ECE 568 Homework 4: Scalability

Authors: Huiling Yan and Junqi Sun

1. Code design

Our multi-threads version of Scalability project is made up of several part:
1) Client.cpp: this part is consist of thread pool creating, generation of random number for sending (delay, bucket_to_add) and use each thread to perform as client. The while loop guarantee that all idle thread will be used to perform new client task.
2) Server.cpp: this part is consist of thread pool creating, thread per request function, thread pre-create function. For thread per request functions: we put the entire function within a while(true) loop to run, then call accept function to accept messages from client then we use std::thread to generate a thread to handle this. For pre-create function, we create thread_pool before we enter the while loop, then accept and handle each request by threads stored in thread pool.
3) Makefile: creating exe file.
4) ctpl_stl.h: This is a external library, we quote this part of code from https://github.com/vit-vit/CTPL/blob/master/ctpl_stl.h(Copyright (C) 2014 by Vitaliy Vitsentiy). It is used to generate thread pool.

2. Test items:

(a) Two threading strategies: (1) create per request and (2) pre-create.
(b) The throughput of server code when running with 1, 2, and 4 cores available (this is the performance scalability of the code).
(c) Small vs. large variations in delay count (with 4 cores active). The variations here means delay counts.
(d) Different bucket sizes (32, 128, 512, 2048) buckets (with 4 cores active).

In this project, we combine (a), (b) which would be 6 data points, we also set delay range and bucket size to test. We also test (a), (c) and (a), (d) combination with 4 cores.

3. Test data

All data values are collected that tests run for more than 3 minutes when outputs become stable.

Test 1: We compare throughput per 10 seconds between per request method and pre create method with different number of cores running. The output graph with error bars shows as figure1.

	Throughputs: num of requests/ 10 sec	
Number of core	Per-request thread	Pre-create thread
1	216,184,188	272,226,229
2	556,551,667	1208,1177,987
4	1122,1249,1170	1454,1434,1428

Form 1: data for test 1

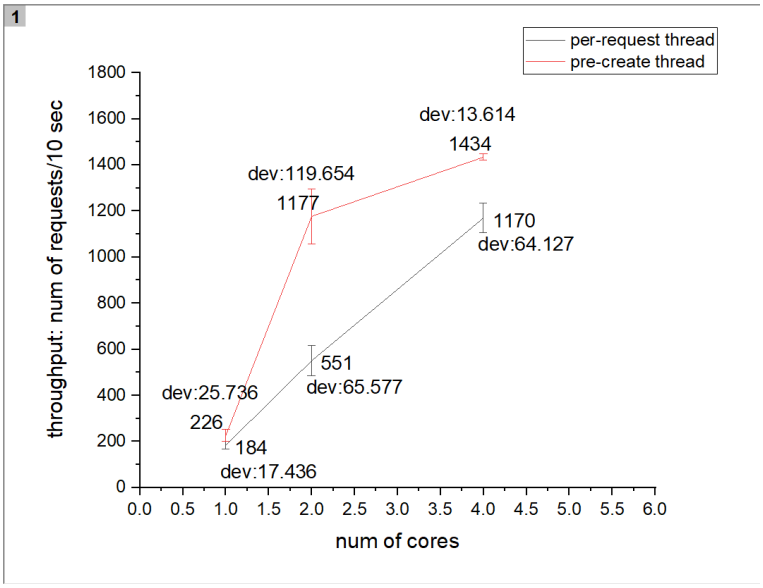

Figure1: two methods' throughput with different cores number comparison

Test 2: We compare throughput per 10 seconds between per request method and pre create method with different size of bucket. The output graph with error bars shows as figure2.

	Throughputs: num of requests/ 10 sec	
Size of bucket	Per-request thread	Pre-create thread
32	1122,1249,1170	1454,1434,1428
128	1863,2078,1966	1715,1115,1265
512	2285,2322,2376	1396,1029,1191
2048	2808,2879,2812	1081,1170,1138

Form 2: data for test 2

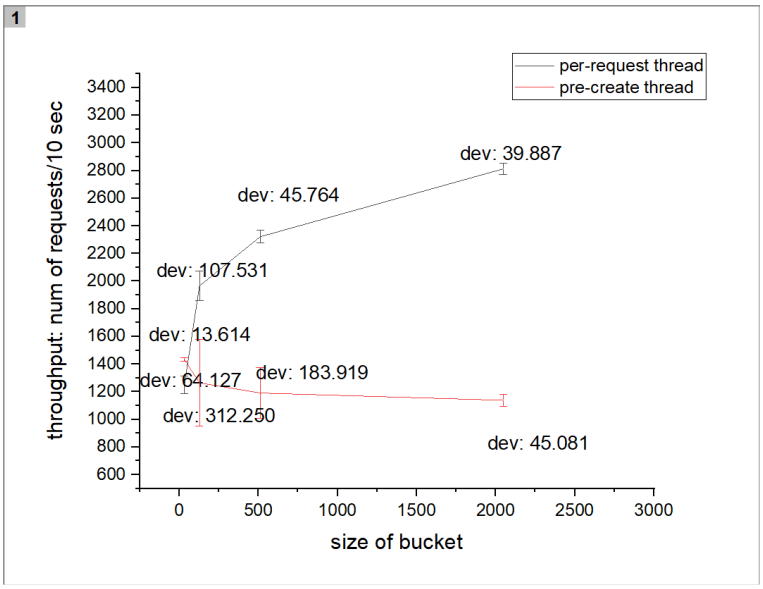

Figure2: two methods' throughput with different bucket size comparison

Test 3: We compare throughput per 10 seconds between per request method and pre create method with different delay time. The output graph with error bars shows as figure3.

	Throughputs: num of requests/ 10 sec	
Delay time(s)	Per-request thread	Pre-create thread
1~3	1863,2078,1966	1112,1115,1265
1~20	865,883,851	255,252,257

Form 3: data for test 3

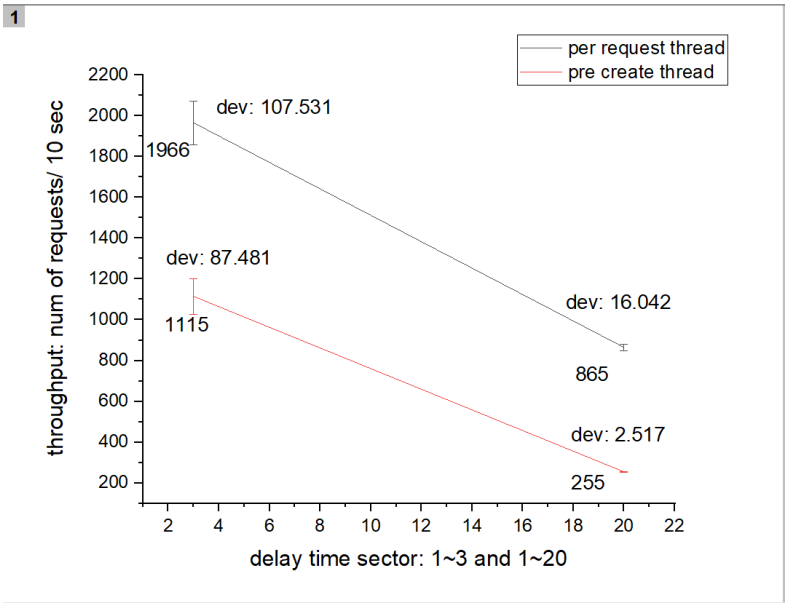

Figure3: two methods' throughput with different delay time comparison

4. Result analysis

For test 1 (a)+(b): In this test, there is a pattern that with the increasing number of cores involve, the throughput of each method grows larger. When comparing two methods' throughput with the same cores function, pre-create method outcome per-request method in all cases. Since when cores number change from 1 to 4, more resources will be used for creating threads, perform function, so the throughput will increase. However, since pre-create methods only need to deal with latency of creating thread once and the other method need to deal with that whenever new request come, so its performance outcome per-request method.

For test 2 (a)+(d): In this test, there are different patterns for both methods: for per-request method, larger the size of bucket, larger the throughput it has. For pre-create method however, the pattern just go opposite. Since the per-request method accept the connection, then create a thread for this connection, and finish it right away, when bucket size grows, the creating, destructing latency portion will be less for this method. In that case, the per-request method will run faster with bigger bucket. For pre-create method, however, it need to create all the threads at the same time and handle the task (we saturate the thread pool when running tasks), so it will run slower if all threads need to handle bigger bucket at almost the same time.

For test 3 (a)+(c): In this test, there is a same pattern for both methods: the longer delay we set, less throughput they have. It can be easily understood that when delay time increase, the total time for task handling will increase too, so throughput will decrease no matter what method we choose. Comparing two methods under the same condition, however, per-request method run faster than pre-create function. Also, pre-create function seems to suffer more influence than per-request method, I think the reason is handling all threads' task together will amplify the latency factors.