ECE568 HW4 REPORT

Yisong Zou(yz558)

Yuting Zhang(yz579)

Introduction: Two threading strategies comparation

This report contains four parts, all this experiments are aimed to compare two threading strategies under different restrictions. In the first part we test the throughput of different cores with two threading strategies separately, and give the analysis of the results. The second part is to test the affect of different delays. The clients may conducted different delays, we simulate the behavior of clients and generated delay in two ranges: small delay which range from 1 second to 3 seconds, and large delay which range from 1 second to 20 seconds. The behavior of server with different threading strategies also show different performance. Part 2 gives comparation of the results. Part 3 test affect of different buckets sizes, we test the data under 4 categories: 32 buckets, 128 buckets, 512 buckets, and 2048 buckets. All part 2 and part 3 are base on 4 core environment. Besides, part 3 is also based on small delay as background.

The final part is about the scalability of our project, the performance change with the load and resources. As an example, we describe the process of how the performance change with the load change and finally find the boundary load which as higher performance than others. In this part we described how we enhance the performance and introduce our strategy about how to make multiple clients stress the server enough to give the saturate results.

Principles and our approach:

We choose to use C++ to implement this project.

Pre-create a set of thread：

As for the detailed implementation, we have two design for pre create threads at the first time. The first design is to put all threads in the thread pool and take the available threads as needed. Each thread inside the client side run in a while loop continuing to connect the server at the same time. And the second design is to make all pre created threads in server each run in a while loop, and accept the request from the threads inside the client side. Each thread inside the client side run in a while loop continuing to connect the server at the same time. We finally choose the first deign to reduce avoidable resource waste.
For the thread pool, we use a library CTPL:https://github.com/vit-vit/CTPL. More specifically, there are some threads dedicated to the pool and a container of jobs. The jobs come to the pool dynamically. A job is fetched and deleted from the container when there is an idle thread. The job is then run on that thread. A thread pool is helpful when we want to minimize time of loading

and destroying threads and when we want to limit the number of parallel jobs that run simultaneously.

Creat a thread for each request：

As for detailed implementation, we use the same client part as pre-create a set of thread. However, for the server part, we will create a thread and detach it whenever we accept a request from the client. And the thread will deal with a single request and it will finish its job.

Test Announcement:

- Each individal test are 5 minites long, and we calculate the throughput by following formular:

`total request the server handle/total seconds  = request/second(throughput)`

Here the total seconds are 300 s (5 minites)

- Each test data are the average result calculated from three tests.

Part1: Throughput of 1, 2, 4 cores comparation

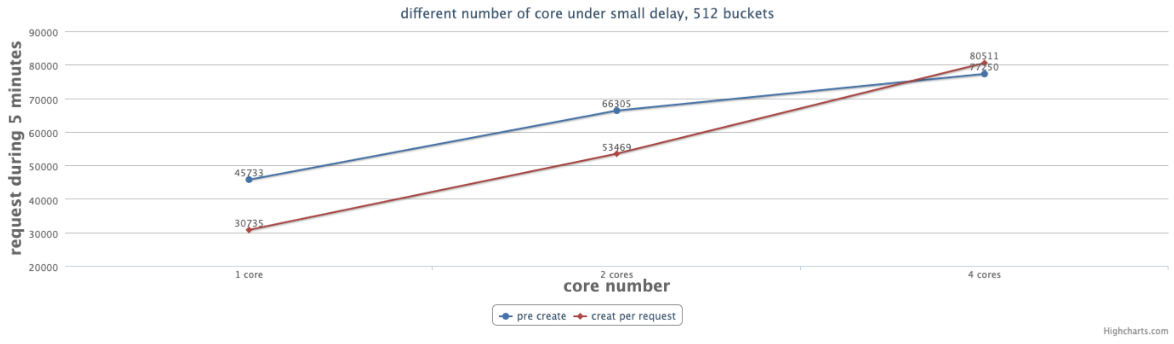
This part is based on different core environments. We test it on 1 core, 2 cores, and 4 cores separately. To control other variables, we test this part based on small delay(1-3 seconds) and 512 cores. This part reveals the relationship between hardware and scalability, we have the following results, We test each one 5 munities long and calculate the throughput.

The content of the table has the following format:

`total request the server handle/total seconds  = request/second(throughput)`

core number	pre create thread	thread/request
1	45733/300 = 152.44 requests/second	30735/300 = 102.45 requests/second
2	66305/300 = 221.02 requests/second	53469/300 = 178.23 requests/second
4	77250/300 = 257.5 requests/second	80511/300 = 268.37 requests/second

To virtualize it, we draw a graph base on the data above, the x axis is the core number and the y axis is the total request the server received during 5 minutes.



Analysis:

We find out that the throughput of server isn’t enhanced four times when we change from 1 core to 4 cores. We think there are several reasons: Firstly, according to Amdahl's Law, there might be some part of the code that can not be paralleled. Secondly, different cores may have shared hardware , data Movement and our lock also make the code not embarrassingly parallel.

Pre-create threads create threads and collect them in a thread pool, that will cost some overhead to set up the thread pool at the first time. However, once the thread pool has been set up, we can use the threads from thread pool immediately, and that only costs very little time. Besides, the thread poll overhead is controllable since we can control the total number of threads.

On the other hand, creating thread per request strategy has no overhead at beginning. Since we don't control the total thread number, how many threads will be created totally depend on how many request the server received. Each time the server get a request, it costs extra time than pre-create strategy to create thread and communicate with client using the new thread.

When we run the program on 1 core and 2 cores, the calculation time is slow and it cost considerable time to create a thread per request. But the pre-create can take thread immediately and don't need to cost time creating new threads. The data we collected has the results according to our analysis, pre create thread is faster than create thread per request when the core number is 1 and 2. Thus, the pre-create thread has higher throughput than create thread per request. However, when the core number is 4, the operation performance is enhanced and the calculation ability is promoted. The overhead of creating thread per time is reduced and the time is considerably reduced. At this time, the thread pool overhead is obvious and it takes amount time to set up the thread pool, the pre-create thread has lower throughput than create thread per request.

Part2: small vs. large delay with two threading strategies

This part is based on 4 core environment. To give the right result and give a higher view of the project, we test the small and large delay under different bucket size. The following are the results. We test each one 5 munities long and calculate the throughput .

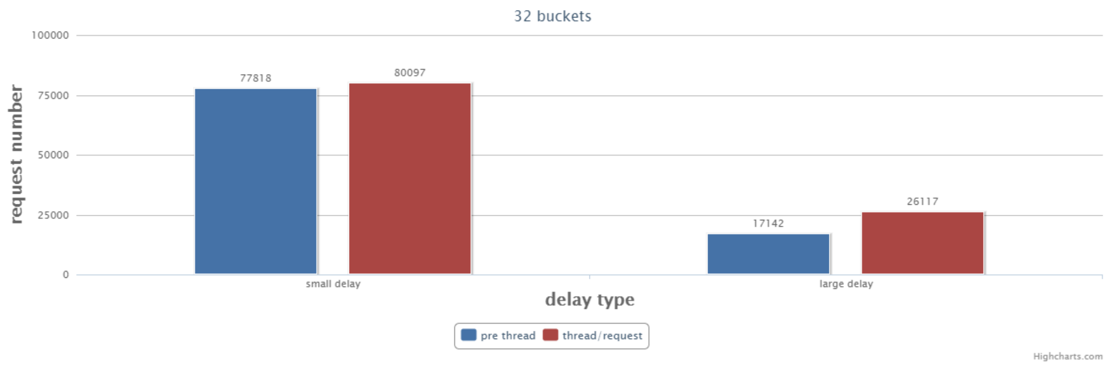
The content of the table has the following format:

`total request the server handle/total seconds = request/second(throughput)`

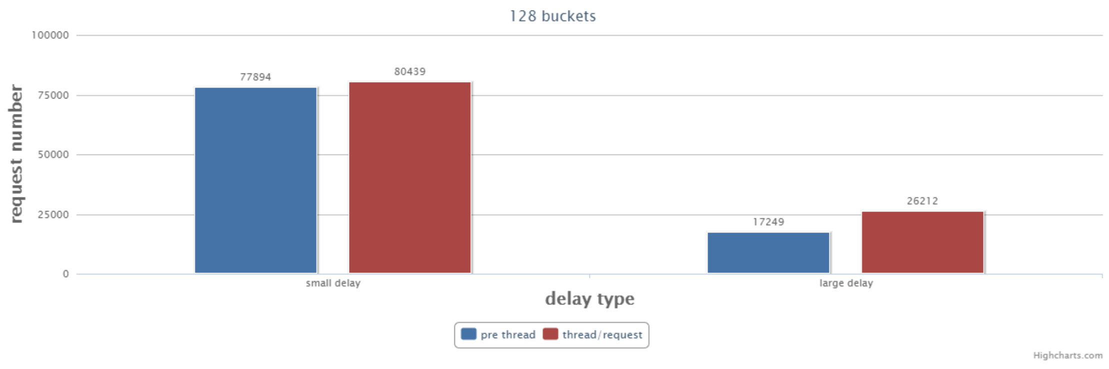
bucket number	pre create thread with small delay	thread/request with small delay	pre create thread with large delay	thread/request with large delay
32	77818/300s = 259.39	80097/300s=266.99	17142/300s = 57.14	26117/300s = 87.06
128	77894/300s = 259.64	80439/300s=268.13	17249/300s = 57.49	26212/300s = 87.37
512	77250/300s = 257.5	80511/300s=268.37	17407/300s = 58.02	26115/300s = 87.05
2048	77433/300s = 258.11	79814/300s=266.04	17313/300s = 57.71	26068/300s = 86.89

To virtualize it, we give 4 graphs base on the data above, the number in graph is the total request number server received during 5 munities.

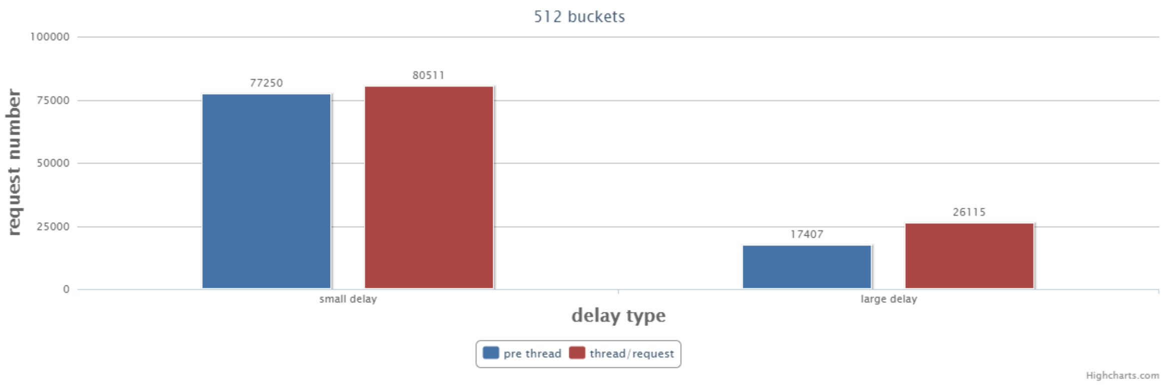
- 32 buckets



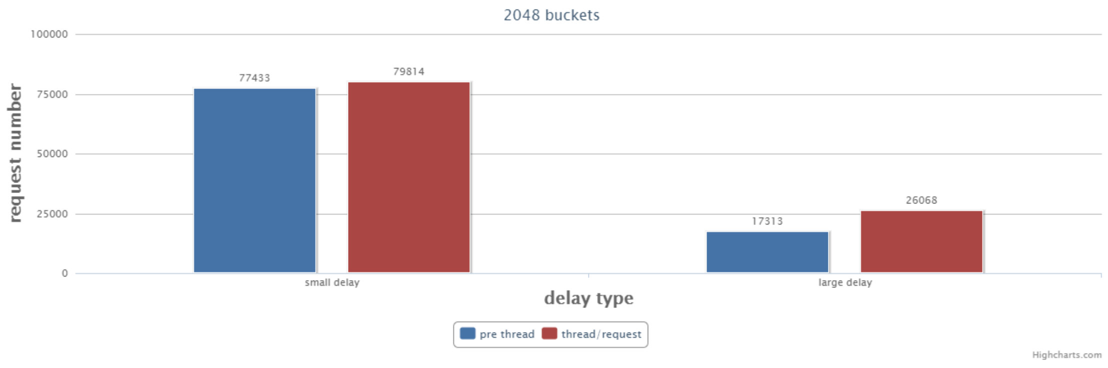
- 128 buckets



- 512 buckets



- 2048 buckets



Analysis:

When the client has small delays(1-3 seconds), the throughout is average three times of large delay when we take thread per second strategy, and the throughput is average four times of large delay when we take pre create thread strategy. That means the increment of delay from client will reduce the performance of the server.

When the delay is small, the server can deal with more threads. When the delay is large, the requests the server can deal with will decrease. In the pre create thread version, the avaible threads in thread pool will significantly decreased since most threads are occupied by previous requests. In the thread per request version, the threads will lock the buckets when the server operate on buckets, that will block other threads and let multiple threads waiting for that. Above all, the scability will be affected and the server can deal with less load due to larger delay.

Part3: different bucket size comparation

This part is based on 4 core. To gain a better observation, we still test it fully on both small delay and large delay to see if the trend is the same.

The following are the results. We test each one 5 munities long and calculate the throughput .

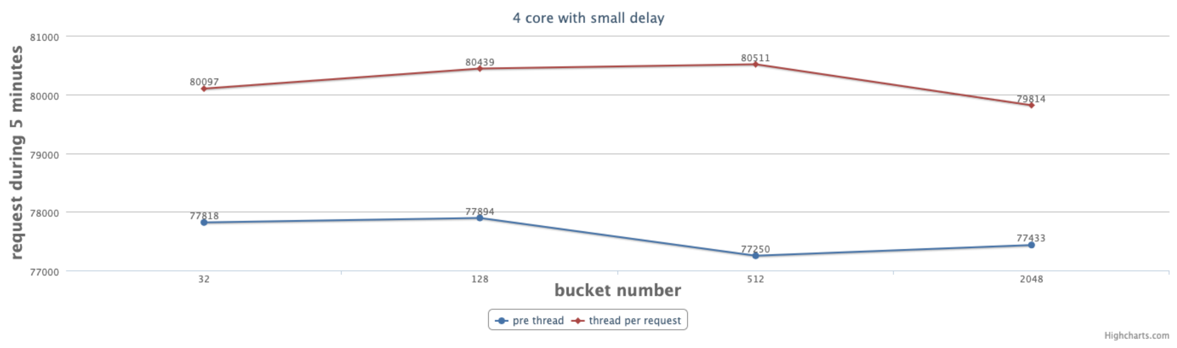
The content of the table has the following format:

`total request the server handle/total seconds = request/second(throughput)`

different bucket size comparation under small delay

bucket number	pre create thread	thread/request
32	77818/300s = 259.39	80097/300s=266.99
128	77894/300s = 259.64	80439/300s=268.13
512	77250/300s = 257.5	80511/300s=268.37
2048	77433/300s = 258.11	79814/300s=266.04

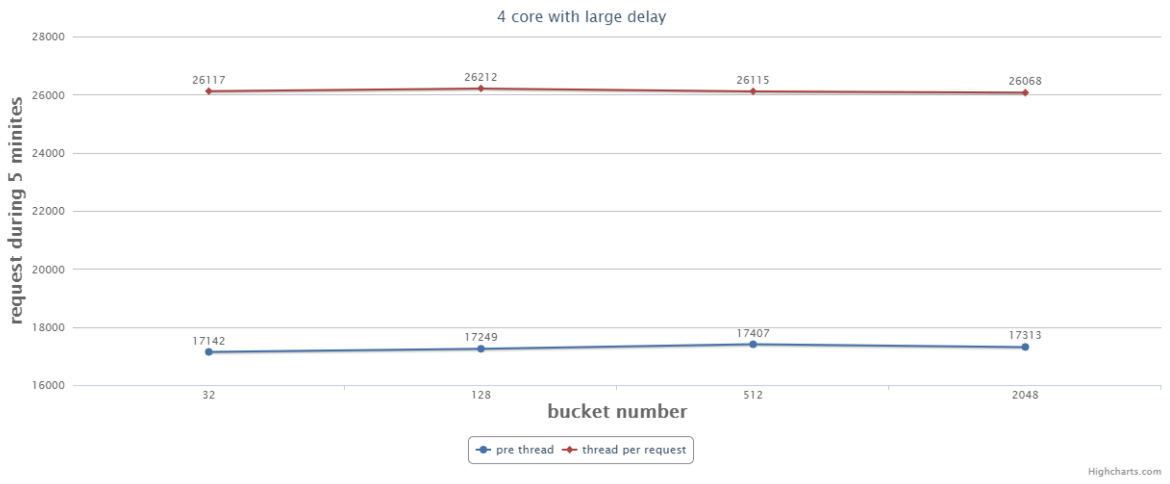
To virtualize it, we draw a graph base on the data above, the number in graph is the total request number server received during 5 munities.



different bucket size comparation under large delay

bucket number	pre create thread	thread/request
32	17142/300s = 57.14	26117/300s = 87.06
128	17249/300s = 57.49	26212/300s = 87.37
512	17407/300s = 58.02	26115/300s = 87.05
2048	17313/300s = 57.71	26068/300s = 86.89

To virtualize it, we draw a graph base on the data above, the number in graph is the total request number server received during 5 munities.



Analysis:

The results from small and large delay has the same trend when bucket size changes. In a word, the bucket size don't have much affect on the throughput, the throughput may decrease a little bit. In our approach, each time the server want to edit the bucket, the server will lock the vector of all buckets, and edit the specific bucket. Thus, the bucket size will not have much affect on the result since the operation is the same. There is another approach which lock the specific bucket when doing operation, that approach will have higher throughput when bucket size increase. But in this project, we choose the first approach to reduce avoidable chaos.

Part4 Conclusion: scalability analysis

The scalability is based on hardware and load. With the increment of more cores, the scability will be enhanced. Since the calculation ability increase along with more cores, more request the server can handle and higher throughput can be gained. Also we noticecd that when the throughput of 4 cores are not exactly 4 times of 1 core throughput. Since hardware, locks and blocking IO all have affect on scallablity, the performance may be affected by lockings on buckets and shared hardwares.

Besides, the load also have affect on scalability. We test different load to find the load which can lead to higher throughput. To stress the server, the client needs to send multiple requests rapidly enough to to saturate the server throughput. To do this, we created multipel threads on client side and let the client request constantly. That simulate the behavior of multiple clients request constantly at the same time. We test the boundary to find the bigest load the server can suffer and then test the load which can lead to higher throughput.

The following is the process of find the load leading to higher throughout.

- for pre create thread, find the peak throuput when server change the amount of thread in thread pool.

thread number in thread pool	request number in 20 seconds
3000	2761
2000	3547
700	5400
650	5606
600	5819
550	4841

- for pre create thread, find the peak throuput when load changes(server has fixed 600 thread)

client threads number	request number in 20 seconds
1110	4925
1000	5819
950	4870
900	4963

- for thread/request, find the peak throuput when load changes(server has fixed 600 thread)

client threads number	request number in 20 seconds
1110	4935
1000	5160
900	5008
800	2874

Thus, we take 600 server and 1000 client for pre create threads, and 1000 client for thread/request threads to gain the higher performace.