Overview

To test the scalability of our system, we used the “randomize_test” program to test it.

This program used the following parameters:

1. The number of threads
2. The number of users created by each thread
3. The number of orders for each user created in each transaction
4. The number of symbols
5. The number of orders for each user created in each transaction
6. The number of cancels for each user created in each transaction

This program can measure the scalability of our server by outputting the following data:

1. Overall running time: the running time of the client program
2. Overall request: the total number of request sent (calculated as thread_num*(1+user_num))
3. Average latency: The average time of start to end for each thread
4. Overall throughput: How many requests can be handled in one second

To increase the successful rate of request, our program follows the following steps:

1. Create a random group of symbols according to the input number.
2. For each thread, create a random group of users according to the input number
3. Send the create message, which contains user create part and symbol create part. For the user create part, each user would be given a random balance. For the symbol create part, each user would have a random amount of each symbol.
4. For each user created in the thread, send a transaction request contains (1) Multiple random orders (random symbol, random amount, random limit) according to the input, (2) Multiple random queries according to the input number, (3) Multiple random cancels according to the input number.

The default basic setting for our scalability test is:

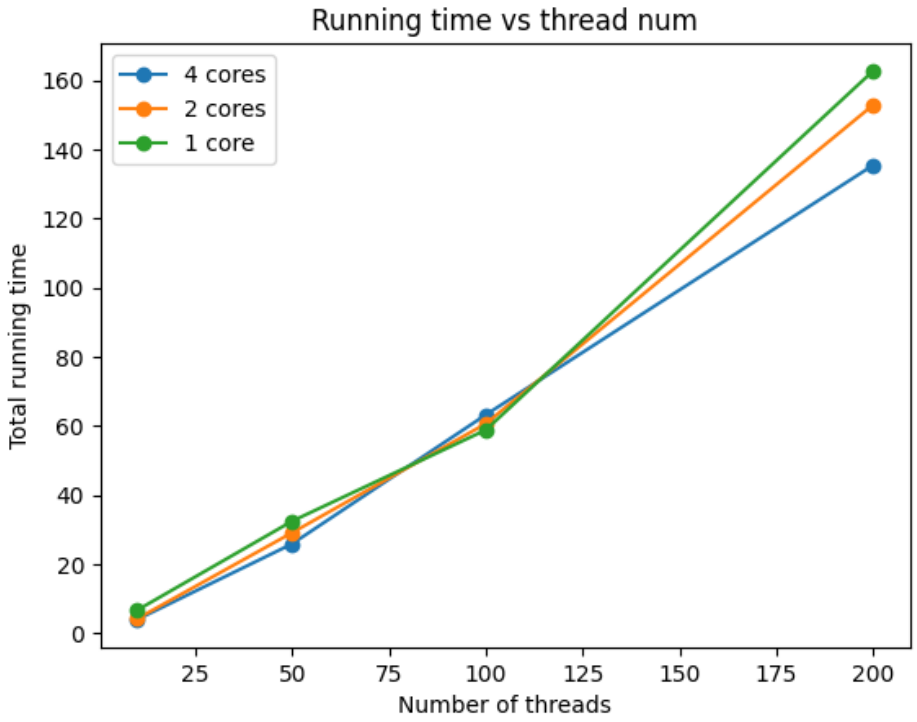
(1) Thread_num: 10
(2) User num per thread: 10
(3) Symbol num: 10
(4) Order num per user: 10
(5) Query num per user: 10
(6) Cancel num per user: 10

As all the orders, queries and cancels for each user would be contained in only transactions. We focus on the influence of changing thread number, and user number per thread to test the scalability of our system. Therefore, to test the scalability of our system, we changed the thread_num and user num per thread, and run the testing program under different number of cores. For each test scenario, we ran multiple times, and draw the plot according to the average value.

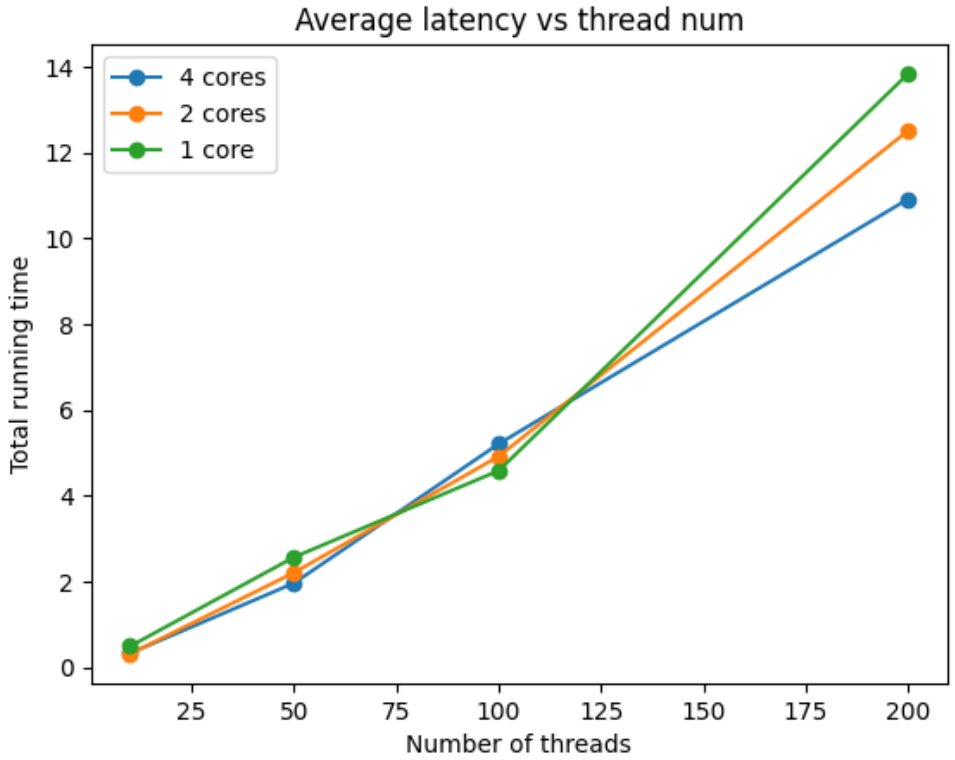
Changing thread number

When changing the number of threads, we chose 10, 50, 100, 200
The number of requests would then become: 110, 550, 1100, 2200

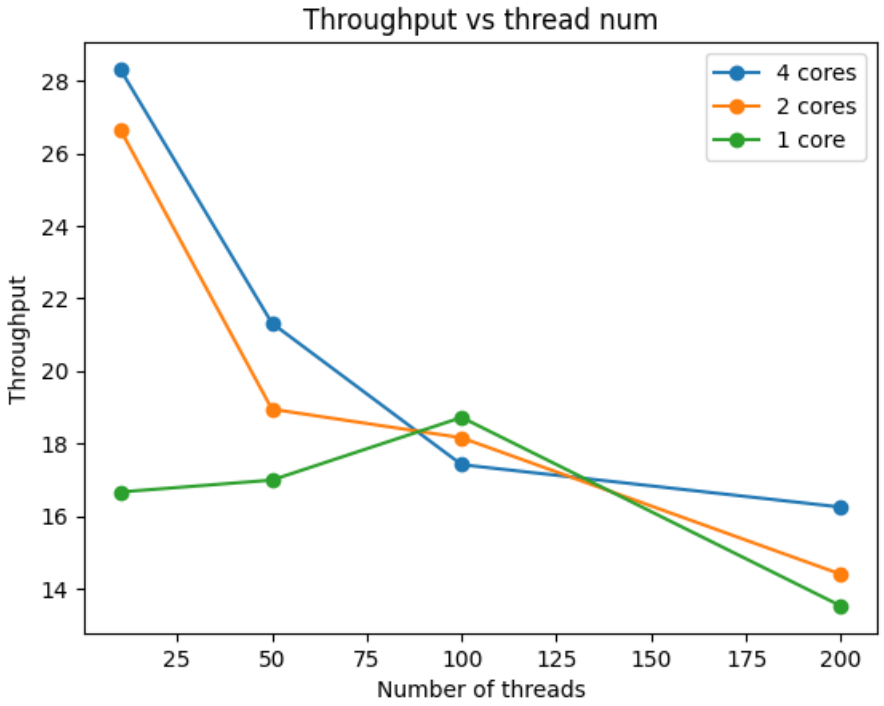
1. We can draw the plot regarding the running time and number of threads.



2. We can draw the plot regarding the average latency and number of threads.



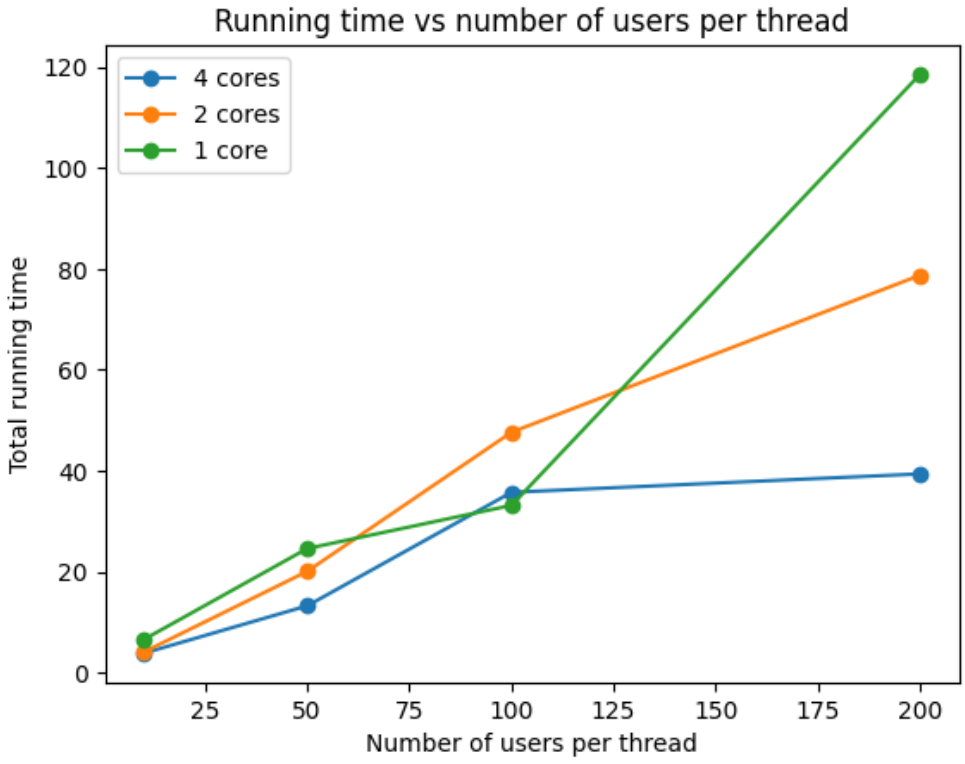
3. We can draw the plot regarding the overall throughput and number of threads.



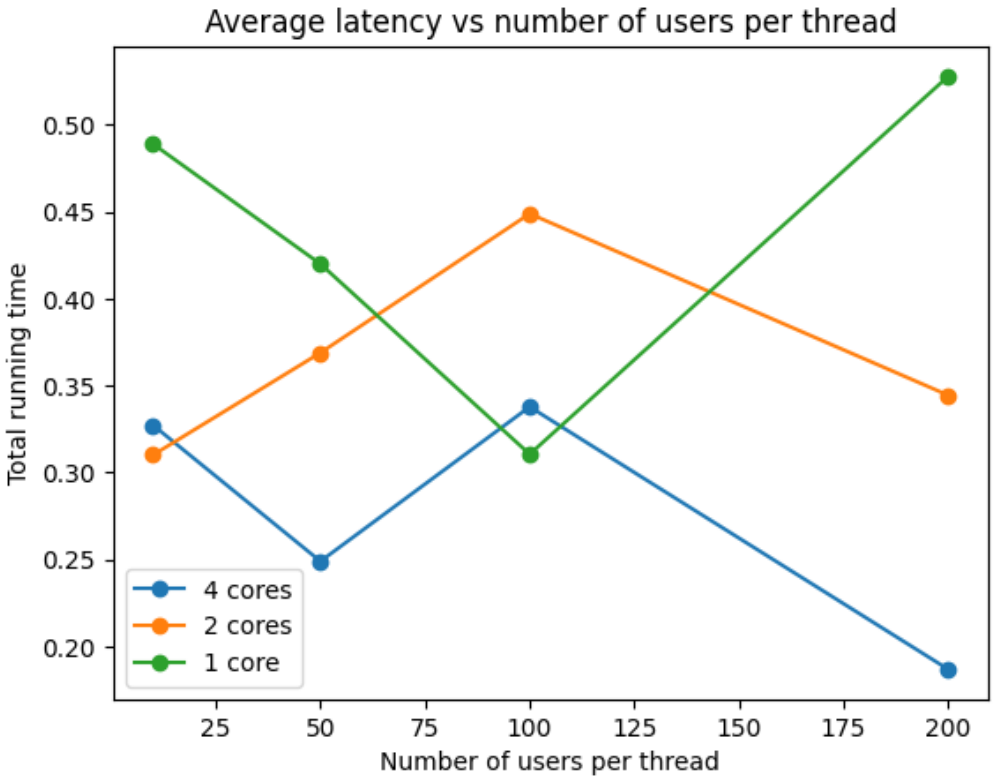
Changing number of users per thread

When changing the number of users per thread, we chose 10, 50, 100, 200
The number of requests would then become: 110, 510, 1010, 2010

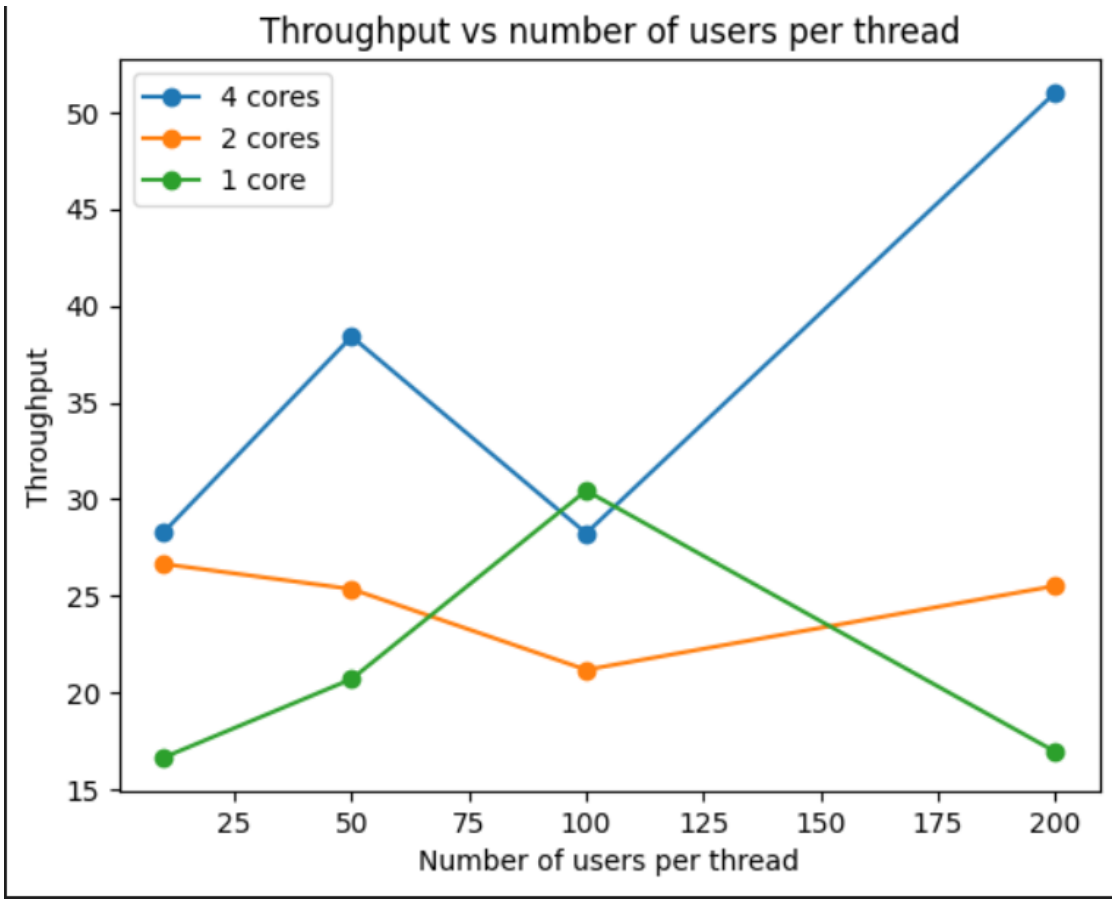
4. We can draw the plot regarding the running time and the number of users per thread



5. We can draw the plot regarding the average latency and the number of users per thread



6. We can draw the plot regarding the overall throughput and the number of users per thread



Conclusion

Generally speaking, we can find out that, with more cores, our system achieves shorter running time, lower latency and higher throughput. These shows that our system has good scalability.

Besides, we can also find that, although by changing the number of threads and number of users, the numbers of total requests are similar. However, we can find the performance of more threads is much worse than more users. We think this may because the resource competition regarding multi-thread.