Scalability Test Report For HW4

This report will mainly talk about the scalability performance of our server code of Homework 4. As required, it will mainly have three parts, How did we write the test and how to reproduce results and what are the results?

Firstly, How did we write the test? Because we finished this homework in language C++, we also need to write test in C++. We have designed 11 customized XML files for testing server code functionalities which reside under functest directory of testing directory. We provide two test files, one is called functional_test.cpp, other is called scalability_test.cpp For functional_test.cpp, we just barely send the 11 customized XML files in order to the server, and receive the response from the server and display the XML response to check if it is correct. For scalability_test.cpp, it haves the code for testing the scalability of our server. We just sequentially send 1000 customized XML requests per each turn to the server and see the response. After each turn, we will calculate the execute time of one turn. For changing the number of cores that our sever will be running on, we used taskset command to set the number of cpus of the running process.

Secondly, How to reproduce the results? First step, change directory to testing, and compile the two files: functional_test.cpp scalability_test.cpp separately, and for functionality test, you can just run the binary file compiled from functional_test.cpp, you can see the results returned by each request displayed in the terminal accordingly. For scalability test, to test single thread version, just run the binary compiled from scalability_test.cpp , and after it finishes, you can see the running time displayed at the bottom. To test concurrency, you can open multiple sessions of VM, and test scalability_test.cpp concurrently. You will see that this server may never crash.

Thirdly, what are the results and what are the meanings of them? As required in the homework statements, we test our server code in a reserved 4 core VM, through using taskset command(find the running process first, and change the cpu assignment of it) , we can let the process run on 1 or2 or4 VMs, First of all, surprisingly, the performance results of our server code on 1,2,4 core VMs are quite similar. Here are the results with corresponding fields of our tests.

For testing on 4 cores:(we have 1000 requests generated from our XML file per turn)

Turn Number	1	2	3	4	5	6
EXECUTE_TIME	124seconds	138seconds	142seconds	150seconds	137seconds	145seconds

So, the average execute_time of our server on 6 runs is 139.3 seconds, that means the average latency is 139.3/1000=0.1393 , the throughput of our server is 1000 requests/139.3 seconds=7.18 requests/seconds. It is highly possible that the results may be vary when you run it second time. And Here we just run the test in one VM session.

For testing on 2 cores(we have 1000 requests generated from our XML file per turn)

Turn NUmber	1	2	3	4	5	6
EXECUTE_TIME	122	145	138	150	155	142

So, the average execute_time of our server in 6 runs is 142 seconds, that means the average latency is 142/1000=0.142 , the throughput of our server is 1000 requests/142seconds=7.04 requests/seconds. It is highly possible that the results may be vary when you run it second time. And Here we just run the test in one VM session.

For testing on 1 cores(we have 1000 requests generated from our XML file per turn)

Turn Number	1	2	3	4	5	6
EXECUTE_TIME	138	145	155	155	142	140

So, the average execute_time of our server in 6 runs is 145.83 seconds, that means the average latency is 145.83/1000=0.14583, the throughput of our server is 1000 requests/145.83seconds=6.86 requests/seconds. It is highly possible that the results may be vary when you run it second time. And Here we just run the test in one VM session.

As previously stated, the test result above is about the scalability test, and the way we test it is to run the program just in one VM session, and the results may be or must be varied when you run it a second time. From the results, not only we have analyzed the latency and throughput, we have also learned a lot of things about multi-threading. As you can see from our code, we put a big lock on the entire part of XML and database processing, which may cause a huge amount of time. However, apart from the results given, when we remove all the locks added and run it again, the performance are still similar. We think the reason is that the intrinsic algorithm of our server is over complex which can take a lot of time wherever the locks are. Also, if we run the server in multiple VM sessions, we can see that whether we remove the locks of our threads or not, the server will run a long time even longer than single thread version. We think that is because muti-threading may can taken lots of system resources and increase the execute time not as expected. Fortunately, however we run the server, because we use a big lock on the parsing and database, our running time is quite slow but our response is definitely accurate. In conclusion, our program will finish 1000 requests in a reasonable time frame, however, latency and throughput sees unsatisfactory compared to great works. Which may be the reason of inner algorithms.

In short, our server code can behave as expected (except slower) and run successfully in VM across different number of cores. Which shows a good scalability but the execute_time still needs to be short.