - **wait_and_pop:** a **unique_lock** was used with a mutex to protect access to the shared queue. **cv.wait(…)** was used to make the thread wait until the queue is not empty. This stops the thread from accessing the queue when it’s empty.
   - For **size** and **empty**, lock with mutex was used to ensure that these methods return consistent results when multiple threads access the queue at the same time.

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

   - In **pop** and **wait_and_pop** methods from the **SafeQ** class, threads need to synchronize when accessing and modifying queue **q**.
   - In the **worker** function, threads need to synchronize when updating the shared variables.

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?

   The most overhead is caused by **Mutex locking and unlocking** and **Condition variable waiting and signaling.** These mechanisms cause overhead because they cause delays and also use context-switching costs. When multiple threads compete for the same mutex, they may experience some contention, which leads to delays. Also, waiting for a condition variable can be expensive because it suspends a thread until a condition is met. Signaling can also cause overhead. It may signal one or more waiting threads.

   In the atomic version, the most overhead is caused by **atomic operations, Spinlock,** and **producer-consumer synchronization.** Atomic operations are generally faster than locks, they still have a performance cost due to synchronization between multiple threads. Spinlock causes a significant overhead when there is contention. They wait for a thread to try to acquire the lock continuously. This can lead to high power consumption and processor usage. Also, the consumer has to check **producer_done** variable constantly to see if the producer is finished or not.

   The **Mutex** version works better with a larger number of threads because Atomic operations require hardware support for synchronization. When the number of threads becomes larger, the hardware struggles to maintain cache coherency (**Relation to memory and cache**). Mutexes, on the other hand, rely on the operating system to manage thread synchronization. Also, Mutexes can be more efficient when contention is high.

   The following implementations were used to overcome the performance issues:

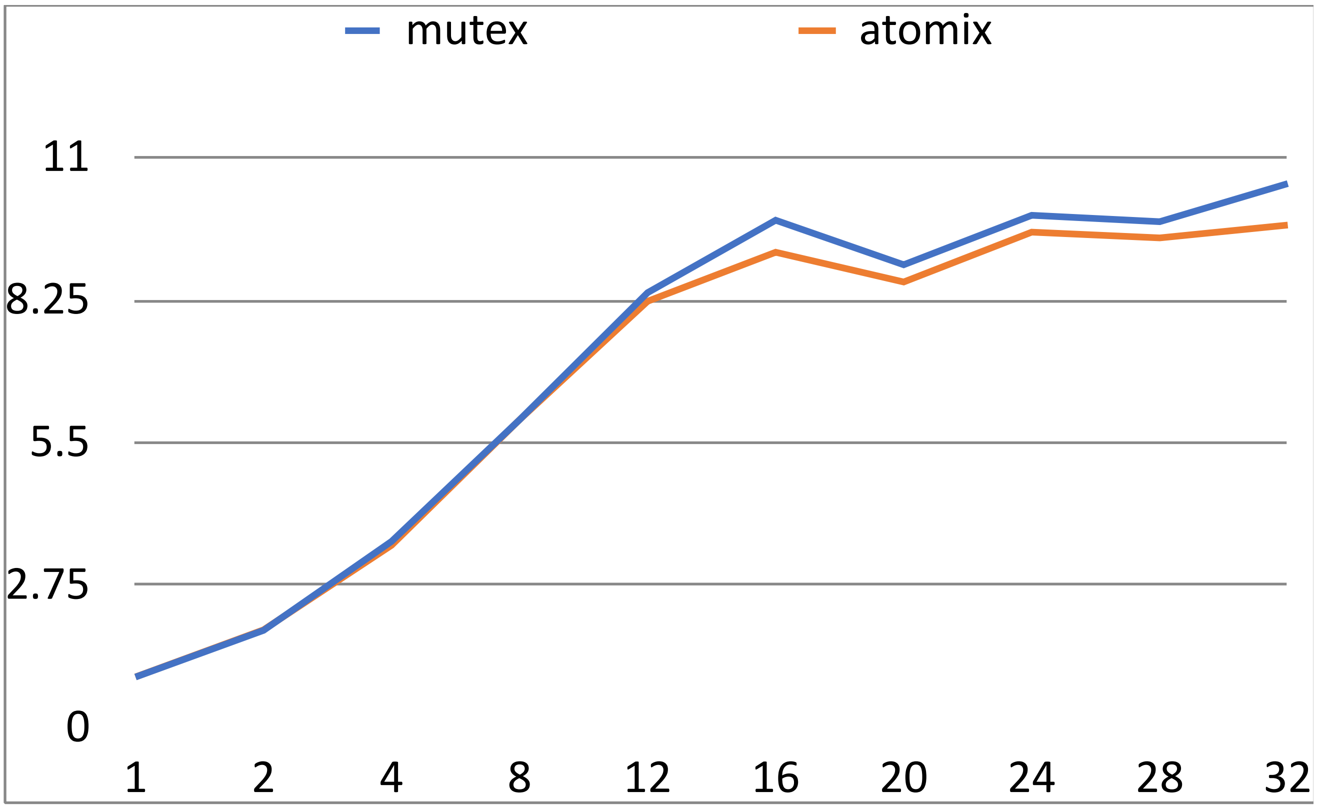
   - **Condition variable:** Used to synchronize access to the shared queue. It helps ensure that there is data in the queue before the worker thread tries to access it.
   - **Mutex: U**sed to ensure that only one thread can access shared resources at a time, which prevents data races.
   - **Fine-grained locking:** Used to minimize contention among threads.
   - **Local variables:** Minimizes the contention by reducing access to shared resources. (**Relation to memory and cache**)

   In the atomic version:

   - **memory_order_relaxed:** Helps to relax the memory ordering for atomic operations. It provides less strict ordering requirements. (**Relation to memory and cache**)
   - **Atomic operations:** For shared variables, atomic operations were used instead of locks. Atomic operations have a lower overhead compared to locks.
   - **Spinlock:** The **SafeQ** class uses a spin lock with **atomic_flag** to protect access to the queue. Spinlock can be more efficient than other locking mechanisms for smaller sections with low contention.

- **Separate workers:** Spawning separate worker threads reduces the need for synchronization.
- **ref:** Passing an object to a function or a thread by value, creates a copy of the object. This can cause some overhead. Using **ref** for shared objects ensures that they are working in the same shared state.
- **Future and async:** Used to let the **producer** work asynchronously with the worker threads. This makes the system more parallel.
- **push_back:** Helps performance by dynamically resizing the workers' vector, enabling better memory allocation.
- **Static work distribution**: In this assignment, a static work distribution strategy is employed to distribute the work evenly among threads. This approach is suitable here, as each task has a comparable execution time, resulting in a balanced workload for each thread. The static distribution helps to minimize overhead in managing task distribution during runtime and reduces the idle time for threads, as they don't need to wait for each other to fetch tasks from a single shared queue. This leads to improved efficiency and better overall performance.

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!).



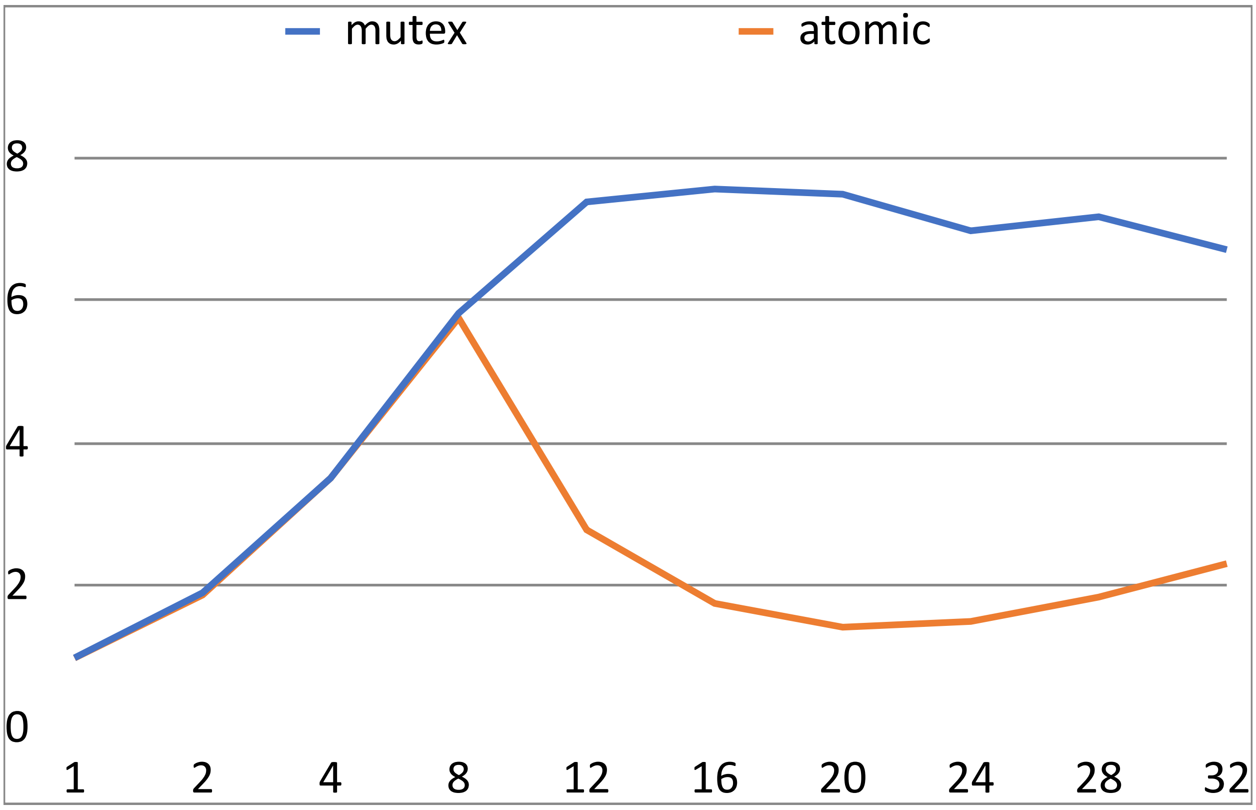
Sequential: 10.183

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.

10.183	Mutex	M-Speedup	Atomic	A-Speedup
1	10.37 s	0.982 x	10.34 s	0.985 x
2	5.41 s	1.882 x	5.38 s	1.893 x
4	2.83 s	3.598 x	2.89 s	3.524 x
8	1.713 s	5.945 x	1.713 s	5.945 x
12	1.213 s	8.395 x	1.238 s	8.229 x
16	1.04 s	9.791 x	1.11 s	9.174 x
20	1.14 s	8.933 x	1.184 s	8.601 x
24	1.03 s	9.886 x	1.065 s	9.562 x
28	1.043 s	9.763 x	1.078 s	9.451 x
32	0.97 s	10.498 x	1.05 s	9.698 x

You write here any additional content and analysis that does not fit into the above categories.

When running the atomic-based program, it becomes apparent that synchronization issues arise with an increasing number of threads. As the number of threads grows, the results display greater inconsistency. This inconsistency in the results can be traced back to insufficient synchronization, which becomes more noticeable as more threads are involved. The disparity between the atomic and mutex versions is more evident in the context of dynamic work distribution. The following plot and speed-up information are derived from a dynamic work distribution implementation of the assignment. It is clear that the atomic version faces challenges in maintaining accuracy as the number of threads escalates.



10.183	Mutex	M-Speedup	Atomic	A-Speedup
1	10.296 s	0.989 x	10.33 s	0.986 x
2	5.356 s	1.901 x	5.447 s	1.87 x
4	2.903 s	3.508 x	2.903 s	3.508 x
8	1.75 s	5.819 x	1.77 s	5.753 x
12	1.38 s	7.379 x	2.783 s	3.659 x
16	1.347 s	7.56 x	5.813 s	1.752 x
20	1.36 s	7.488 x	7.178 s	1.419 x
24	1.46 s	6.975 x	6.793 s	1.499 x
28	1.42 s	7.171 x	5.533 s	1.841 x
32	1.517 s	6.713 x	4.41 s	2.309 x