Homework 4: Scalability: Exchange Matching

Lijing Yin ly161
Iris Zhang mz197

Design and Implementation

- **Datebase:**
    1. db.cpp - This file is basically dealing with the creation, insertion, destruction and query of our database tables by using PostgreSQL. The main function getMatching() is the key point of the exchange matching. We first get two tables from open_order and select queries with some constraints such as seller_price <= buyer_price. And the findMatch() function will keep executing matching pairs.

    2. handle.cpp - The handle file contains the actions functions after checking Create/Transaction, including error checking/handling and output printing. Reading XML input is also implemented in this file. We use a XML parser, “Tinyxml” to parse the input file.

- **Request/Response:**
    1. server.cpp - On the server side, we implemented two strategies to handle multiple requests sent by clients, including thread pool strategy and multi-threading strategy. Firstly, we call an open-source thread pool library to process these requests. We can set up the size of the pool, representing the maximum number of threads to handle the requests and send back the responses. Secondly, we used multi-threading, which generates a thread as needed when it handles a new request of a client.

    2. client.cpp - On the client side, we used two ways to test the runtime of the request. The first one is to input one argument, the name of the XML file, which creates one request. The second one is to input two arguments, the flag of multi-threading and the name of the XML file, which creates multiple requests for the XML file. And we also include a time calculation function to measure the time spent processing this request.

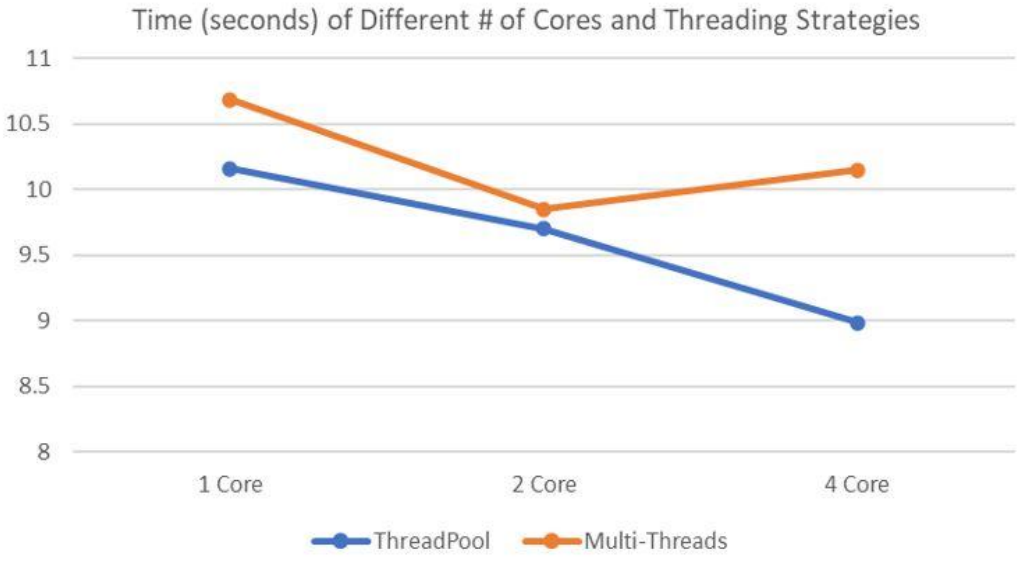
- **Multi-thread Design:**
    1. Thread Pool: We use an open source “ThreadPool.h” to pre-create some threads. The thread pool maintains multiple threads. It not only ensures full utilization of the core, but also prevents over-scheduling.

But creating too many threads will waste a certain amount of resources, and some threads are not fully used.

Performance Result

Threading Method	# of Cores	Latency (1000 threads)	Throughput (every 10 sec)
ThreadPool	1	10.1573s	985
ThreadPool	2	9.6984s	1036
ThreadPool	4	8.9840s	1113
Multi-Threads	1	10.6845s	936
Multi-Threads	2	9.8526s	1015
Multi-Threads	4	10.1499s	985

- **Different Cores**
  In theory, multiple cores will run faster. Based on our test, when using the ThreadPool strategy and the number of requests are the same, the one with more cores is better, with lower latency and larger throughput. Running time: 4 cores < 2 cores < 1 core. Throughput: 4 cores > 2 cores > 1 core. The actual test results match expectations.



But for multi-threads, the result for four cores against the theory result. When a new thread is created, the machine will allocate an amount of resources. And this will cause more consumption. Speed trades consumption.
When using multi-threads strategy, plenty of threads will be created. Everytime calling a new thread the CPU affinity will be reduced. And the more threads are created the more CPU affinity factor influences. Such that, for multi-threads performance the effect of CPU affinity will exceed the effect of resource consumption. This will cause more latency.

- **Different Threading Method**
  The performance of ThreadPool is better than Multi-Threads. As mentioned before, ThreadPool has fixed size but Multi-Threads creates thread when needed. The consumption of Multi-Threads is greater than ThreadPool. Therefore, the result of ThreadPool is better.