Code Report

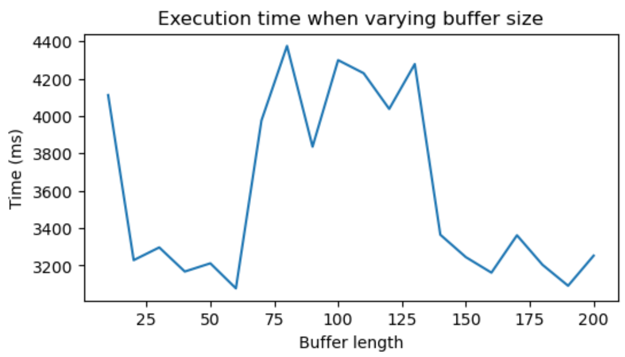
Design

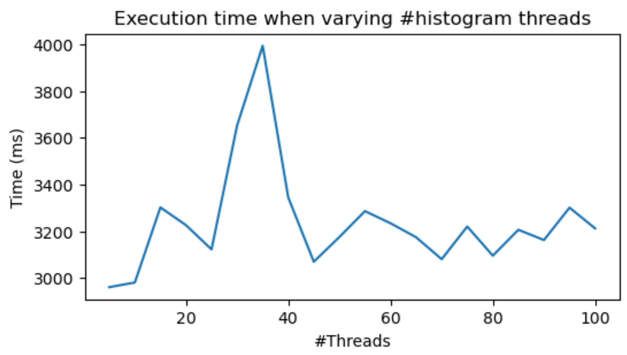
The task consisted of implementing a thread-safe BoundedBuffer that allows for effective communication between the threads. To do this, two functions, push and pop, were implemented, that make use of two condition variables and a mutex for synchronising both the producers and the consumers.
To lock the mutex, the code makes use of the unique_lock feature, which allows it to be unlocked when required. This lock is then unlocked within a call to the conditional variable that contains a lambda function highlighting when it is safe to proceed. In pop, it proceeds when the size is larger than zero, while in push is when the size is less than the capacity of the buffer.
Finally, after a thread has finished performing its pop/push function, it releases the lock and notifies a single thread to wake up. To acquire and release the lock, the code makes use of C++'s object scope, and so it's wrapped within a block, allowing for easy readability of the code.

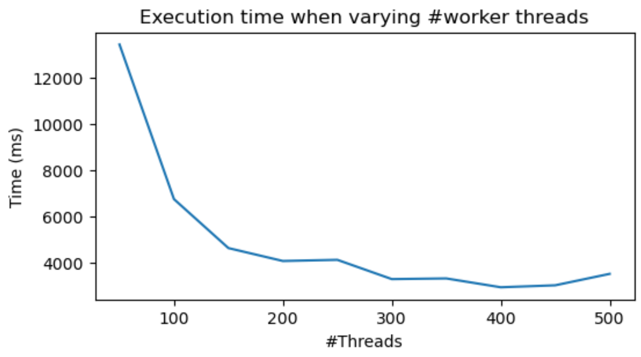
The communication between threads is done exclusively via the BoundedBuffer class, which is thread-safe and thus there are no race conditions. However, some communication must be done between the threads to notify that the producer is complete. The main thread performs this after detecting the termination of all producer threads (via thread.join()) and then places #Consumer Threads quit messages in that buffer so that all following threads will quit correctly. This behaviour maintains the temporal dependency requirements of the requests and removes any data races thanks to the correct implementation of the BoundedBuffer class.

For testing, an M1 Macbook Air was used, which contains an ARM architecture with 4 high-performance cores and 4 high-efficiency cores.

Data Requests







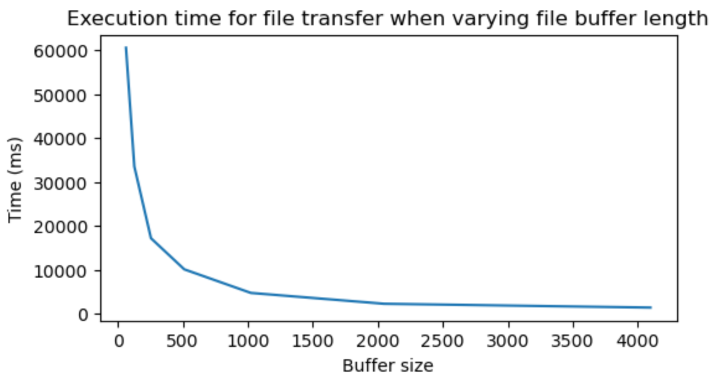
To generate these graphics, the following default parameters were used: 250 worker threads, BoundedBuffer capacity of 100, 50 histogram threads. For each one, one of these numbers would be variable, thus giving the behaviour for that dimension.
As visible in the graphics above, only changing the number of worker threads changed the performance considerably. Changing the capacity of the BoundedBuffer class showed strange behaviour but all within a 1s interval. Changing the number of histogram threads didn't improve performance at all, and it is visible that having fewer histogram threads provided the best performance of all of them.

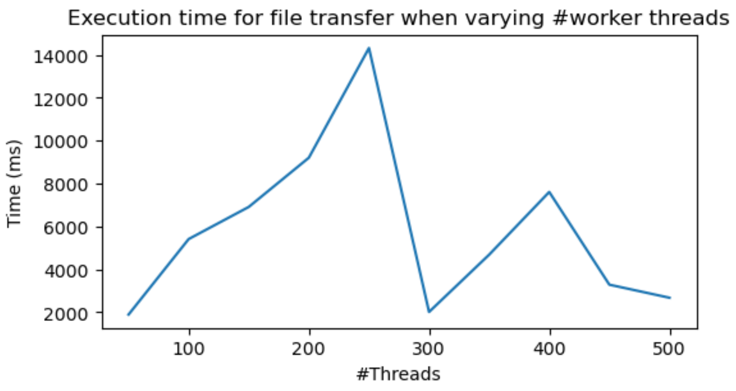
After considering the design of the system, there are two points to highlight:

- The system is being oversubscribed, and thus there isn't a linear increase in the performance after using 8 threads. In reality, the more threads we use, the more context switching will occur between the threads, and it might actually impact performance negatively;
- Changing the # worker threads provided an improvement thanks to the sleep call in the server that slows down the throughput. By increasing the number of threads, some of the cores that would be waiting for data to arrive can perform other work (pushing/popping from the buffers) that can accelerate the process.

Thus the performance is as expected.

File Request





Similarly to the data request situation, default parameters were used. Here, the buffer length was 2048, and the number of worker threads was 250.
For the file requests, we can see a curious behaviour, where changing the number of worker threads doesn't improve the performance and can, in fact, worsen it. However, changing the length of the messages being sent increased the performance substantially.

After further analysis, contrary to the data request code, there is no sleep occurring in the file transfer, and it is very unlikely that the threads won't be doing useful work. Instead, by oversubscribing the system, we can actually worsen performance and it can take up to 14s sometimes.
On the other hand, increasing the length of the buffer is better, and provides much higher performance. This is due to fewer messages being sent on each pipe, which require fewer system calls (which are slower) as well as fewer interactions with the bounded buffers, thus reducing the synchronisation requirements.