ECE 568 Hwk4 Report

Kewei Xia (kx30)
Weihan Zhang (wz125)

Important Note: to gain similar result you need to set up the server code on a 4-core VM from duke and set up the testing infrastructure code on **another duke VM** (no need to be 4-core but do need to be a duke VM). The reason is that the internet connection of your local computer is very likely to be unstable which may affect the testing result (we tried to use our laptop, and the throughput will degrade among 10 requests/s compare to VM).

• Session1: Test Throughput

We instrumented our server code to be able to collect throughput measurements by saving the data (the number of requests that the server responded to within every 15s, note that to avoid the overhead this logging bring to our code, we make a separate TimerThread, which responsible to log data) into the log file. To minimize the experiment error, we collected multiple measurements and calculate the average.

Then, we took the experiment by making multiple threads send requests on the client side in order to generate enough load. As we increased the thread numbers on client side, we could find that the number of requests that have been completed in given time period stopped increasing which means the server side is saturated.

We tried multiple numbers of threads on client side to get the exact throughput for different threading strategies. Also, we used 4 cores to run the server code.

Figure 1 demonstrates the actual data we collected. And by analyzing it, we found that our server would become saturated when we created roughly 120 threads to send requests on client side for the pre-create thread strategy and 80 threads for creating thread per request. Since server could reach its maximum throughput no matter what threading strategies when using 120 threads on client side, we decided to use 120 threads for further testing.

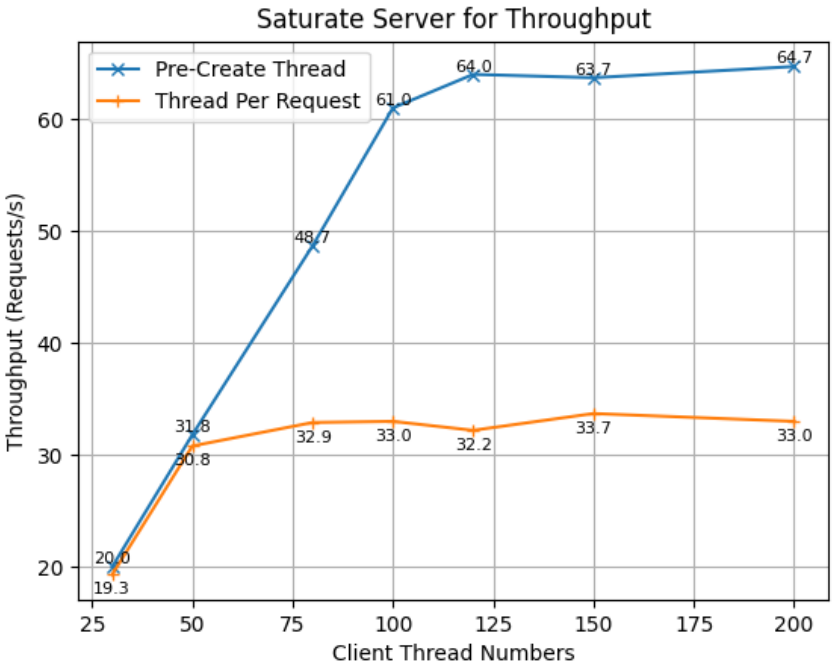

Figure 1. Client Thread Numbers vs Throughput

At the same time, I think this small experiment demonstrates the advantage of thread pool compare with thread per request very well. For the thread pool strategy, the overhead of creating a new thread will only happen at the beginning (which is great), but this overhead will happen in the critical path for the thread per request strategy, this will be one important reason why the throughput difference between these two strategies will be this large. Another reason is that for thread per request there are no limitation of the thread numbers, although thread is relatively a lightweight resource, but unreasonable large amount of threads (e.g. thousands of threads in one process) will still slow down the process.

• Session 2: Scalability Analysis

- 2.1 Threading Strategies & Core Numbers

In this part, we tried to run our code to compare the throughput of different core numbers with different threading strategies. And we fixed the bucket size to be 128 and used the small delay 1~3s for testing. The data collected is represented in the Appendix A and Figure 2 shows the tendency of the result clearly.

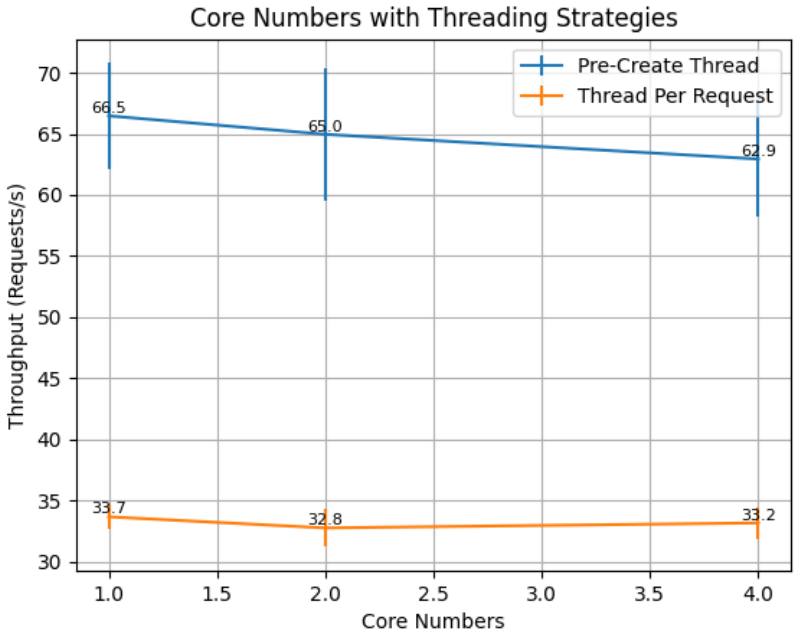

Figure 2. Core Numbers vs Throughput

We can see that pre-create threads strategy could get a higher throughput than create thread per request (the reason is stated in session 1).

For pre-create threading strategy, as the core number increases, the figure shows that the throughput would decrease slightly. We think the main reason of this is that the extra overhead caused by data movement (e.g. the bucket, the mutex), and we are aware that we use synchronized key word in java to protect the write operation to the bucket (which means that each time we write, we will lock the whole array), this may also be a reason that throughput slightly degrade.

For thread per request, the throughput is basically the same, this is because the overhead of creating new thread is still dominate here, data movement overhead is relatively small compare to this.

- 2.2 Bucket Size & Threading Strategies

We used the delay 1~3s and 4 cores to run the tests and analyze the effect on throughput of different bucket size and threading strategies. The data is represented in the Appendix B and Figure 3 shows average value of the results.

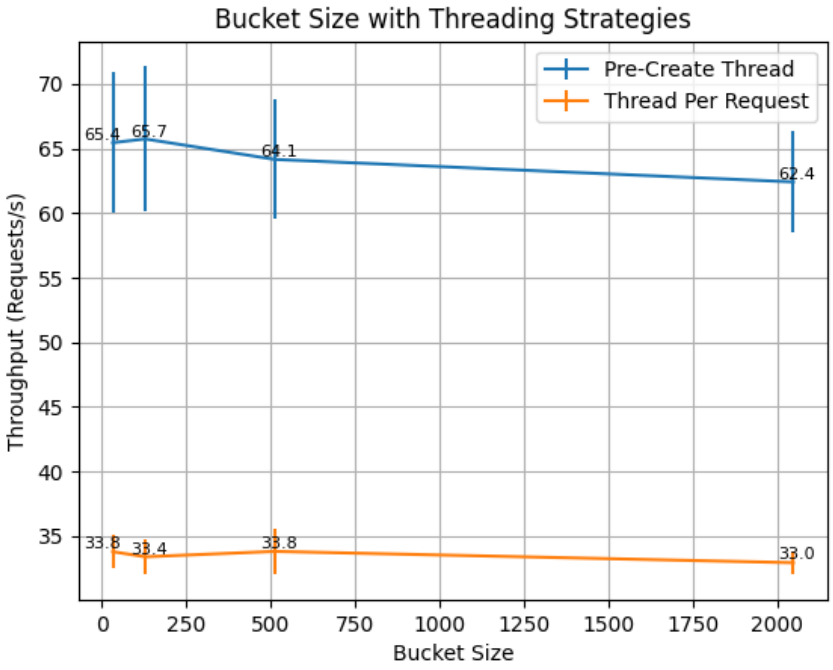

Figure 3. Bucket Size vs Throughput

From this image, we can see that with the bucker size increasing, the throughput with degrade slightly (especially for thread pool strategy). Based on our analysis, one main reason for this is that with the size of bucket grows, it may not be fitted in cache (e.g. size 32 can put in L1 cache, size 2048 may only put in L2 cache), so it will take more times to access the bucket.

For the locking granularity, we will lock the whole array(with *synchronized* keyword in java) each time we want to write (compare to have a separate lock for each entry), this will lead to almost the same lock contention even the size of bucket grows.

At the same time, from the curve of thread per request, we can still see that the overhead of creating new thread is still dominate here.

- 2.3 Delay & Threading Strategies

These tests are aimed to find out the effect of different threading strategies and different delays (small delay 1~3s & large delay 1~20s) regarding to the throughput. We configured to run with 4 cores and the bucket size 512. The data is represented in the Appendix C and Figure 4 shows average value of the results.

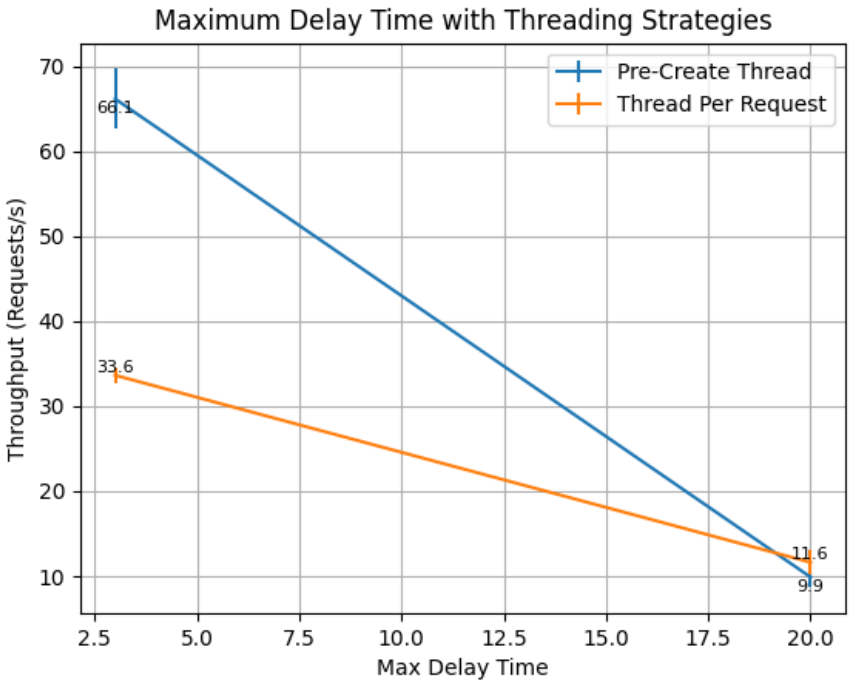

Figure 4. Delay Time vs Throughput

The delay simulates the time to finish each request. It is obviously that big range of delays would result of the smaller throughput, since every request will require more CPU cycles to complete.

There are two interesting point in this figure. 1) throughput of **both** threading strategies will decrease drastically for large delay range; 2) the throughput of **both** threading strategies will be very closed for large delay range.

1) The reason why throughput of thread per request will also degrade is the time to process request become dominate here (i.e. the overhead of creating a new thread become trivial compare to the longer process time).

2) As we analysis in session1, one main difference of two threading strategy is where does the overhead of creating new thread happen, so due to the same reason, with the process time increasingly dominant, the throughput of these two strategies become closed to each other.

Appendix A

We sample every 15s and run each testcase for 5 mins.

Core Numbers & Threading Strategies

		Pre-Create Thread			Thread per Request		
Core Numbers		**1 Core**	**2 Cores**	**4 Cores**	**1 Core**	**2 Cores**	**4 Cores**
Request Numbers / 15s	**1**	962	853	820	517	443	556
	2	935	1025	1015	535	488	479
	3	1059	967	890	534	479	514
	4	1025	1107	1061	505	498	512
	5	1069	969	889	498	471	479
	6	1034	1042	891	522	482	489
	7	1015	970	975	490	499	470
	8	872	1046	929	506	516	510
	9	954	966	1031	493	492	500
	10	934	890	1077	512	483	514
	11	924	883	965	517	505	494
	12	1034	913	1014	507	519	474
	13	939	876	995	477	456	483
	14	1026	912	923	484	545	487
	15	991	1003	920	501	510	498
	16	1043	1171	897	495	509	511
	17	959	901	836	506	480	495
	18	1007	1043	884	485	469	486
	19	1161	976	925	516	497	506
Throughput (Requests/s)		66.47	64.96	62.94	33.68	32.78	33.18

Appendix B

Bucket Size & Threading Strategies

		Pre-Create Thread				Thread per Request			
Bucket Size		**32**	**128**	**512**	**2048**	**32**	**128**	**512**	**2048**
Request Numbers / 15s	**1**	840	934	887	959	459	489	482	497
	2	912	1090	1069	922	507	488	586	512
	3	998	989	872	887	487	478	526	530
	4	887	906	996	949	482	485	503	496
	5	940	829	1104	1004	519	521	525	485
	6	1145	976	1008	915	514	527	498	494
	7	1083	1001	1033	844	509	520	476	499
	8	1017	995	984	930	495	484	493	502
	9	976	905	938	781	504	522	471	496
	10	921	963	965	970	531	506	487	500
	11	976	964	937	997	492	495	493	494
	12	926	1008	936	1018	540	468	504	471
	13	1006	969	903	869	531	551	531	480
	14	1080	1081	961	903	522	493	539	495
	15	1134	1063	975	1002	522	478	522	478
	16	996	921	819	933	499	525	514	480
	17	994	937	904	956	526	506	511	507
	18	890	1229	1049	959	502	494	473	484
	19	930	973	942	988	495	493	508	493
Throughput (Requests/s)		65.97	65.92	64.43	62.32	33.99	33.46	33.93	32.95

Appendix C

Delay Time & Threading Strategies

		Delay times 1~3s		Delay times 1~20s	
		Pre-Create Thread	**Thread per Request**	**Pre-Create Thread**	**Thread per Request**
Request Numbers / 15s	**1**	881	553	118	117
	2	928	471	149	159
	3	1030	536	159	160
	4	1051	511	132	146
	5	948	504	145	177
	6	963	497	155	173
	7	1083	504	122	163
	8	965	514	174	168
	9	938	520	174	185
	10	969	497	163	193
	11	1000	519	191	216
	12	945	514	157	197
	13	987	497	151	190
	14	1098	504	144	154
	15	995	505	135	185
	16	1082	500	124	201
	17	978	510	134	185
	18	961	495	129	142
	19	934	480	133	144
Throughput (Requests/s)		66.13	33.79	9.79	11.42