Casey Henderson (29038802)

Concurrent and Parallel Systems – Discussion Forum Assignment Report

I have conducted various experiments to investigate the throughput of my server, the results of which are attached to the end of this report. To test my server, I created a testing harness that would create *m* poster threads and *n* reader threads, posting messages continuously for 10 seconds (or any given time) and outputting how many requests were completed in the given time. My strategy when generating POST and READ requests in my test harness was to generate random 1-character long strings from a pre-defined array to maximise performance. I stored 7 different values in an array and selected one of them randomly each time a request was generated. This value was then added request as the relevant component (topicId, message etc.). Keeping the values short and pre-defined was intended to maximise performance and deliver the best test data, while also randomly selecting them each time to ensure requests were varied. I used a counter to track the number of requests being sent, and the QueryPerformanceCounter to track the time taken. Users supply the time they wish requests to be sent for, along with the IP address and number of poster and reader threads, in command-line arguments. By using QueryPerformanceCounter, I could accurately measure the time elapsed, ensuring my experiments were robust and that requests were being sent for exactly the time specified. QueryPerformanceCounter returns a high-resolution timestamp, making it a superior choice to alternatives that don't (such as the timer() function).

Analysis of the data from my experiments shows some interesting trends. Firstly, the fastest combination was my server and my test harness (on average 4.45% faster than the reference implementation), followed by my server and the reference test harness (1.77% faster than the reference implementation). The reference implementation was faster at generating READ requests on several occasions, however, suggesting a strength in fast random access compared to my own implementation, and was particularly strong when there was only one thread of poster and reader each, beating my implementation both on READ requests and overall requests (as can be seen on the respective graphs and tables). This suggests that my implementation performs better with more threads, while the reference implementation is better suited to smaller numbers of threads.

The fastest combination at generating requests per thread per second was (3, 3), suggesting this is the optimum number of threads for my implementation. More READ requests were sent on average than POST requests (30,508.68 to 29,986.13), suggesting a slight speed advantage for READ requests. Unsurprisingly, the high throughput tests for (5, 5) saw the most requests generated per second (78160 POST and 79242.57 READ), although it still fell short to (3, 3) for the most requests generated per thread per second (15740.26 to 18139.28), reinforcing the idea that (3, 3) is the optimum number of threads for request generation with my implementation. The trend observed in both POST and READ requests per second is that the overall number of requests generated increases as the number of threads increases, reaching its peak at (5, 5): however, this is not the case for the number of requests generated per thread per second, suggesting maximum thread efficiency is reached at (3,3).

One way I would have changed the design of my server to improve performance is by employing thread affinity, where threads that share data are placed next to each other to gain the advantages of inter-thread locality. This is where remnants of the threads that were run on a given core may remain in its state (i.e., still in its cache memory) when another thread that's also in need of that data takes place. If we schedule that specific thread to run on the same core, we can improve performance by reducing the occurrence of events such as cache misses (which negatively impact on application performance). To do this, the programmer must provide scheduling guidance for the application (as the OS only handles general scheduling) – then, threads can share copies of cache lines, reportedly delivering up to 3x speedup (Le, 2020). This is complicated to implement in C++, however, and time pressures meant I couldn't investigate implementing it this time.

Another way I would've changed the design of my server to improve performance is by implementing methods to reduce false sharing. This is a problem that happens because different variables accessed by different threads are mapped to the same cache line in main memory, which can cause cache misses. A potential solution to this would've been to map different variables to different cache lines by padding each variable with extra bytes, meaning that the variable size would fit the whole cache line where it's mapped to. To do this, I would have created a struct – however, when investigating this I realised how complex it would be to implement, and decided to prioritise other methods of performance improvement, such as choosing to implement a map data structure for its faster read/write speeds. If I had chosen to implement anti-false-sharing, methods, however, I could potentially have avoided 'cache ping-pong' and other issues (CodeGuru Staff, 2020), leading to performance improvements in my application. This could've been particularly useful in improving my application speed in certain READ request scenarios where its performance falls below that of the reference implementation (as can be seen on the average READ graph attached), as it would've improved my read speeds and provided me with faster, more reliable access to my data structure.

Bibliography

CodeGuru Staff. (2020, August 11). *C++ Programming: False Sharing in Multi-threaded Programming, and a Glance at the C++0x std::thread* . Retrieved from CodeGuru: C++ Programming: False Sharing in Multi-threaded Programming, and a Glance at the C++0x std::thread

Le, D. (2020, Aug 13). *Optimizations for C++ multi-threaded programming*. Retrieved from medium.com: https://medium.com/distributed-knowledge/optimizations-for-c-multi-threaded-programs-33284dee5e9c

Testing Tables and Graphs

Headline Figures

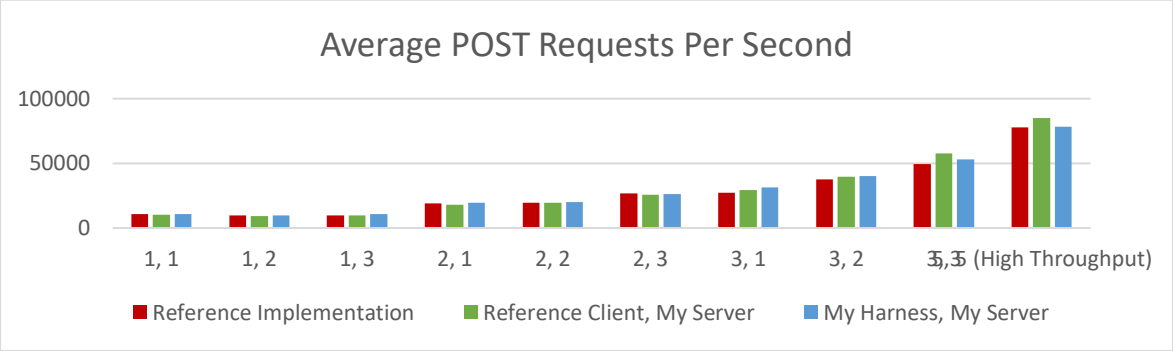
Overall Average Speed Difference Between Reference Implementation and Mine: Mine is faster on average **(+4.45%)**
Overall Average Speed Difference Between Reference Implementation and Mine (Both Using Reference Client): Mine is faster on average **(+1.77%)**

Average POST Requests Per Second

All values to 2dp

Number of poster and reader threads (m, n)	With Reference Client and Reference Server (average)	With Reference Client and My Server (average)	With My Test Harness and My Server (average)	Difference from Reference Implementation (raw)	Difference from Reference Implementation (percentage)	Difference when both using Reference Client (percentage)
1, 1	**10766.90**	**10,285.80**	**10,874.91**	**+108.01**	**+1.00 %**	**-4.46%**
1, 2	**9490.20**	**9041.80**	**9772.85**	**+282.65**	**+2.98%**	**-4.72%**
1, 3	**9567.10**	**9,502.60**	**10,635.94**	**+1,068.84**	**+11.17%**	**-0.67%**
2, 1	**19078.70**	**18150.20**	**19485.19**	**+406.49**	**+2.13%**	**-4.87%**
2, 2	**19391.60**	**19517.60**	**20168.64**	**+777.04**	**+4.01%**	**+0.65%**
2, 3	**26584.90**	**25851.20**	**26178.51**	**-406.39**	**-1.53%**	**-2.76%**
3, 1	**27512.60**	**29136.60**	**31449.17**	**+3,936.57**	**+14.31%**	**+5.90%**
3, 2	**37556.40**	**39478.60**	**40045.91**	**+2,489.51**	**+6.63%**	**+5.12%**
3, 3	**49562.40**	**57944.70**	**53090.12**	**+3,527.72**	**+7.12%**	**+16.91%**
5, 5 (High Throughput)	**77662.60**	**85246.10**	**78160.03**	**+497.43**	**+0.64%**	**+9.76%**

Average Percentage Difference between Reference Implementation and Mine: Mine is faster on average (+**4.85%)**
Average Percentage Difference between Reference Server and Mine (Both Using Reference Client): Mine is faster on average **(+2.09%)**
Average number of POST requests sent per second (My Harness, My Server): 29,986.13



Test Harness Runs

Number of poster and reader threads (m, n)	Run 1	Run 2	Run 3	Run 4	Run 5	Run 6	Run 7	Run 8	Run 9	Run 10
1, 1	**10900.70**	**10911.50**	**10889.70**	**11129.40**	**10913.30**	**10484.30**	**10722.70**	**10913.30**	**10833.40**	**11051.10**
1, 2	**9691.80**	**9576.90**	**10130.00**	**9477.90**	**9573.90**	**9914.90**	**10068.10**	**9612.50**	**9891.10**	**9791.40**
1, 3	**10363.80**	**10615.00**	**10756.40**	**10708.80**	**10833.40**	**10429.60**	**11120.40**	**10573.30**	**10222.50**	**10736.20**
2, 1	**19421.10**	**19579.00**	**19747.50**	**19446.10**	**20583.50**	**18620.30**	**18865.50**	**19316.80**	**19431.90**	**19840.20**
2, 2	**21424.00**	**18776.40**	**20253.60**	**18784.70**	**19749.90**	**19515.70**	**21416.90**	**20481.30**	**20393.90**	**20890.00**
2, 3	**27013.70**	**24636.30**	**22331.90**	**27374.20**	**25894.10**	**25065.30**	**31609.10**	**26472.40**	**21961.30**	**29426.80**
3, 1	**32563.80**	**31373.90**	**32886.10**	**32330.70**	**32709.20**	**32769.80**	**27803.20**	**30062.20**	**31933.00**	**30059.80**
3, 2	**43081.10**	**48044.40**	**38814.80**	**39259.50**	**35939.70**	**40329.00**	**40249.50**	**38383.70**	**37894.20**	**38463.20**
3, 3	**55326.20**	**53341.20**	**41473.80**	**59015.20**	**66709.30**	**59914.60**	**42348.30**	**41302.90**	**50475.00**	**60994.70**
5, 5 (High Throughput)	**86901.90**	**81866.20**	**74631.50**	**80070.30**	**74415.70**	**75762.40**	**77929.30**	**76381.20**	**75159.60**	**78482.20**

Average READ Requests per second

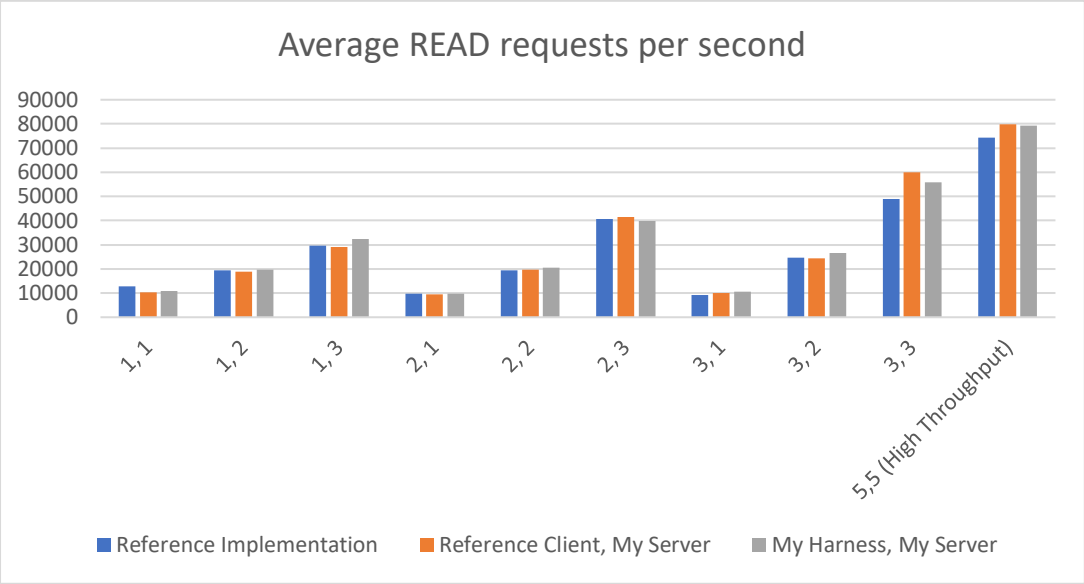
All values to 2dp

Number of poster and reader threads (m, n)	With Reference Client and Reference Server (average)	With Reference Client and My Server (average)	With My Test Harness and My Server (average)	Difference from Reference Implementation (raw)	Difference from Reference Implementation (percentage)	Difference when both using Reference Client (percentage)
1, 1	**12802.80**	**10411.20**	**10977.10**	**-1,825.70**	**-14.26%**	**-18.68%**
1, 2	**19537.40**	**18752.10**	**19713.14**	**+175.74**	**+0.90%**	**-4.02%**
1, 3	**29705.90**	**28982.40**	**32269.13**	**+2,563.23**	**+8.63%**	**-2.44%**
2, 1	**9736.30**	**9594.80**	**9748.19**	**+11.89**	**+0.12%**	**-1.45%**
2, 2	**19426.50**	**19817.30**	**20492.05**	**+1,065.55**	**+5.49%**	**+2.01%**
2, 3	**40573.50**	**41347.20**	**39888.90**	**-684.60**	**-1.69%**	**+1.91%**
3, 1	**9144.00**	**10057.90**	**10491.94**	**+1,347.94**	**+14.74%**	**+9.99%**
3, 2	**24651.80**	**24278.00**	**26518.33**	**+1,866.53**	**+7.57%**	**-1.54%**
3, 3	**49039.90**	**60057.00**	**55745.53**	**+6,705.63**	**+13.67**	**+22.47%**
5, 5 (High Throughput)	**74441.10**	**79831.90**	**79242.57**	**+4,801.47**	**+6.45**	**+7.24%**

Average Percentage Difference between Reference Implementation and Mine: Mine is faster on average (+**4.16%**)

Average Percentage Difference between Reference Server and Mine (Both Using Reference Client): Mine is faster on average (+**1.55%**)

Average Number of READ Requests sent per second (My Harness, My Server): 30,508.68



Test Harness Runs

Number of poster and reader threads (m, n)	Run 1	Run 2	Run 3	Run 4	Run 5	Run 6	Run 7	Run 8	Run 9	Run 10
1, 1	**10972.30**	**11084.80**	**11065.00**	**11229.80**	**11120.70**	**10549.20**	**10786.20**	**11029.90**	**10946.30**	**10985.90**
1, 2	**20008.20**	**19424.10**	**20702.30**	**19631.80**	**19445.50**	**19745.40**	**19328.20**	**18981.10**	**20048.40**	**19816.40**
1, 3	**31526.00**	**32804.60**	**32223.30**	**32825.40**	**32227.10**	**32164.30**	**32866.70**	**31947.80**	**31298.50**	**32807.90**
2, 1	**9558.60**	**9756.60**	**9703.30**	**9847.60**	**10340.90**	**9437.20**	**9636.30**	**9536.30**	**9653.00**	**10012.10**
2, 2	**21842.20**	**18035.50**	**20342.30**	**18744.40**	**20835.40**	**19500.40**	**21860.90**	**21387.00**	**21426.00**	**20946.40**
2, 3	**48062.80**	**36920.10**	**33488.80**	**38974.80**	**39729.60**	**41032.80**	**42201.20**	**40267.70**	**35289.10**	**42922.10**
3, 1	**10825.80**	**10826.50**	**11043.20**	**10236.20**	**10833.30**	**11172.80**	**9569.90**	**9596.20**	**10708.20**	**10107.30**
3, 2	**26611.80**	**34177.60**	**24091.70**	**25672.60**	**23058.30**	**23973.30**	**26591.20**	**27505.70**	**25808.50**	**27692.60**
3, 3	**54391.90**	**70467.30**	**41114.10**	**68987.30**	**64358.50**	**56629.10**	**433272**	**45053.00**	**50321.80**	**628051**
5, 5 (High Throughput)	**80905.90**	**81937.90**	**77910.50**	**78367.50**	**79954.70**	**83067.50**	**77230.00**	**73686.60**	**77419.50**	**83532.80**

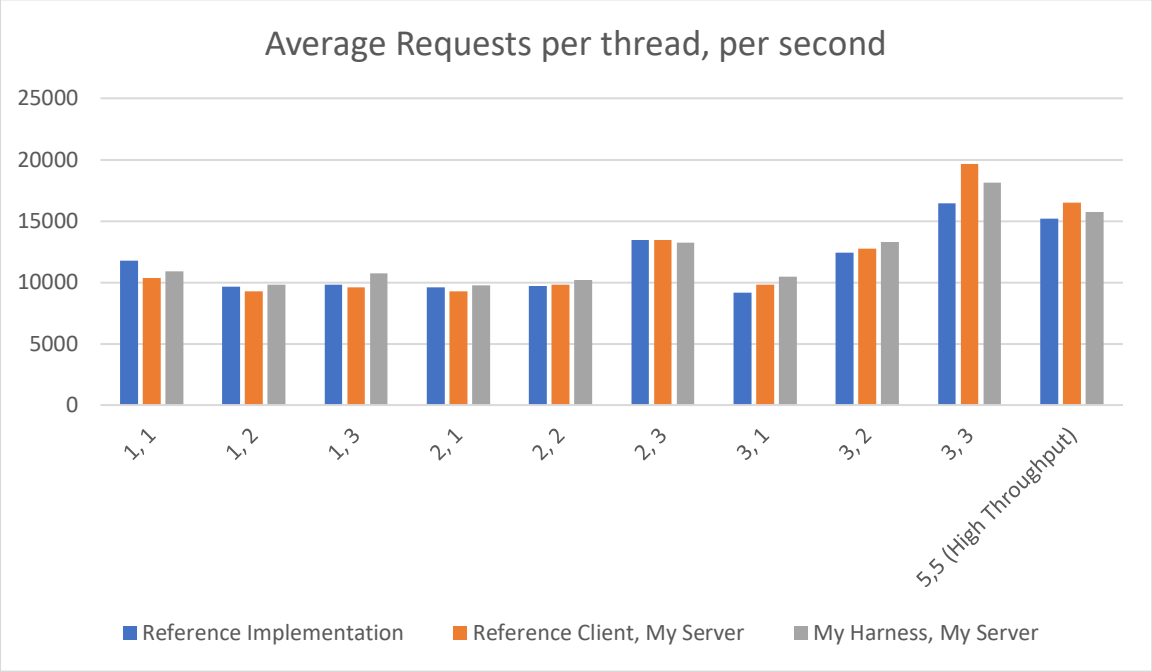
Average Requests per thread, per second

All values to 2dp

Number of poster and reader threads (m, n)	With Reference Client and Reference Server (average)	With Reference Client and My Server (average)	With My Test Harness and My Server (average)	Difference from Reference Implementation (raw)	Difference from Reference Implementation (percentage)	Difference when both using Reference Client (percentage)
1, 1	**11784.80**	**10348.40**	**10925.96**	**-858.84**	**-7.29%**	**-12.19%**
1, 2	**9675.81**	**9264.55**	**9828.66**	**+152.85**	**+1.58%**	**-4.25%**
1, 3	**9818.21**	**9621.21**	**10726.27**	**+908.06**	**+9.25%**	**-2.05%**
2, 1	**9604.96**	**9248.28**	**9744.46**	**+139.50**	**+1.45%**	**-3.71%**
2, 2	**9704.46**	**9833.67**	**10165.17**	**+460.71**	**+4.75%**	**+1.31%**
2, 3	**13431.60**	**13439.60**	**13213.48**	**-218.12**	**-1.62%**	**+0.06%**
3, 1	**9164.12**	**9798.57**	**10485.28**	**+1,321.16**	**+14.42%**	**+6.92%**
3, 2	**12441.60**	**12751.30**	**13312.85**	**+871.25**	**+7.00%**	**+2.49%**
3, 3	**16433.70**	**19666.90**	**18139.28**	**+1,705.58**	**+10.38%**	**+19.67%**
5, 5 (High Throughput)	**15210.30**	**16507.70**	**15740.26**	**+529.96**	**+3.48%**	**+8.53%**

Average Percentage Difference Between Reference Implementation and Mine: Mine is faster on average (+4.34%)

Average Percentage Difference Between Reference Server and Mine (Both Using Reference Client): Mine is faster on average (+1.68%)



Test Harness Runs

Number of poster and reader threads (m, n)	Run 1	Run 2	Run 3	Run 4	Run 5	Run 6	Run 7	Run 8	Run 9	Run 10
1, 1	**10936.50**	**10998.10**	**10977.30**	**11179.60**	**11017.00**	**10516.70**	**10754.40**	**10971.40**	**10889.80**	**11018.50**
1, 2	**9900.00**	**9667.00**	**10277.40**	**9703.20**	**9673.10**	**9886.70**	**9798.70**	**9531.20**	**9979.80**	**9869.20**
1, 3	**10472.40**	**10854.90**	**10744.80**	**10883.50**	**10765.10**	**10648.40**	**10996.70**	**10630.20**	**10380.20**	**10886.00**
2, 1	**9659.90**	**9778.50**	**9816.90**	**9764.50**	**10308.10**	**9352.50**	**9500.60**	**9617.70**	**9694.90**	**9950.70**
2, 2	**10816.50**	**9202.90**	**10148.90**	**9382.20**	**10146.30**	**9754.00**	**10819.40**	**10467.00**	**10454.90**	**10459.10**
2, 3	**15015.30**	**12311.20**	**11164.10**	**13269.80**	**13124.70**	**13219.60**	**14762.00**	**13348.00**	**11450.00**	**14469.70**
3, 1	**10847.40**	**10550.10**	**10982.30**	**10641.70**	**10885.60**	**10985.60**	**9343.20**	**9914.60**	**10660.30**	**10041.70**
3, 2	**13938.50**	**16444.40**	**12581.30**	**12986.40**	**11799.60**	**12860.40**	**13368.10**	**13177.80**	**12740.50**	**13231.10**
3, 3	**18286.30**	**20634.70**	**13764.60**	**21333.70**	**21844.60**	**19423.90**	**14279.20**	**14392.60**	**16799.40**	**20633.30**
5, 5 (High Throughput)	**16780.70**	**16380.40**	**15254.20**	**15843.70**	**15437.00**	**15882.90**	**15515.90**	**15006.70**	**15257.90**	**16201.50**

NOTE: Due to a recurring illness, I had to conduct my testing at home on my MacBook Pro, as by the time I'd recovered university had closed for the holidays and I wasn't able to access Cantor 9341. I checked with Chris before returning home and have listed my laptop's specs here for comparison. Apologies for any inconvenience.

Testing PC Specs:
MacBook Pro (15-inch, 2018)
2.2 GHz 6-Core Intel Core i7
16 GB 2400 MHz DDR4 RAM
251 GB Flash Storage