report.md

# ECE 568: HW4 Exchange Matching Server

Authors: Jingheng Huan, Vincent Choo

## Introduction

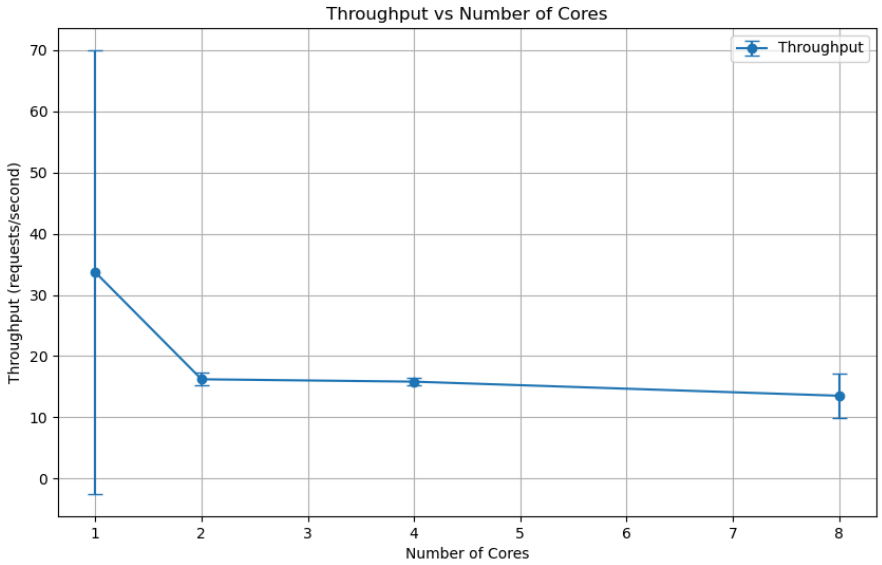
For this project, we created a stock exchange matching server using python. In our report, we will explore the scalability of our code by exploring how throughput and latency of our server changes as we increase the number of CPU cores. For our experiment, we used 1, 2, 4, and 8 cores.

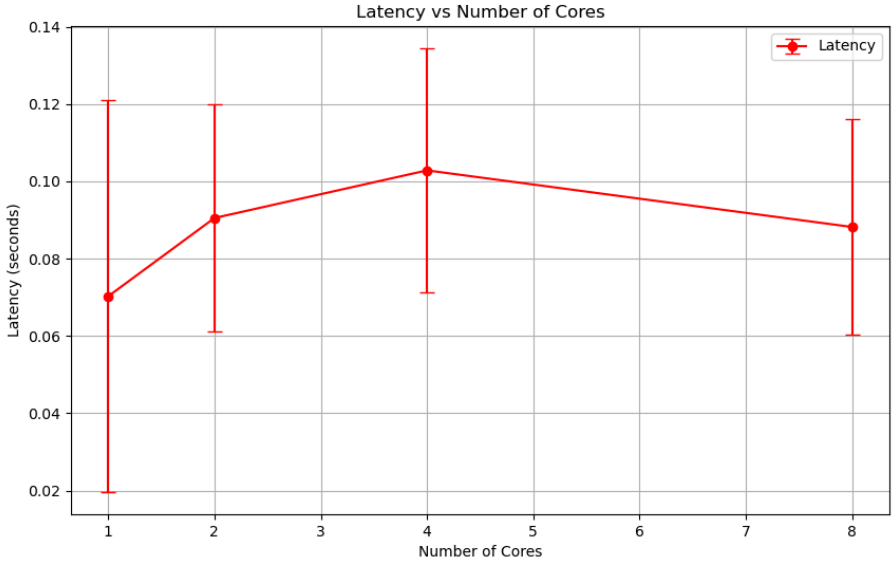
## Testing Methodology

To test our preformance, we used the linux command taskset -c to specify which cores to use to limit the number of core running the proceses when running the tests of the test for the different core counts. For our performance test, we created 100 random transaction request to the server and measure the average statistics (latency/throughput). This is ran 3 times for each core and the average and standard error is calcualted from the results.

## Test Results

For our tests, we had the following results:


Throughput vs Number of Cores
Throughput
Throughput (requests/second)
Number of Cores
0
10
20
30
40
50
60
70
1
2
3
4
5
6
7
8


Latency vs Number of Cores
Latency
Latency (seconds)
Number of Cores
0.02
0.04
0.06
0.08
0.10
0.12
0.14
1
2
3
4
5
6
7
8

# Analysis

From our experimental results, there was no significant performance improvement when increasing the number of cores used in the server. (No t-test was performed; this conclusion is based on the large standard error.) This suggests that our server does not scale well. Upon closer examination of our server code, we identified the following factors contributing to this limitation:

1. **Contention on Shared Resources** Our matching server uses lock to ensure that only one thread handles one resource at a time. However, it locks per request. Thus, having an extra core did not help much as it would just end of waiting for the other core to finish processing the client request before moving on to the next request from another thread. This contention limits the scalability of our server.

2. **Amdahl's Law** From amdahl's Law, we can see that the overall performance can be improved by speeding up process that can be done concurrently. Similarly, having large parts of the program that is sequential limits the total time we can save when optimizing the system and adding new cores. Since large parts of our system is sequential, this servers as a bottleneck for our mathcing server.

3. **Thread Management Overhead** Managing multiple threads can introduce overhead that diminishes the performance gains from an extra core, especially if the workload per request is not very heavy. Specifically, cores have to share resources through LLC. As a result, we can see that although the averages were not significantly different, the average throughput decreased slightly as we increased the number of cores.

4. **I/O vs. CPU Bound** Since each operation is not load heavy, the server is primarily waiting on I/O operations (such as network communication) rather than performing CPU-intensive computations. Thus, adding more CPU cores did not lead to significant throughput improvements.