Report

Ruolin Xu(rx50)
Zhihao Zou(zz312)

Introduction:

In this assignment, We're currently evaluating the scalability of a Stock-exchange software. This program has been built using C/C++ and employs Postgresql as its primary database. Users can create accounts, add symbols, and perform various actions such as executing transactions, querying, and cancelling their existing transactions. The system can also help match the sell and buy transactions and record the transactions history as executed.

Scalability:

To test the system's scalability, we employed two strategies. Firstly, we checked the single client to ensure that the matched orders were correct and that the account and shares balance were updated accordingly. To conduct this test, we ran the container and then the client in another terminal, using the command "./client IP_address 1," replacing the IP_address with the server's IP address and the content with the transaction we desired. We found that there were no issues with a single client, and all transactions were correct.

Next, we conducted the second strategy by flooding the server with multiple clients. Specifically, we tested the system with 1000 clients simultaneously sending requests to the server, with all clients wanting to perform the "create" transaction. To conduct this test, we used the command "./client IP_address 1000".

Compared to other programming languages, C++ offers native support for multi-threading, allowing for more efficient parallelism and scalability in multi-threaded applications. Multi-threaded C++ applications can fully utilize multiple CPU cores, resulting in better performance for CPU-bound applications that demand a lot of processing power. Additionally, C++ provides a rich standard library for managing threads, locks, and condition variables, enabling the development of concurrent applications with a high level of consistency and data integrity.

To test the performance of our C++ Exchange Matching Server, we measured the number of requests processed per unit of time using 1000 client-initiated requests. We ran the test five times and recorded the average processing time. The results demonstrated that the server's throughput increased as the number of threads was increased, indicating better utilization of the available CPU cores. However, it is important to note that the improvement in performance was not significant, which may be due to factors such as contention for shared resources, cache thrashing, or poor load balancing. Further load testing scenarios with different types of requests and response times would provide a more comprehensive evaluation of the server's performance.

Result analysis:

The server's response time when dealing with 1000 concurrent users can be seen in the following result:

Core	Time(s)
1	5.34422
2	5.36592
3	4.1389
4	6.62184

We analyzed the results and tried to figure out why the time of server with more cores not significantly efficient than the one with less cores:

1. The overhead of managing multiple threads across multiple cores can introduce additional complexity and overhead, which can offset any potential performance gains.
2. The application may be I/O bound rather than CPU bound, meaning that the threads are waiting for input/output operations to complete before moving on to the next task. In such cases, using multiple cores may not provide any significant performance improvements.
3. The threads may be competing for the same resources, such as memory or disk access, which can cause contention and slow down the application. In such cases, using multiple cores may actually degrade performance due to increased contention.

4. The application may not be designed to take full advantage of multiple cores. For instance, it may be poorly optimized or have inefficient thread synchronization mechanisms, which can limit scalability.