Homework 4 Report

Yuzhe Ding (yd160), Jiaying Lyu (jl1000)
April 1, 2022

1. Introduction

We have designed and implemented a working version of an exchange matching engine, a piece of software which will match buy and sell orders for a stock/commodities market. For databases, we use PostgreSQL to store and retrieve order data. For XML parsing and generating, we use the rapidXML library. For communication, we use sockets to establish connection between client and server.

2. Design

Basically, we design a server to receive the XML request from the client. After receiving the request, firstly we judge whether the length and the format of the received request is correct. If it is not correct, we don't process that and respond with a default error message. Otherwise, we will determine whether it's XML of creating an account, creating symbols or transactions, and parse them in different ways.

If the XML is about creating an account, we will select the relative record of the same account id and insert the record into the database. If the XML is about creating a symbol, if the symbol for this account already exists, we just update shares in the database, otherwise we create a new position entry.

When we deal with a transaction of a buying order, we firstly select the record according to the account id, and then update the amount of the symbol. if the price is greater than it should be when it is first placed, we will refund the buyer with the redundant money, and then add a new executed order. If the amount to buy is 0, delete the original order, otherwise update the original order instead of deleting. When we deal with a transaction of a selling order, we firstly change the seller's account balance and then add a new executed order. After that, if the selling amount is 0, delete the original order, otherwise update it.

3. Scalability Analysis

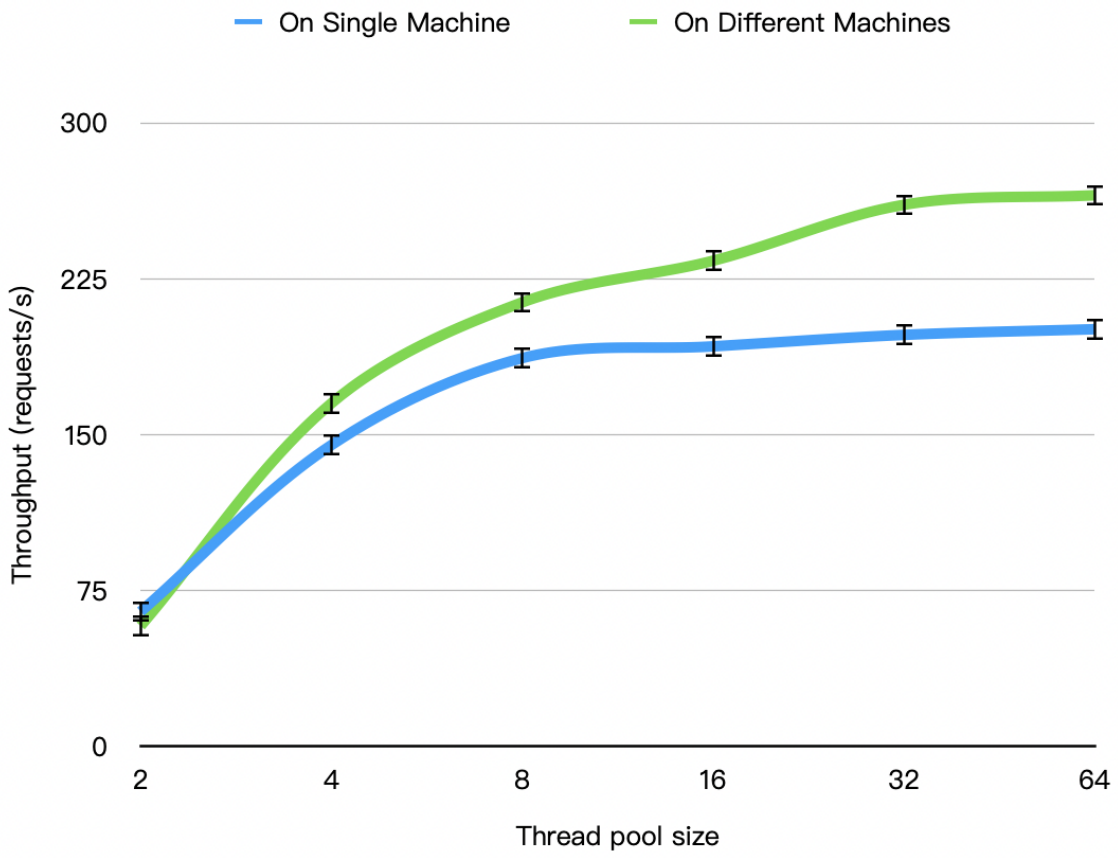
We use thread pool as a multithread strategy, and we use atomic operation of transaction and serialization. For data storage, we create three tables to store order, position and account. The program will perform well when dealing with a large number of symbols, since the transaction of each symbol is isolated.

(1) Thread Pool Experiments

Thread Pool Size	Request#	Throughput (requests/s)	Calculation Time (s)
2	10000	64.8668	154.162
4	10000	144.991	68.97

8	10000	186.815	53.529
16	10000	192.456	51.96
32	10000	197.977	50.511
64	10000	200.711	49.823

In this graph, we calculated the relationship between the throughput and the size of the thread pool (number of threads allocated) with a fixed requests load (10000 requests in total). From the first two lines of the table, the throughput doubles with twice the thread pool size, which is reasonable since we are processing twice more tasks compared to spawning just 2 threads. However, as the thread pool size keeps increasing, the throughput improves much slower. The reason for this is that the vcm only contains 4 cores, thus 8 HW threads, and all the threads are contending for these limited resources. The performance gets saturated when there are 64 threads working. The result is plotted in the following graph in blue.



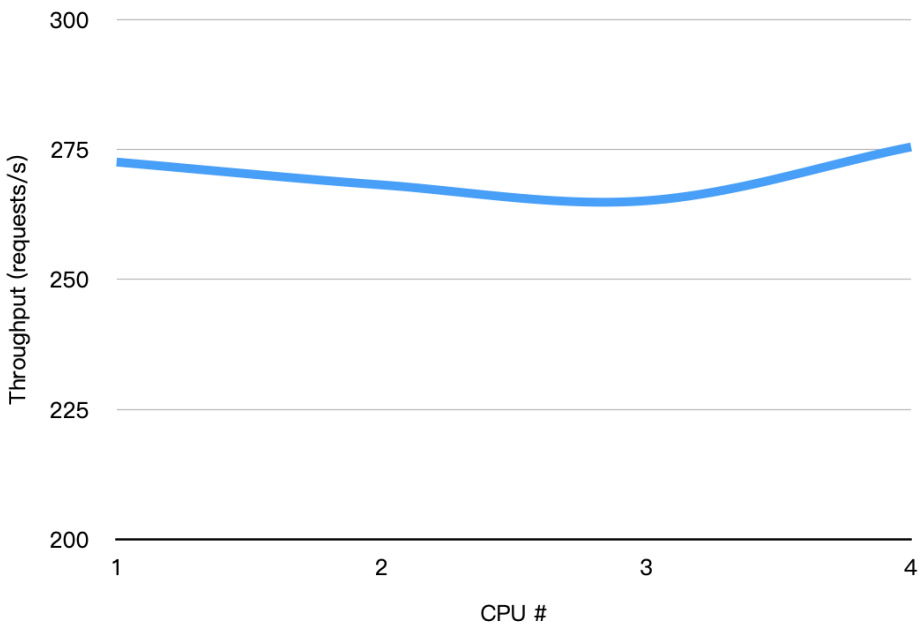
Thread Pool Size	Request#	Throughput (requests/s)	Calculation Time (s)
2	10000	57.9656	172.516
4	10000	165.117	60.563
8	10000	213.634	46.809
16	10000	233.672	42.795

32	10000	260.559	38.379
64	10000	265.161	37.713

In the previous experiment, we carried out both sending, receiving, and processing the requests on a single virtual machine, which may not be the case in real life. Namely, the client requests should be sent from other machines. So, this time, we dispatched the server and clients, and did experiments again. As you can see (shown in green line), the throughput has improved compared to the previous one, and the trends of the relationship between the throughput and the thread pool size remains the same.

(2) Thread Pool Experiments

CPU#	Thread Pool Size	Request#	Throughput (requests/s)
1	64	10000	272.598
2	64	10000	268.204
3	64	10000	265.154
4	64	10000	275.512



In order to test the scalability, we set different CPU affinities to our server while keeping thread pool size, request load fixed.

Specifically, we used the following commands to bind certain number of CPUs to our server:

- taskset -p 0xf [PID] -> CPU# == 4
- taskset -p 0x7 [PID] -> CPU# == 3
- taskset -p 0x3 [PID] -> CPU# == 2
- taskset -p 0x1 [PID] -> CPU# == 1

It’s pretty strange that the throughput remains stable no matter how much CPUs are put in use. We think the reasons are:

(1) We simply didn’t generate enough requests at any given time for all CPU cores. We only have one virtual machine for sending requests.

(2) The program we’ve implemented for automatically generating xml requests gives such random data that most of them are invalid requests. More specifically, for transaction requests, the account id should exist in the database and the symbol of each sell order should be possessed by this account, otherwise, the transactions are responded with error results. As a result, the processing speed is so fast that the request load sent from clients is not enough to keep our server busy. The performance would not be affected by the number of CPU cores