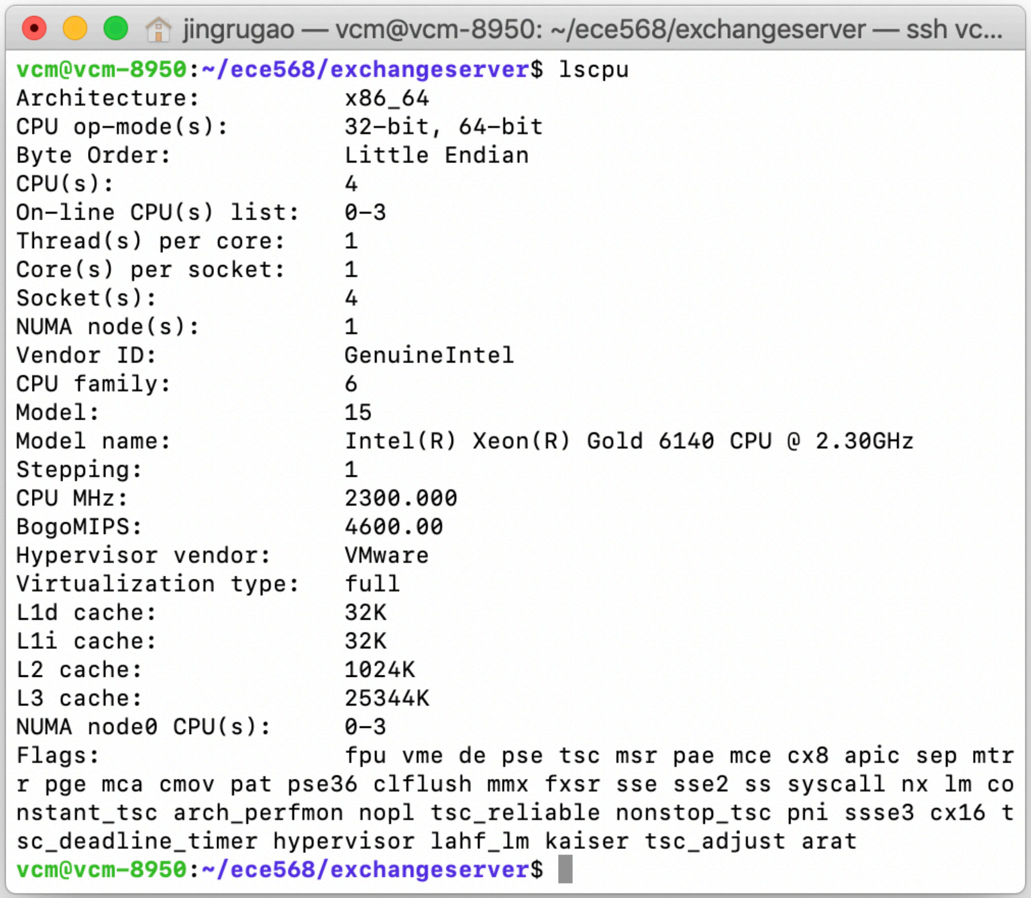
Exchange Server Report

Introduction:

The exchange matching server uses postgres as database. And we use multi-thread to handle requests. To reduce the overhead of creating threads for each requests, we planed to the pre-create thread while in the real situation we are still using create thread per request.

To test our server's scalability, we use the 4-core VMs to run our program. The information of the VM is listed below.


jingrugao — vcm@vcm-8950: ~/ece568/exchangeserver — ssh vc...

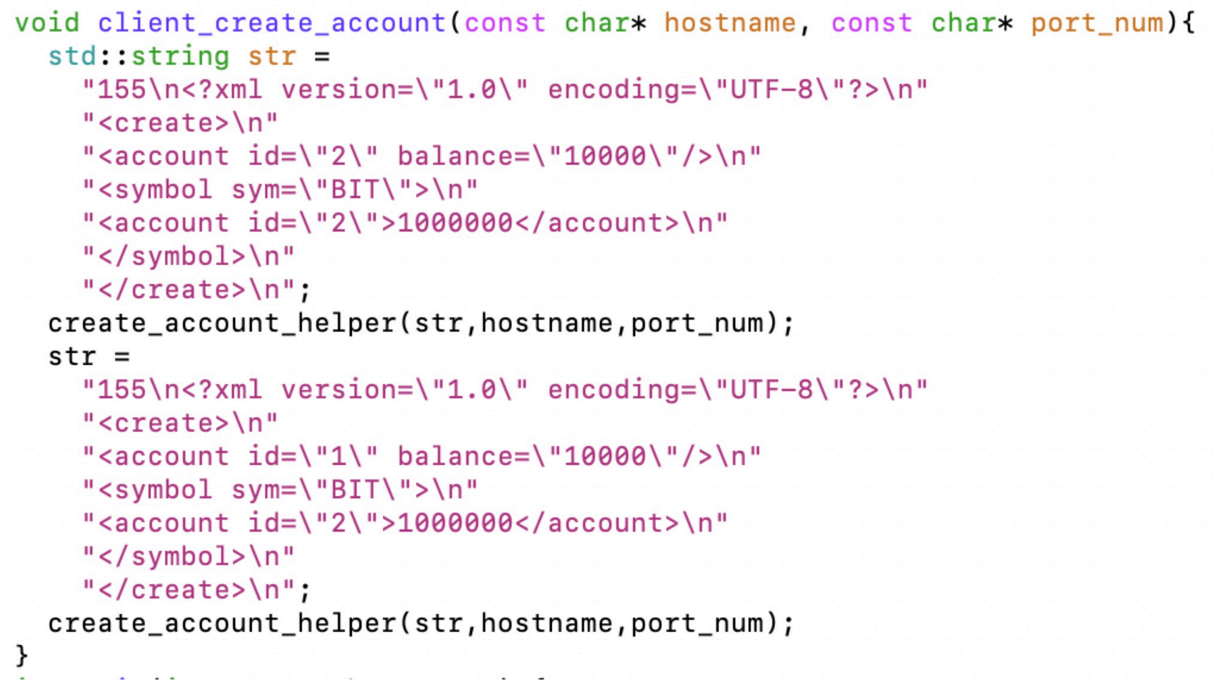
```
vcm@vcm-8950:~/ece568/exchangeserver$ lscpu
Architecture:          x86_64
CPU op-mode(s):        32-bit, 64-bit
Byte Order:            Little Endian
CPU(s):                4
On-line CPU(s) list:   0-3
Thread(s) per core:    1
Core(s) per socket:    1
Socket(s):             4
NUMA node(s):          1
Vendor ID:             GenuineIntel
CPU family:            6
Model:                 15
Model name:            Intel(R) Xeon(R) Gold 6140 CPU @ 2.30GHz
Stepping:              1
CPU MHz:               2300.000
BogoMIPS:              4600.00
Hypervisor vendor:     VMware
Virtualization type:   full
L1d cache:             32K
L1i cache:             32K
L2 cache:              1024K
L3 cache:              25344K
NUMA node0 CPU(s):     0-3
Flags:                 fpu vme de pse tsc msr pae mce cx8 apic sep mtr
r pge mca cmov pat pse36 clflush mmx fxsr sse sse2 ss syscall nx lm co
nstant_tsc arch_perfmon nopl tsc_reliable nonstop_tsc pni ssse3 cx16 t
sc_deadline_timer hypervisor lahf_lm kaiser tsc_adjust arat
vcm@vcm-8950:~/ece568/exchangeserver$
```

Here we can see that the CPU list is 0-3, and we have 4 logic CPUs, which is equal to "Thread(s) per core" * "Core(s) per socket" * "Socket(s)". One socket is one physical CPU package. This VM has 4 sockets, each containing a 1-core Intel® Xeon® Gold 6140 CPU, which has 18 cores and 36 threads.

The postgres database is the one the exchange matching server is continuously interacting with. Postgres has default isolation level which is read committed. Here we used the default isolation level. We checked the header file of libpqxx, the repeatable read tag is commented in default. So to avoid the potential resources competence in matching process (because we need to check the order and account info before execute the order), we use the lock to prevent violation of data.

Test cases:

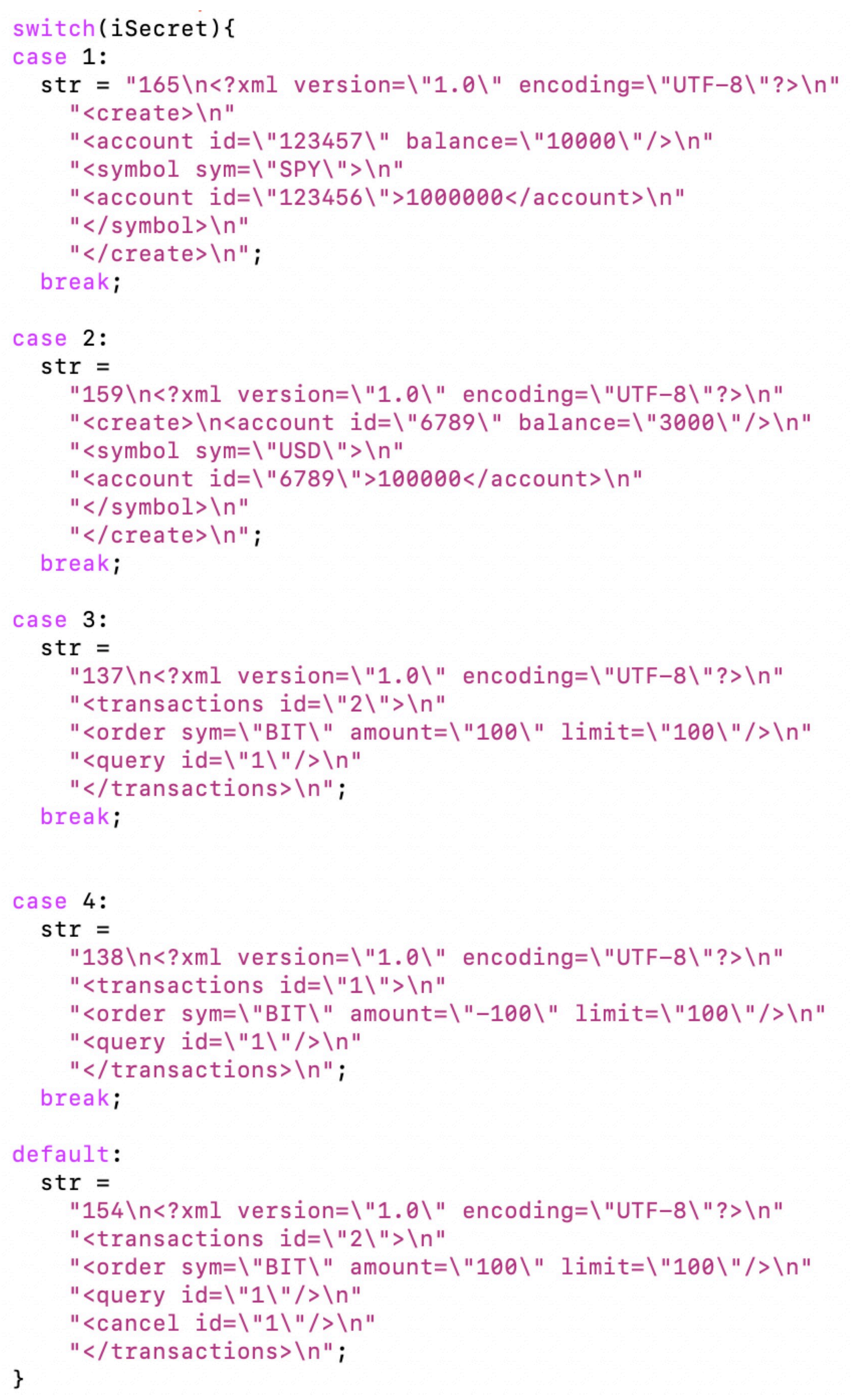
In the testing (client) program, we create two account first, which are not calculated in the total run time of the program handling different requests. The XML requests are listed as below:



```
void client_create_account(const char* hostname, const char* port_num){
  std::string str =
    "155\n<?xml version=\"1.0\" encoding=\"UTF-8\"?>\n"
    "<create>\n"
    "<account id=\"2\" balance=\"10000\"/>\n"
    "<symbol sym=\"BIT\">\n"
    "<account id=\"2\">1000000</account>\n"
    "</symbol>\n"
    "</create>\n";
  create_account_helper(str,hostname,port_num);
  str =
    "155\n<?xml version=\"1.0\" encoding=\"UTF-8\"?>\n"
    "<create>\n"
    "<account id=\"1\" balance=\"10000\"/>\n"
    "<symbol sym=\"BIT\">\n"
    "<account id=\"2\">1000000</account>\n"
    "</symbol>\n"
    "</create>\n";
  create_account_helper(str,hostname,port_num);
}
```

After the creation of accounts, we use threads to send requests to exchange server. We use request per thread here, so there may have some overhead and latency for creating threads. We calculate the total time of all threads to complete, which includes sending data and get back the response and all threads reach finish point. We also redirect each thread's execution time and the total time into "output.txt" file to compare the results and draw graphs.

In each sending request thread, the program uses random number to choose from 5 different requests, and send it to exchange matching server. In the test, those 5 requests can meet each different conditions and database changes.



```
switch(iSecret){
case 1:
  str = "165\n<?xml version=\"1.0\" encoding=\"UTF-8\"?>\n"
    "<create>\n"
    "<account id=\"123457\" balance=\"10000\"/>\n"
    "<symbol sym=\"SPY\">\n"
    "<account id=\"123456\">1000000</account>\n"
    "</symbol>\n"
    "</create>\n";
  break;

case 2:
  str =
    "159\n<?xml version=\"1.0\" encoding=\"UTF-8\"?>\n"
    "<create>\n<account id=\"6789\" balance=\"3000\"/>\n"
    "<symbol sym=\"USD\">\n"
    "<account id=\"6789\">100000</account>\n"
    "</symbol>\n"
    "</create>\n";
  break;

case 3:
  str =
    "137\n<?xml version=\"1.0\" encoding=\"UTF-8\"?>\n"
    "<transactions id=\"2\">\n"
    "<order sym=\"BIT\" amount=\"100\" limit=\"100\"/>\n"
    "<query id=\"1\"/>\n"
    "</transactions>\n";
  break;


case 4:
  str =
    "138\n<?xml version=\"1.0\" encoding=\"UTF-8\"?>\n"
    "<transactions id=\"1\">\n"
    "<order sym=\"BIT\" amount=\"-100\" limit=\"100\"/>\n"
    "<query id=\"1\"/>\n"
    "</transactions>\n";
  break;

default:
  str =
    "154\n<?xml version=\"1.0\" encoding=\"UTF-8\"?>\n"
    "<transactions id=\"2\">\n"
    "<order sym=\"BIT\" amount=\"100\" limit=\"100\"/>\n"
    "<query id=\"1\"/>\n"
    "<cancel id=\"1\"/>\n"
    "</transactions>\n";
}
```

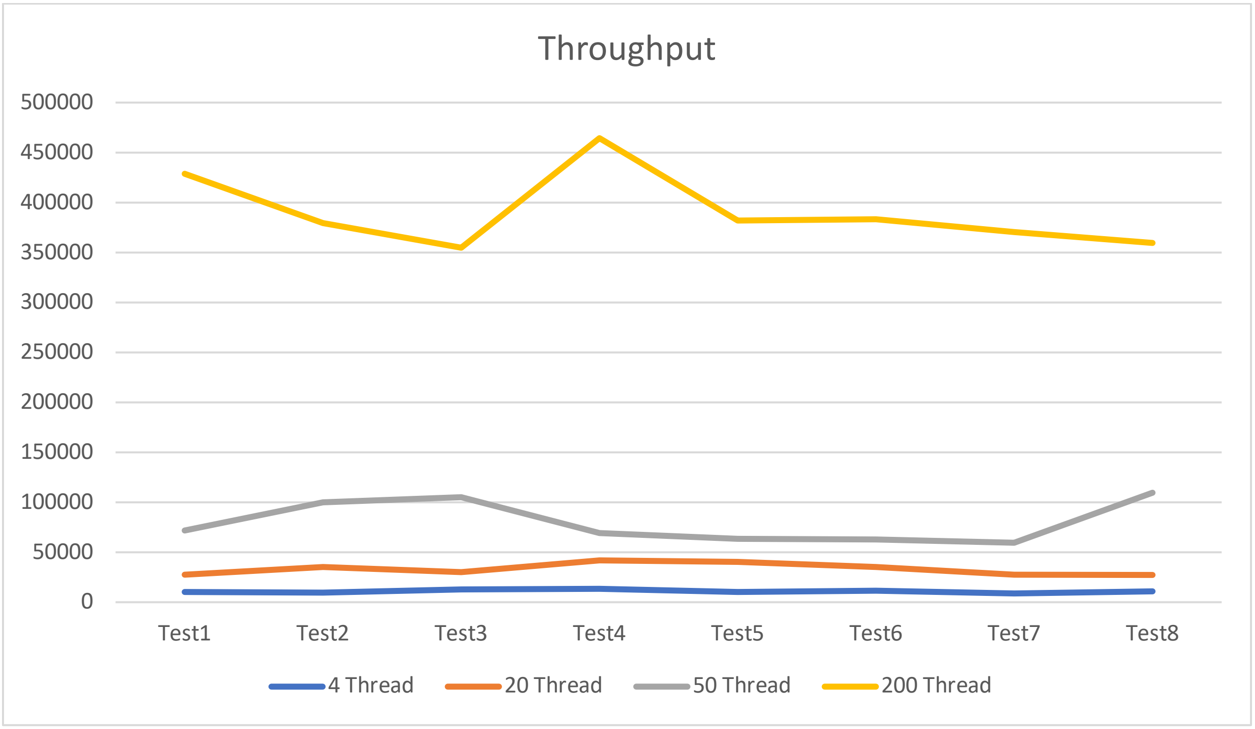
Results and Analysis

Same # of Server cores with different # of Client threads (Time is micro second)
(default setting, 4 cores theoratically)

Thread#	Test1	Test2	Test3	Test4	Test5	Test6	Test7	Test8
4	10366	9916	12881	13420	10245	11677	8767	11152
20	27645	35078	30014	41870	40327	35360	27812	27289
50	71999	100082	105251	69371	63228	62929	59519	109577
200	428999	379208	354763	464399	382183	383333	370698	359220

Computation:

Thread #	Average Time	Throughput(per second)
4	11053 micro second	361.89
20	33174.375	602.87
50	80244.5	623.10
200	390350.375	512.36

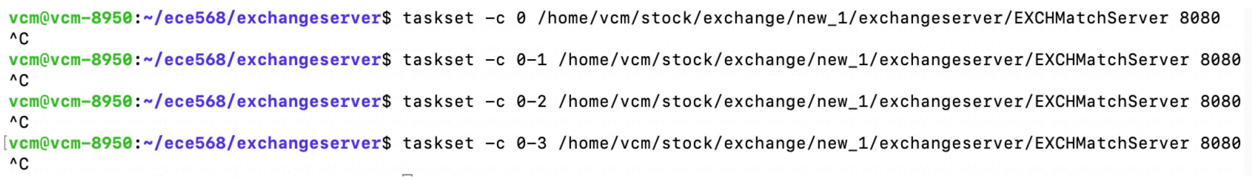


We use vector to create a thread pool for sending the requests. The reason why our server's throughput decreases when the thread number is more than 100 is that, we pre-create 100 thread in server program, and it always re-use those threads. When the thread number is small, the program's throughput is increasing with the number of threads. When it hit the bottleneck point(in the previous setting, 100 thread at the same time), the thread will wait for the

completion of all 100 requests and then to handle next 100 requests. Therefore, the increasing time is caused by queuing time.
Therefore, by increasing this bottleneck we can theoretically get improvement continuously.

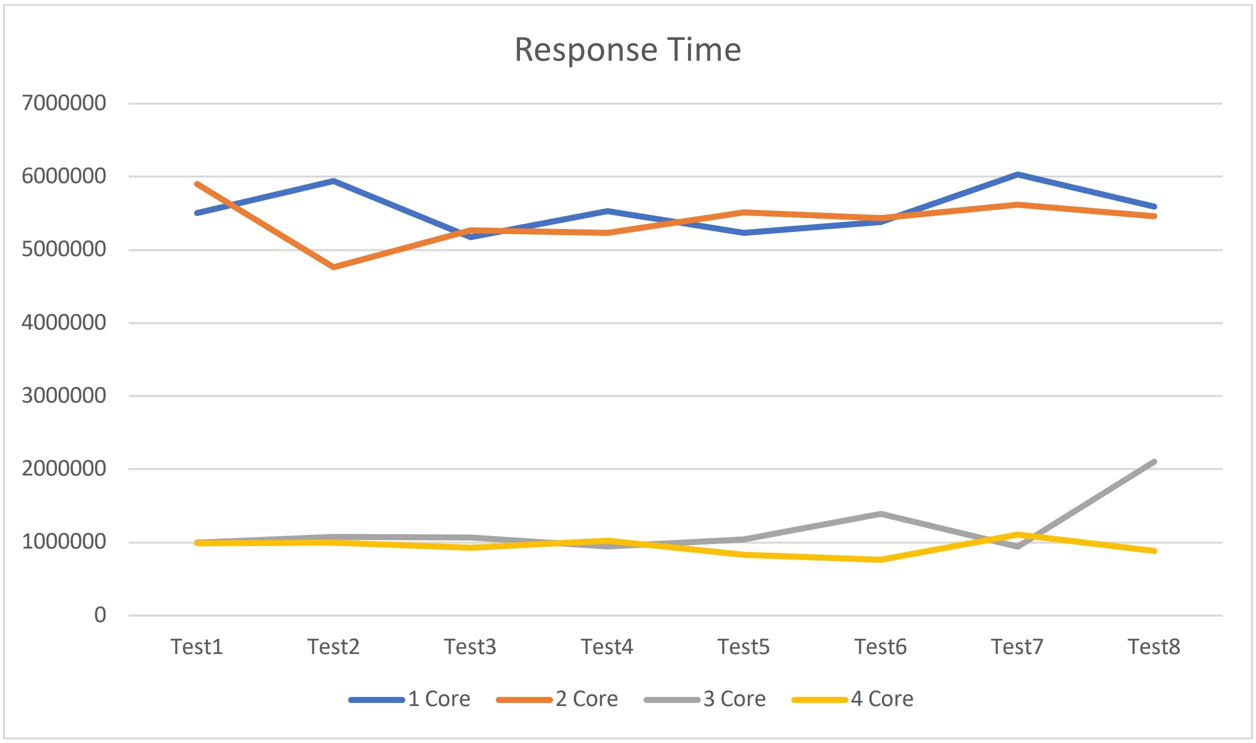
Same # of Client threads with different # of Server cores (2 threads, each run 1000 send request) (All number is micro second)

Core #	Test1	Test2	Test3	Test4	Test5	Test6	Test7	Test8
1	5502863	5938614	5174241	5530983	5236738	5384464	6031308	5587349
2	5901347	4762352	5264219	5231968	5514915	5429938	5620251	5460164
3	997645	1079469	1065780	947419	1037005	1390101	941778	2104325
4	990244	1001408	926457	1021931	832704	762995	1107481	880962



```
vcm@vcm-8950:~/ece568/exchangeserver$ taskset -c 0 /home/vcm/stock/exchange/new_1/exchangeserver/EXCHMatchServer 8080
^C
vcm@vcm-8950:~/ece568/exchangeserver$ taskset -c 0-1 /home/vcm/stock/exchange/new_1/exchangeserver/EXCHMatchServer 8080
^C
vcm@vcm-8950:~/ece568/exchangeserver$ taskset -c 0-2 /home/vcm/stock/exchange/new_1/exchangeserver/EXCHMatchServer 8080
^C
vcm@vcm-8950:~/ece568/exchangeserver$ taskset -c 0-3 /home/vcm/stock/exchange/new_1/exchangeserver/EXCHMatchServer 8080
^C
```

Here are the commands I used to limit the number of cores to use.



To test the response time of the server, we use two threads and we put a for loop in each thread to create more requests. The reason that we only use 2 threads is that we check the info of VMs, it only allows one thread per core. When testing one core situation, it works fine when we use 2 or 3 threads and run the test program and the server both on same 4-core VM. But when we tried to test the 4 core using situation, if we use the same VM, the test program will stuck a long time to manage between different threads. So, the first and second line of data is running the test and server program both on the 4-core VM. And the when testing using 3 and 4 cores, we use another 2-core VM to run the test part.

As you can see the above data and graph, there is difference between 1&2 cores and 3&4 cores. There is a huge performance improvement of 3&4 cores. The potential reason may be when using one core or two cores, there is much more scheduling in resources usage. And when it comes to 3 or 4 cores, the task is divided on different real hardware threads, there might also have some kind of scheduling process. But the overhead is much smaller compared with one and two cores. And you may also notice that there are some interleave parts in the graph. The reason behind this might be we use random number to decide which request to send, and for creating new account or position, there is no competition. But when it comes to creating an order to match and execute, or cancel the order, there may have more time to wait. So, the time was not ideally separate from each other.

And when we are testing, we used to print out each thread's execution time. And we noticed that some threads take longer than others, which might be it was waiting for resources.

And for the total threads' execution time, since we are doing time computation on testing side, not server side, the time may have a lot dependency on the number of threads we are going to create, which could cause similarity in time.

Other thoughts

1.Huge Test Problem:
In the first version of our testing function, there are several problems worth noting:

a. call thread.join() immediately after we create the thread, which means we only create a thread(make a request) after we finish the previous one. Therefore, we serialize our requests.

b. Using the multi-thread program described in a., we run our test program on the same machine where sever runs.

We found a strange phenomenon, when we used 2 cores, we got the best performance. When we used 4 cores, we got the worst performance. Why? It is contrast to our common sense.

After we realizing the problem 1, we found a way to explain problem 2. When we assigning 4 cores to server, there is no idle core for client program, therefore, client program must scramble for resources with server, which cause server run slower.

2.Performance Problem:
We came up with two solution:1.create thread per request 2. pre-create thread
Theoretically, the time to create thread will be in the critical path, therefore, the performance will not as good as pre-create thread.

However, in the actual situation, we didn't have a good implementation to solve the contention in pre-create thread. We used a queue to store current tasks, each thread would check this queue to fetch any available task. Std::queue is not a thread-safe data structure, therefore, we locked the critical section (fetch the front of queue and pop it). According to the result, this part lead to a contention which made sever slow extremely.

Therefore, we are still using the first solution.