ECE 420

Lab 2: A Multithreaded Server to Handle Concurrent Read and Write Requests

Lab section H42

Kelly Luc
1498694
Nicolas Serrano
1508361

Implementation methods:

The implementation is based on the simple client server. However the simple server needs to be modified so that it can handle multiple client requests. This can be achieved by using pthreads that run a function that handles the client's request. While there a multiple thread handle an client's request the array stored in the server must be able to handle race conditions and critical sections. This can be done by protecting the server array if there is a client that is either reading or writing to it. The lab requirements are to make 4 different implementations on memory locking.

Solution 1: Mutex locking the entire server array:

Within the critical section when the client requests to either read or write, a mutex lock should be placed and lock the entire server array. This ensures that only one client can access the server array. However this implementation provides sufficient locking, it is not as efficient compared to our other implementation. The runtime will be slower compared to locking an individual element in the server array.

Solution 2: Mutex locking elements in the server array

Create an array of mutex lock for each individual element in the server array. This can be achieved by initially creating a mutex array pointer and creating all the mutex lock of the array size that the server has. In the critical section, only lock the element in the server array that the client needs. This ensures that only one client can access a certain element in the server array, increasing its efficiency as more than one client can possibly access the server array.

Solution 3: Read and Write locking the entire server array

This solution involves using read write locks to protect critical sections of the code. For every read operation, a read lock is applied. Because this is a read operation, it will be safe for multiple threads to obtain read locks. For every write operation, a write lock is applied. Because this affects the data, only one write thread will be able to obtain the lock. By using these two locks, we can ensure that multiple read threads can read data at the same time, and overall can expect a reduced time in memory access latencies.

Solution 3: Read and Write locking the entire server array

This solution is similar to solution 3, except a read write lock is applied for each element in the array. Since there is a lock on each element, we can expect a reduced memory latency time as writing threads will be doing less waiting. A write thread will only have to wait if it is trying to write to a specific element that is already being written to by another thread. This change though will not be as drastic as going from solution 1 to solution 2, as solution 3 already allows read threads to obtain multiple locks.

Performance Discussion:

We expected that the performance of the first solution to be the slowest as it only allows one client to access the server array, while our second method to be significantly faster than the first. For the different methods of using mutex lock it can be seen that the method of using locks on each element of the server array had a quicker response time than locking the entire server array. As the number of strings in the server increases exponentially, our first solution showed a decrease in overall performance, while the performance of our second solution was consistent. This can also be further supported by the average memory access latencies. The first implementation had an increase in the average memory access latencies of 1.26E-1s, while the second method stayed roughly the same at 4.67E-3s. We also expected to see our third method to be slow but faster than our first solution and our fourth method to be the fastest. From our results of the solutions that used read and write locks, the third method was the second slowest in terms of average memory latencies. However it consistently had roughly the same average memory latency overall which was around 3.39E-2s as n increased. The fourth method was the second fastest solution for response time with an average memory latency of 5.01E-3s when n =10000. We expected this method to be the fastest but the results we obtained goes against our theory. This could likely be due to the 100 runs either having more requests to write than reads or that the threads that are reading are blocked by multiple writes.

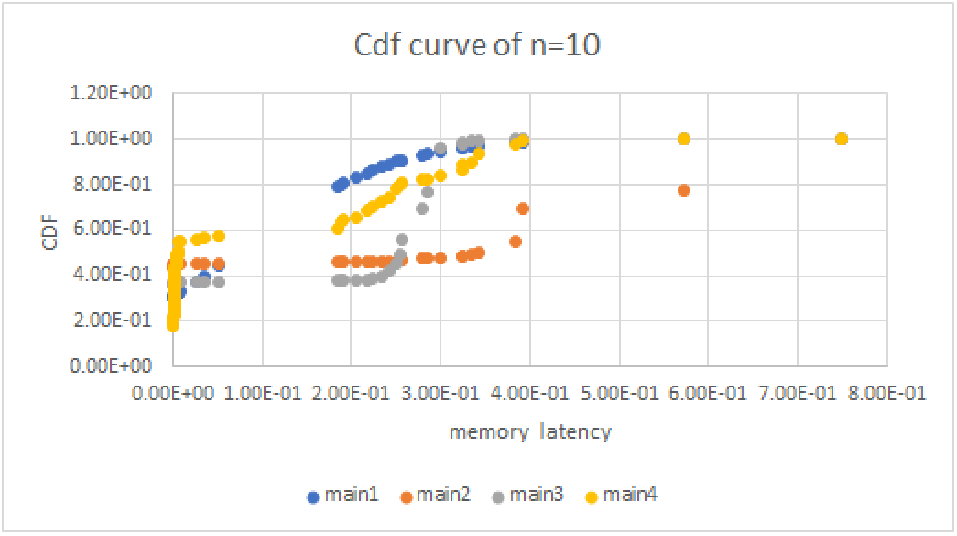
CDF Plot Discussion

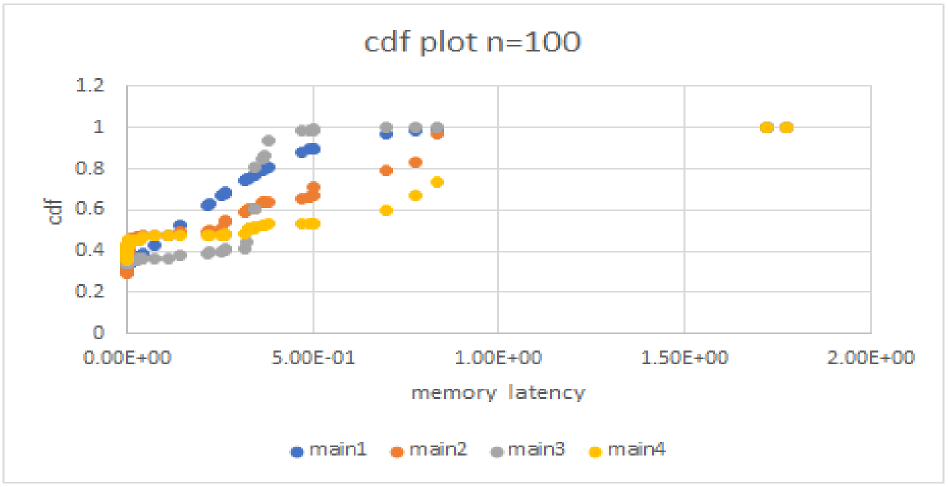
The Cumulative distribution function shows the probability of a memory latency time to occur. Using this function, we can use this to verify the performance of our implementations. For instance, for the single mutex solution(main1), we know that it is more likely to have a longer memory latency time then all our other implementations. If we look at our 4 cdf plots, the probability for the memory latency time to be fast is very low(the main plot is initially always lower than the other plots). As the memory latency times increase, the probability increases, meaning that our first solution is more likely to have a higher memory latency time. Our fastest average time of 2.60E-03s was achieved from the multi mutex array solution for 1000 string elements. If we take a look at our cdf plot for n=1000, we can see the main2 plot has a higher probability to be faster than all the other solutions.

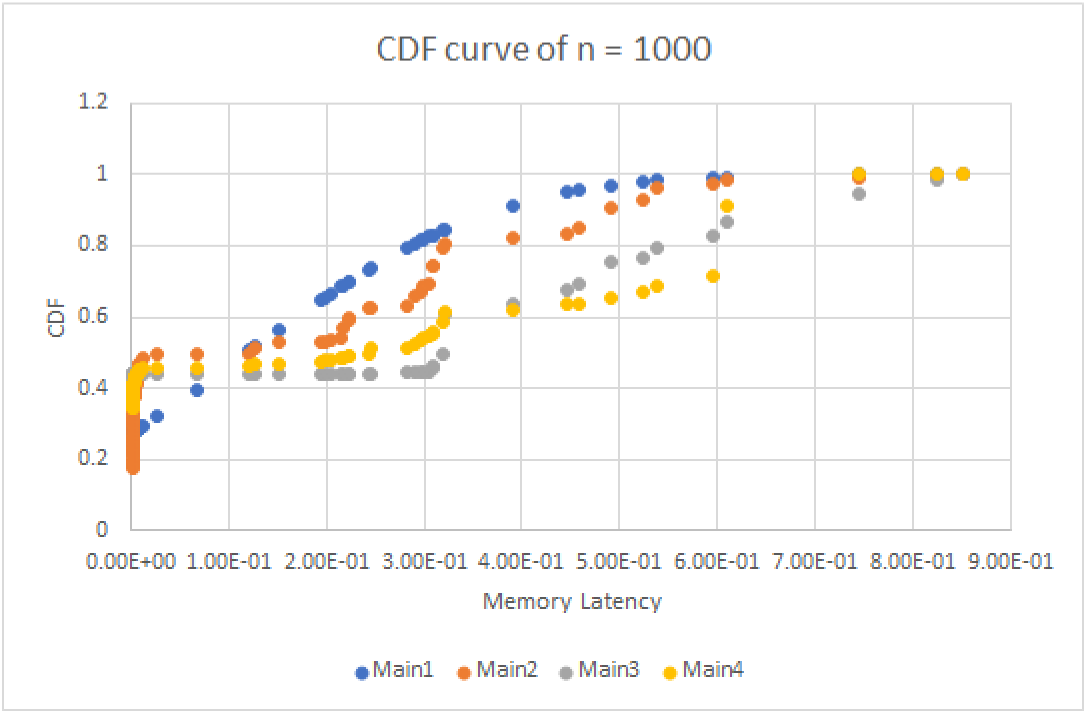
Appendix:

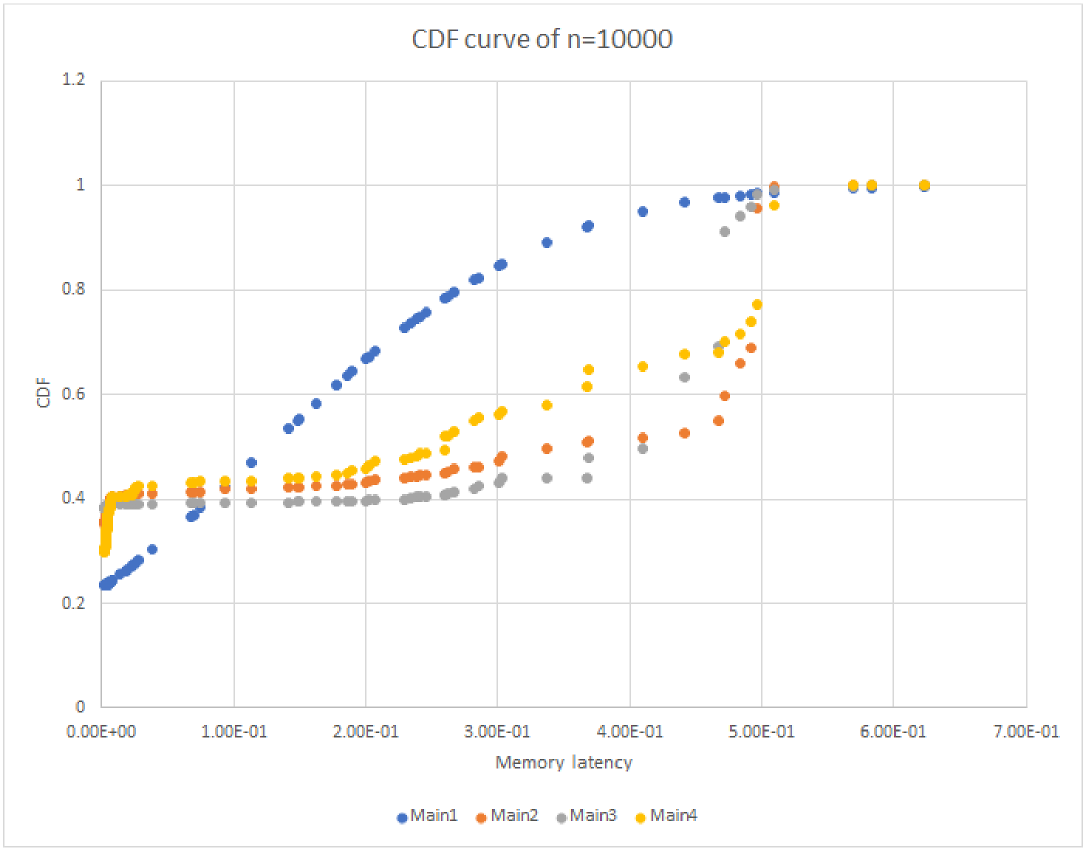
Table 1:

Average memory access latencies of 1000 request				
	n= 10	n=100	n=1000	n=10000
Main1	7.05 e -2	1.25 e -1	1.19 e -1	1.26 e -1
Main2	6.08 e -3	2.67 e -3	2.60 e -3	4.67 e -3
Main3	3.31 e -2	3.48 e -2	3.39 e -2	2.89 e -2
Main4	2.86 e -3	3.15 e -3	4.66 e -3	5.01 e -3






CDF curve of n = 1000
CDF
Memory Latency
1.2
1
0.8
0.6
0.4
0.2
0
0.00E+00
1.00E-01
2.00E-01
3.00E-01
4.00E-01
5.00E-01
6.00E-01
7.00E-01
8.00E-01
9.00E-01
Main1
Main2
Main3
Main4


CDF curve of n=10000
CDF
Memory latency
1.2
1
0.8
0.6
0.4
0.2
0
0.00E+00
1.00E-01
2.00E-01
3.00E-01
4.00E-01
5.00E-01
6.00E-01
7.00E-01
Main1
Main2
Main3
Main4