Scalability: Exchange Matching Report

Engineering Robust Server Software Homework 4

az161 rh328

Introduction

For this assignment, we build an exchange matching engine system which can process multiple requests at the same time. We use C++ as our programing language and postgresql as the database. This system can operate user requests such as create accounts, add balance, add symbol amount, make orders, query and cancel orders. All these requests can be operated parallelly. A single thread server with just one database connection can handle user requests one by one easily. To handle multiple threads at the same time, we implemented multithreading, thread lock and database isolation serialization in our system.

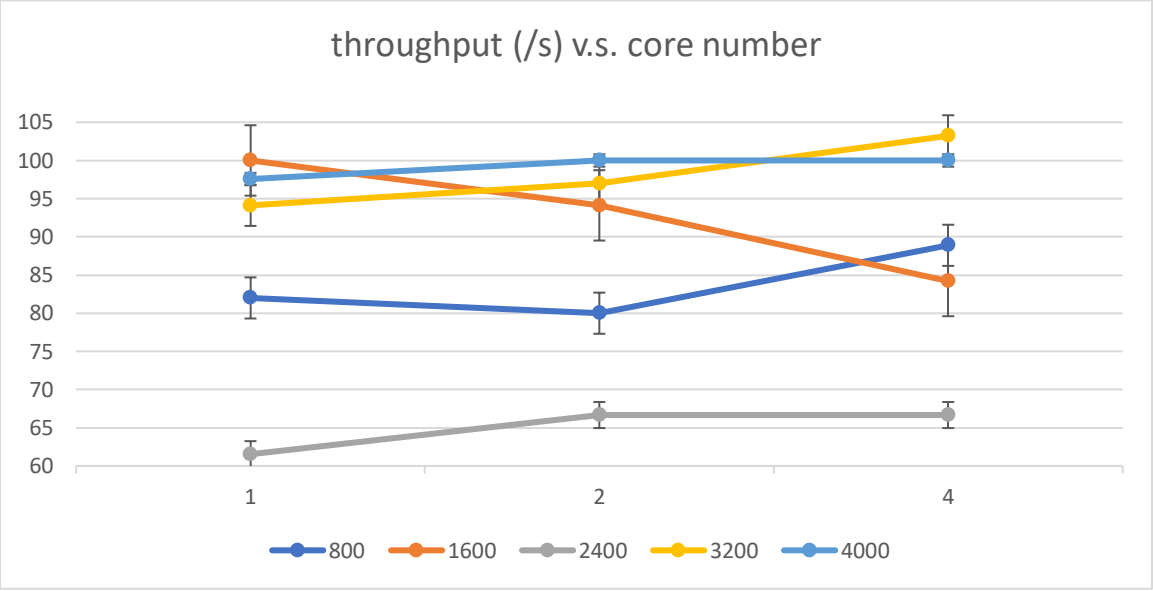
For database side, to avoid read-modify-write cycles, we applied row level locking and SERIALIZABLE transactions to make it run with isolation. For server side, we create a single thread to handle the request every time we receive a request. For each request, we implement a database connection to allow each thread to access the database.

We used two test methodologies to test this system, one is functionality test and the other one is scalability test.

Test Methodology

Single thread

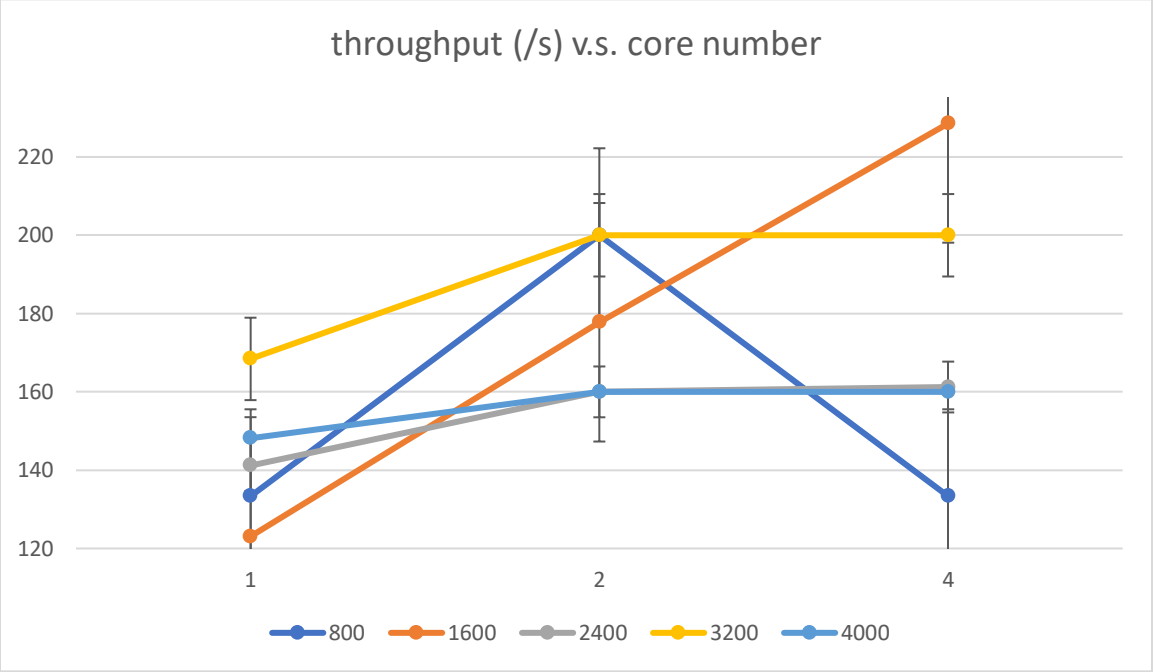
For Scalability test, we have first written test_func.sh which run 8 different request 100, 200, 300, 400 and 500 times respectively with different number of cores. In other words, for each test, we run 800, 1600, 2400, 3200 and 4000 requests with 1, 2 or 4 respectively cores one by one. That is, one client only sends one request and then the next client send another.



The performance of 2 cores is usually better than the one core version, and 4 cores version is usually better than 2 cores version with more than 2400 requests. The performance does not follow the trend for 800 and 1600 requests cases may because that the workload is insufficient: The performance boost could not be significant if the workload is insufficient to utilize all the cores. More cores might not make much of a difference in these circumstances.
The performances are very similar for each test cases. The reason may be that managing several cores is a major administrative burden: Scheduling, synchronization, and communication overhead increase when managing several cores. Increasing the number of cores may not significantly enhance performance if this overhead is substantial. The programs performs well in 3200 and 4000 cases, the reason can be that the workload is still not sufficient for the performance to decrease.

Multiple thread

For Scalability test, we have then written test_scala.sh which run 8 different request 100, 200, 300, 400 and 500 times in the background respectively with different number of cores. So multi-thread is applied here.



First of all, it is obvious that the throughput is improved compared to single thread version. Besides, the performance does not follow the trend for 800 requests cases may also because that the workload is insufficient. 4 cores performs better than 2 cores in other cases. Additionally, there may be a performance bottleneck in the software sometimes: If the program has a performance bottleneck, adding more cores may not be beneficial. For instance, increasing more cores could not have a substantial impact on speed if the application spends the majority of its time waiting for response output operations. On average, the program performs the best in 3200 case and the performance gets worse with larger throughput. In conclusion, the program performs well in comparatively large throughput.