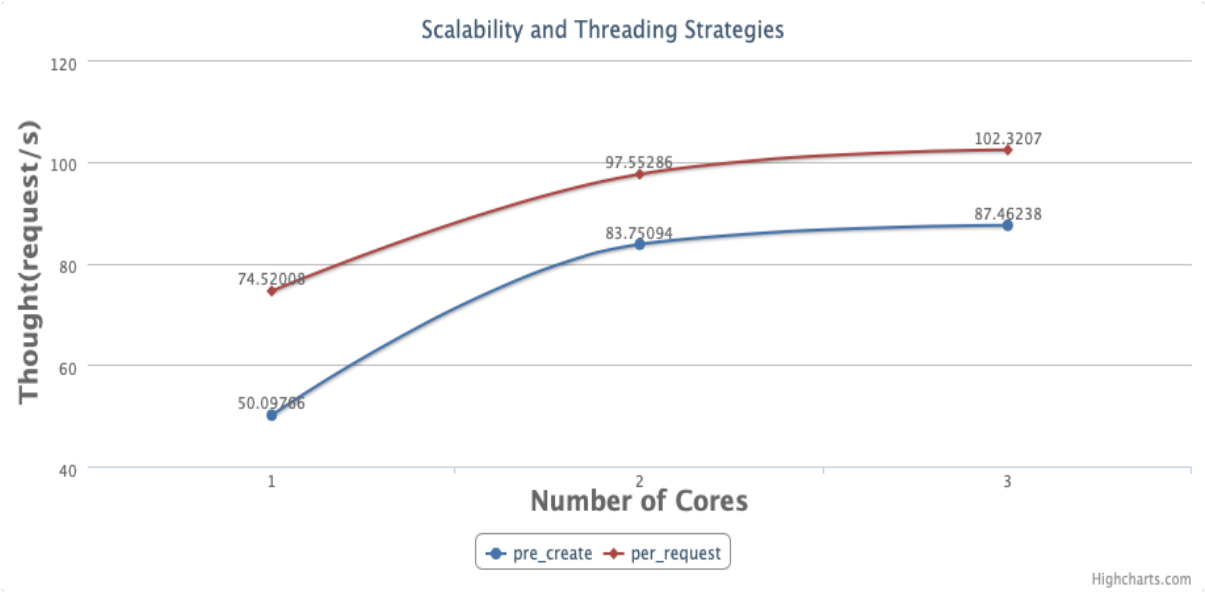
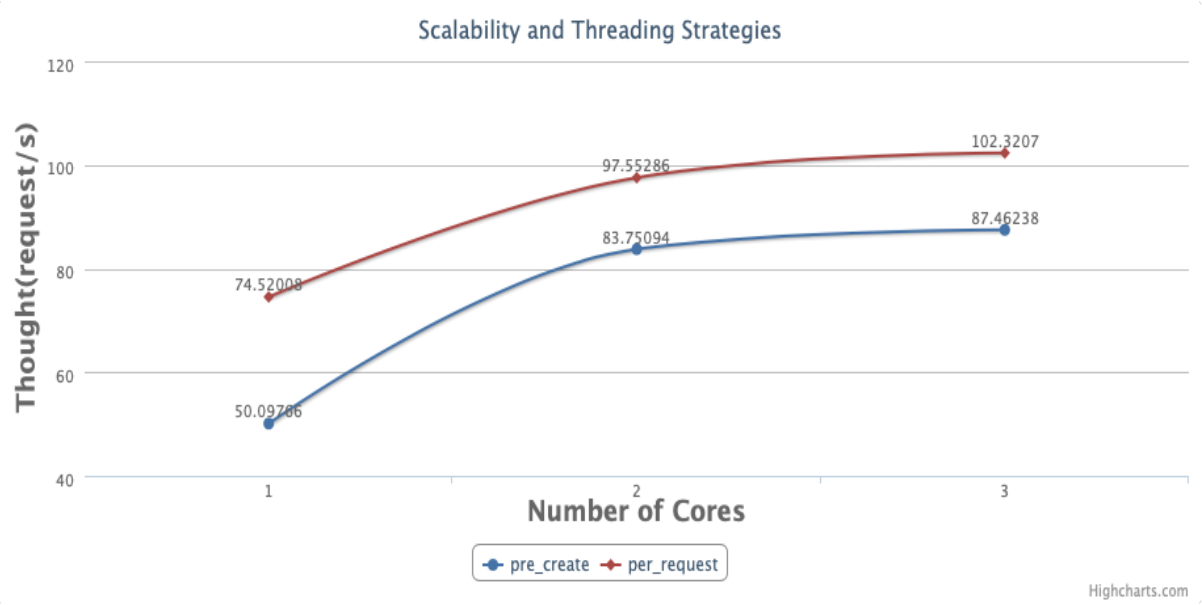
1. The throughput of your server code when running with 1, 2, and 4 cores available (this is the performance scalability of the code).



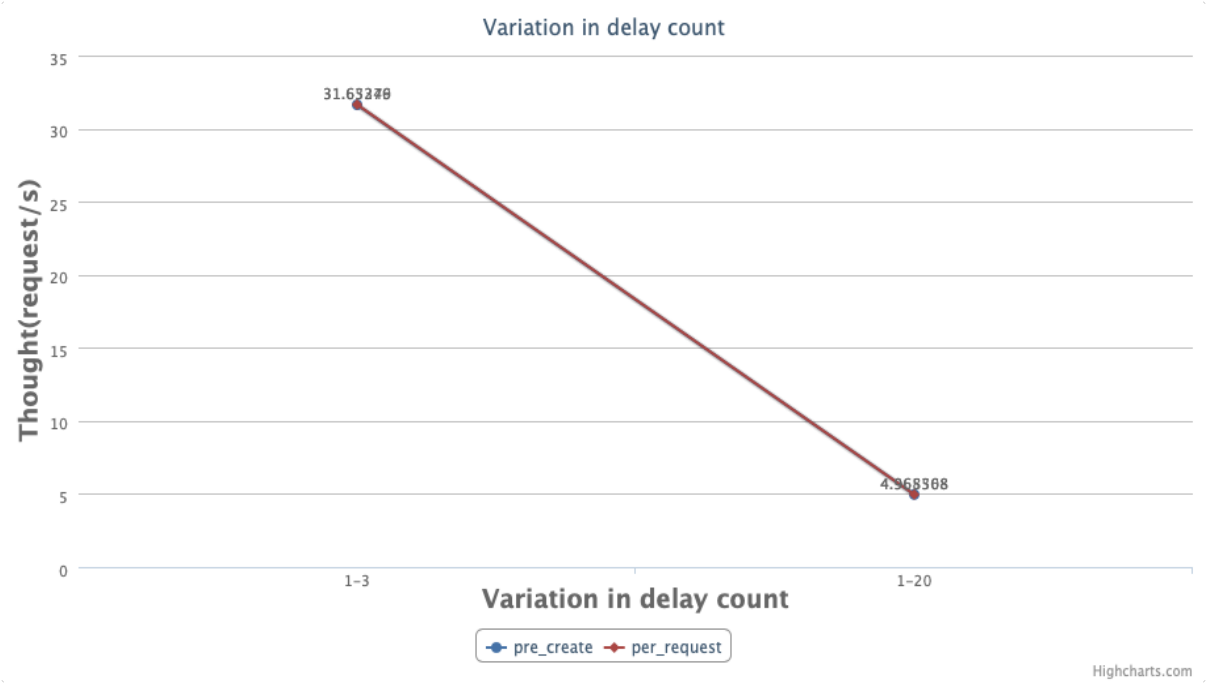
As it's shown in the above picture, throughput is increasing with using more cores. It is true that with more cores, the processor can handle multiple threads. In this way, it reduces the overhead of context switching among threads. We used taskset the "cpuset" attribute of the docker compose file to restrict the usage of cores. After multiple tests, we got the lower bound of threads to saturate the 4-core server throughput was about 600 hundred requests. The 600 requests should be sent out in 2 seconds. In other words, to saturate the 4-cores server for threadpool, it needs 300 requests per second from the client.

2. Two threading strategies: (1) create per request and (2) pre-create.



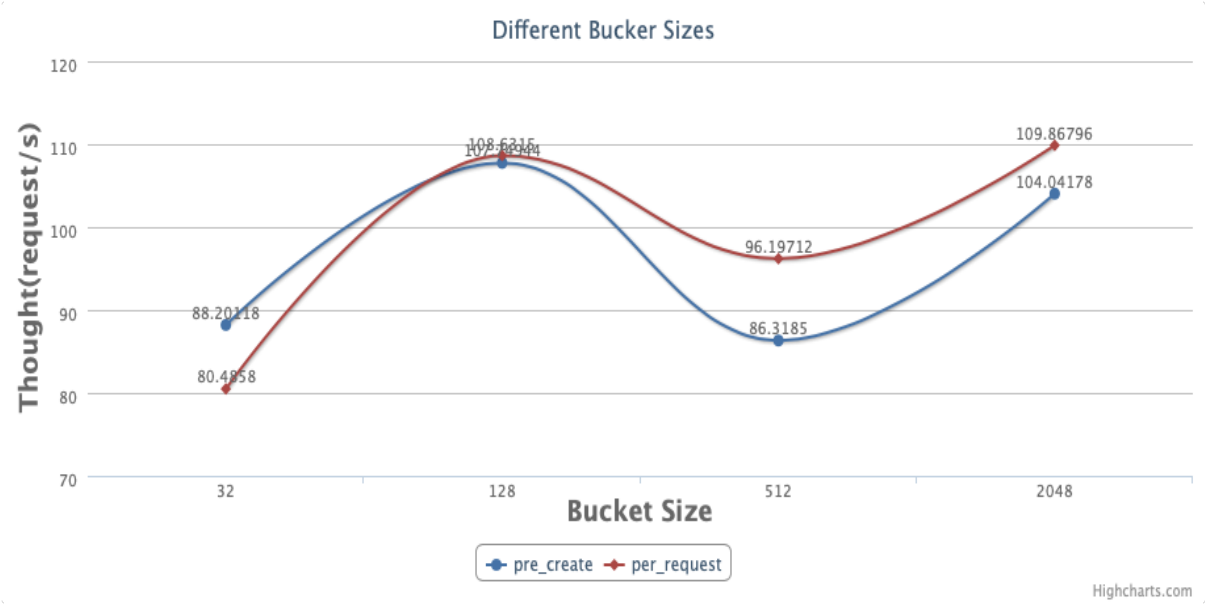
Ideally, the pre-create method is more efficient than creating a thread when accepting a request. This is because the pre-create method reduces the overhead of creating thread. However, the result can be different. From the picture above we conclude that the pre-create method is less efficient than the other way. It is because of the structure of the threadpool. In our program, we used Boost.asio library to create the threadpool. The Boost::asio::io_service should have its own structure of communication and manipulation of threads. To make sure the threads are isolated with each other, it might have additional overhead from locking the data. In this way, the efficiency of threadpool in our program is worse than creating one thread for incoming every request.

3. Small vs. large variations in delay count (with 4 cores active). For small delay count variability, you might try delays of 1-3 seconds. For large delay count variability, you might try delays of 1-20 seconds.



With small variation in delay count, the process can handle more threads in a certain time. It is because the time that one thread occupied the core is relatively less than in larger variations. And as it's show in the above picture for the two threading strategies, with all factors being equal, their performances are almost identical to each other.

4. Different bucket sizes (32, 128, 512, 2048) buckets (with 4 cores active).



In theory, when bucket size gets larger, it is less likely for 2 locks trying to lock the same bucket at the same time (i.e. less contended). As a result a lot of locking overhead could be saved, so the performance should be better. But that is not what we observed, we think the reason for that is: the delay caused by locking overhead is overwhelmed by the delay we set. To reflect the difference caused by different locking overhead, we need to set the delay count comparable to the locking overhead (which is very small). However, the requirement states that the delay count needs to be integer (i.e. at least 1), so there is no way to reflect the influence of different locking overhead.