Concurrent Distributed Dynamic Sized Queue

Yunjia Wang (yunjiaw@andrew.cmu.edu)
Jiaan Dai (jiaand@andrew.cmu.edu)

Dec 9, 2019

Project URL: `https://flowersh0026.github.io/618-final/`

1 Summary

2 Background

3 Approach

4 Results

4.1 Performance Metric and Experimental Setup

We use Google Benchmark library [2] to do the performance measurement. Since the accurate wall-clock time measurement in Google Benchmark requires disabling the CPU frequency scaling, we use the average CPU time for a single thread as an estimate of the wall-clock time. The metric we use for performance comparison is the push and pop operation throughput, which is defined as the number of items being pushed or popped in a second.

We set up the experiments using a Multiple Producer Multiple Consumer (MPMC) framework. In each experiment, we create equal number of producer and consumer threads to repeatedly do the push and pop operations. The number of producer threads is from 1 to 16, and the number of consumer threads is also from 1 to 16. The number of iterations is determined by the Google benchmark library (i.e., the size of inputs, typically varying from 100,000 to 7,000,000). When the benchmark is running, the producer and consumer threads continuously push and pop items to and from the shared concurrent queue. They counts the number of pushed items and popped items separately. On tear-down, it reports the sum of the number of items pushed and items popped, and we will use these two numbers to calculate the push throughput and pop throughput in this

benchmark case. A typical number of pushed and popped items varies from 50,000 to 3,000,000 in our results.

The problem size of our benchmark is the number of concurrent threads, rather than the number of items being pushed and popped. This is because the number of concurrent threads reflects the level of contentions, which is an important factor when benchmarking the performance of a concurrent data structure. Thus, in all results shown below, we use the number of concurrent threads as the x axis. Since the machines we are targeting (the GHC cluster machines, ghcX.ghc.andrew.cmu.edu) have 16-core CPUs, we will run at most 32 threads (considering Intel's hyper-threading) in each benchmark case, which means that we run at most 16 producer threads and 16 consumer threads at the same time.

4.2 Effects of Optimization Techniques

We first compare the effects of two different optimization techniques in our implemented concurrent queues, including using a better memory allocator (jemalloc [3]) and conducing cache line alignment on variables to avoid false sharing. We mainly focus on the results of RTM-based queue and CAS-based queue here. In the figures, "None (baseline)" version is the baseline version, which uses none of these two optimization techniques. "jemalloc" version is the version using the jemalloc memory allocator. "Aligned" version is the version that aligns all internal variables of the concurrent queues to the cache line. Finally, the "jemalloc & Aligned" version is the version uses both jemalloc and cache line alignment.

4.2.1 Push Throughput of RtmQueue

First, we shows the comparison of push throughput on RTM-based queue when applying different combinations of optimizations.
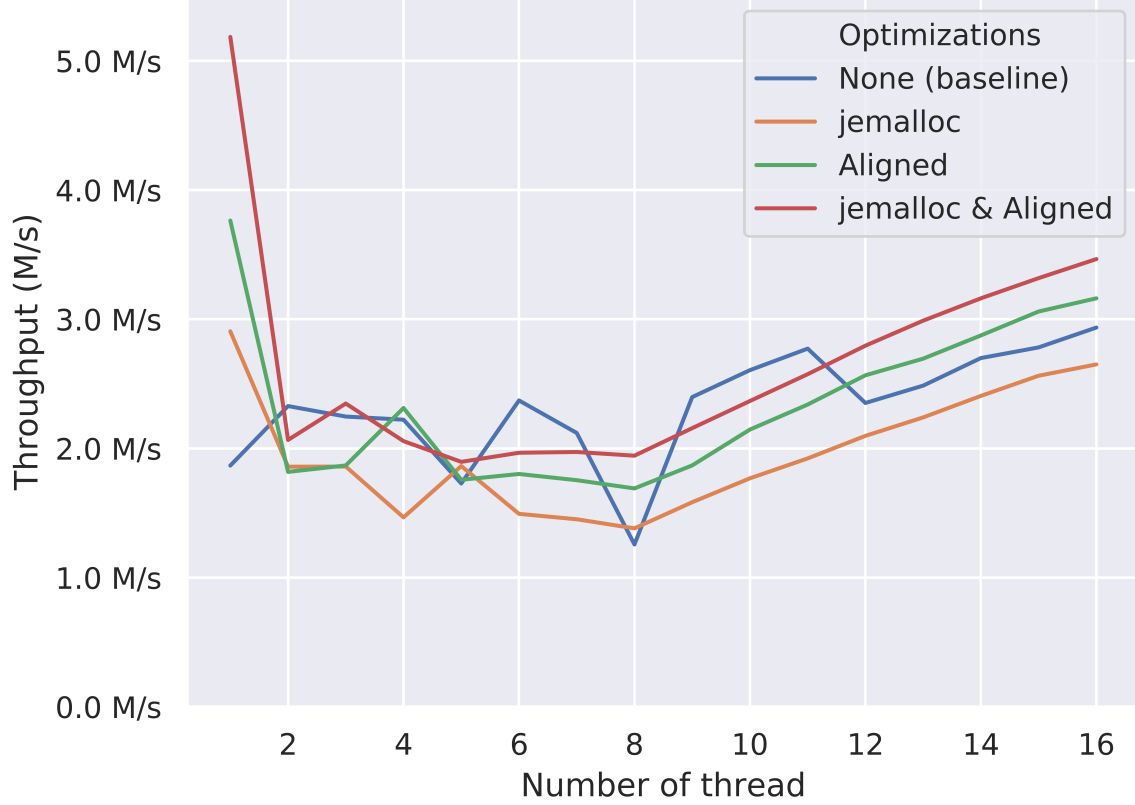
Figure 1: Effects of optimizations (push throughput of RTM-based queue).

We first look at the low-contention region (thread = 1). From the figure, we know that using jemalloc is able to significantly improve the performance of both non-aligned and aligned version. In this scenario, the contention only comes from the consumer when checking an empty queue, if variables are aligned to the cache line, since both producer and consumer will access the same dummy node when the queue is empty. Also, because the producer roughly does nothing between two push operations, the memory allocation may takes non-negligible portions of time. Therefore, using a better memory allocator is able to improve the performance in this scenario.

Cache line alignment brings higher improvement in this low-contention region. From this figure, the "Aligned" version is higher than the "jemalloc" version, showing that this optimization technique has a higher speedup compared with using a better memory allocator. By aligning the variables within the data structure to the cache line (basically the head and tail pointer), it greatly reduces the false sharing between the producer and consumer threads, so significantly improves the performance.

In the high-contention region (thread = 16), the effects are roughly the same, except that the "jemalloc" version performs worse than the baseline non-optimized version. We believe that this is due to the performance variance when false sharing exists. The consistent improvement of the "jemalloc & Aligned" version compared with the "Aligned" version in the region from thread = 5 to thread 16 partially illustrates our explanation.

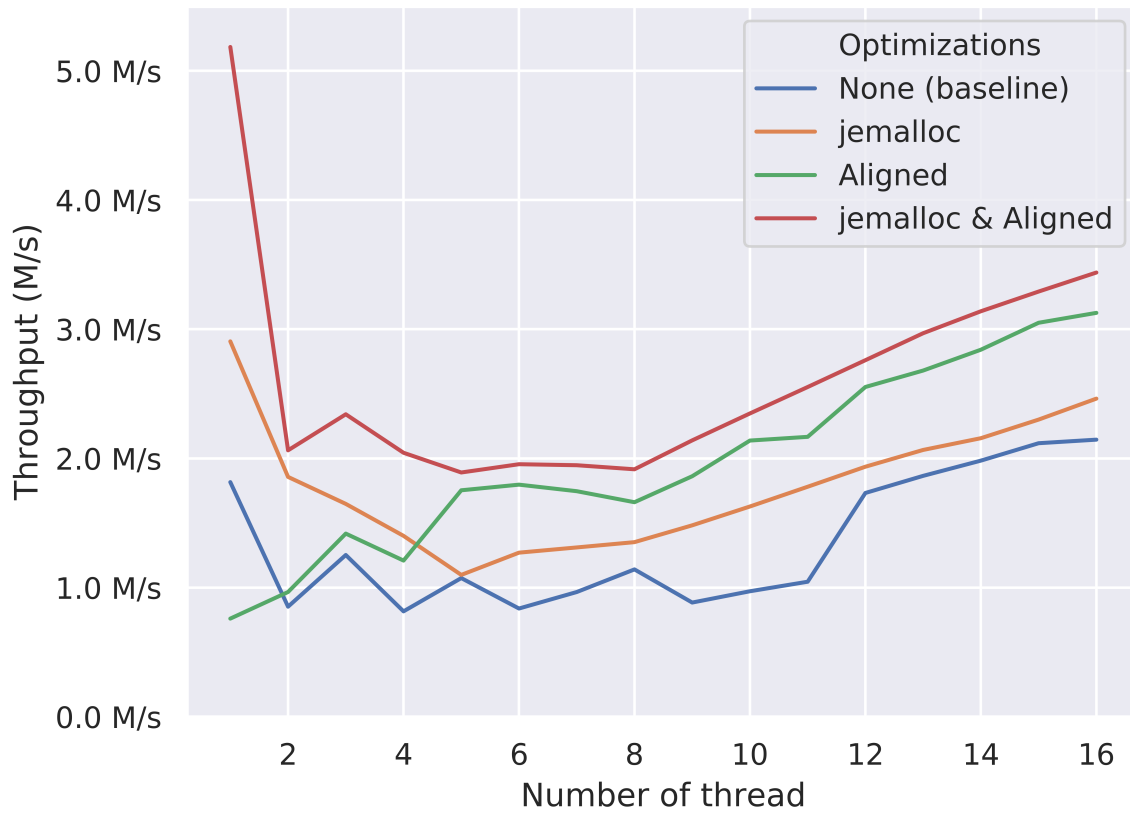### 4.2.2 Pop Throughput of RtmQueue

Figure 2: Effects of optimizations (pop throughput of RTM-based queue).

The effects of optimizations on the pop throughput of RTM-based queue are similar. The "jemalloc & Aligned" version uniformly outperforms other poorly optimized versions. However, from the figure, the "Aligned" version performs worse than the baseline version when thread = 1, which is possibly due to the high variance nature of microbenchmark.

4.2.3 Push Throughput of CasQueue

Figure 3: Effects of optimizations (push throughput of CAS-based queue).

Similarly, the "jemalloc & Aligned" version of the CAS-based queue generally outperforms other versions across the whole x axis. However, the performance improvement is not stable and it does not perform better when thread = 1. We believe that this is due to the fact that the CAS version uses more memory access than others in its lock-free algorithm, which possibly creates more chances of memory contention and cache coherence traffics, resulting in a higher variance in performance. Also, the performance gain of these two optimizations in CAS-based queue is less than the one we obtain in the RTM-based queue.
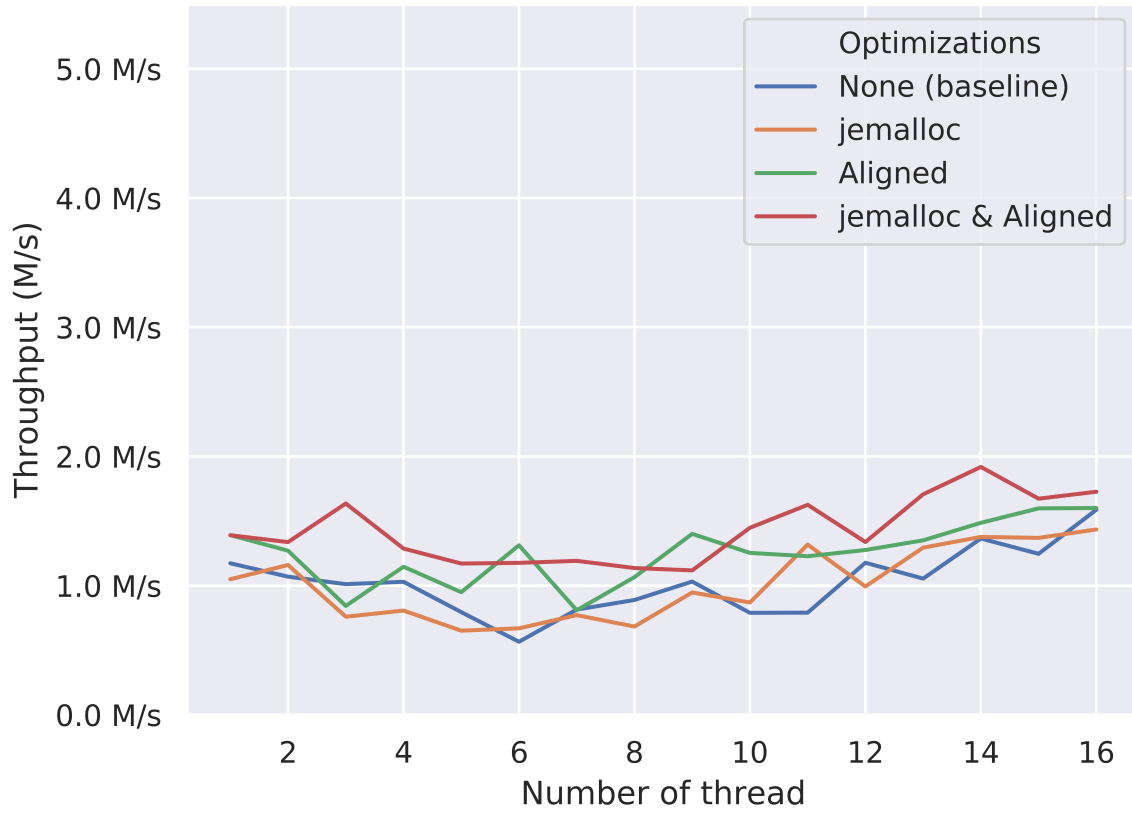
4.2.4   Pop Throughput of CasQueue

Figure 4: Effects of optimizations (pop throughput of CAS-based queue).

We can observe an overall better performance of the "jemalloc & Aligned" version. However, similar to the push throughput result of CAS-based queue, the performance gain is not quite significant. Though the pop operation in the CAS-based queue only requires one CAS operation when fetching the head node, it also needs to fix the tail pointer position during this operation, and runs a while loop inside the lock-free memory deallocation. In contrast, the push operation merely access the tail pointer. In addition, we only count the succeeded pop, so the failed pop on the empty queue also lowers the overall pop throughput of the CAS-based queue.

4.2.5   Summary on Optimizations

After benchmarking the effects, we are convinced that applying these optimization techniques does improve the performance of concurrent queues. Hence, we apply them in all our implemented concurrent queues, and the results shown below all use the "jemalloc & Aligned" version.

4.3 Performance comparison

After exploring the best optimizations, we compare the performance metric between all our implemented concurrent queues, including two locked versions (coarse-grained and fined-grained versions). We separately compare the push throughput and pop throughput.

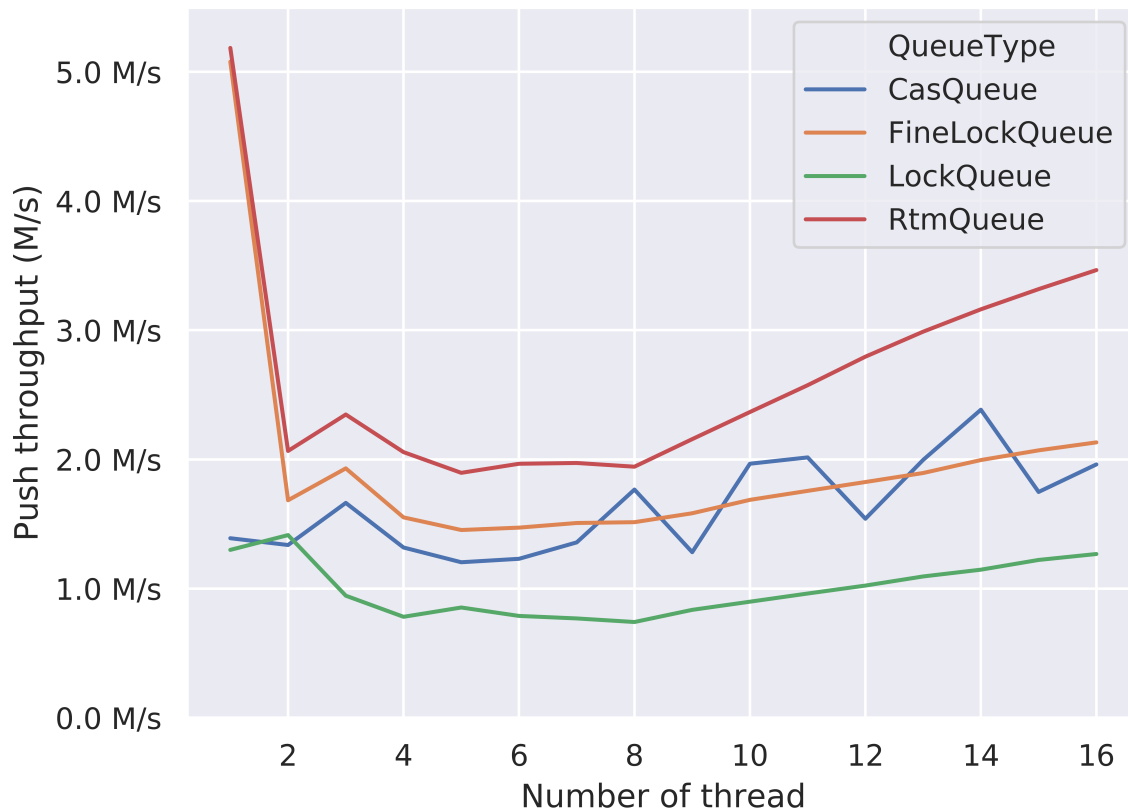### 4.3.1 Push Throughput Comparison

Figure 5: Performance comparison on push throughput.

In the low-contention region (thread = 1), RTM-based queue and fine-grained lock queue greatly outperforms other two versions. Because the fine-grained versions separate the mutex for head and tail pointers (also properly aligned), the producer thread is less likely to be affected by the consumer thread. In contrast, the coarse-grained queue takes the global mutex for both head and tail pointers, which creates contention between the producer and the consumer.

The RTM-based queue is basically evolved from the fine-grained lock queue, by adding an optimistic path using transactional memory. Therefore, under the low-contention scenario, it uses the optimistic path in most cases. From here we can also see that the mutex performance is pretty well when the contention is low. For the CAS-based queue, its lock-free algorithm has a higher constant factor, which probably negatively affects its performance in low-contention region.

In the high-contention region (thread > 8), RTM-based queue significantly outperforms all other versions, which means that the optimistic path using the transactional memory has a high success rate, because the fallback alternative path of RTM-based queue requires two extra memory-synchronized atomic operations, which leads to a higher constant factor compared with the fine-grained lock version. The performance of CAS-based queue is not stable, which may be the result of high contention. The fine-grained lock queue shows a similar performance compared with CAS-based queue. Finally, the coarse-grained lock version undoubtedly shows the worst throughput performance.

An interesting fact from this figure is that the absolute throughput for RTM-based queue and fine-grained lock queue in thread = 1 greatly outperforms the throughput in thread = 16. However, from thread = 2 to thread = 16, we indeed observe the performance improvement as the number of threads increases. We guess that the low-contention scenario greatly reduces the cache coherence traffics, leading to a faster memory access and finally the overall better throughput. From thread = 1 to 2, the cache coherence traffics increase and the memory access is no longer cheap. So, we observe a rapid drop from thread = 1 to 2. This result also help explain why people use two different designs for MPMC and Single Producer Single Consumer (SPSC) scenarios, for example, boost [4] and folly [5].

4.3.2 Pop Throughput Comparison

Figure 6: Performance comparison on pop throughput.

The performance comparison on pop throughput is quite similar to the result of push throughput. When thread = 1, both RTM-based queue and fine-grained lock queue significantly outperforms other two implementations. As the number of threads increases from thread = 2, the pop throughput increases. And finally, when thread = 16, the RTM-based queue greatly outperforms other queues including the fine-grained lock version. Therefore, from these two figures, we conclude that the optimistic path in the RTM-based queue greatly improve the overall performance. However, the machine dependency of RTM (requires Intel TSX) probably limits its applications, especially in the era of cloud computing.

4.3.3   Analysis and Deeper Analysis

One of the factor that limits the speedup of our concurrent queues is the cache coherence overhead (synchronization). Thus, we did the cache line alignment which reduces the false sharing and improves the overall performance. The great improvement of cache line alignment demonstrates our hypothesis that the synchronization is one of the limiting factor. Another factor that limits the speedup of our concurrent queues is the memory allocation. Through the experiment that compares the performance with and without jemalloc shown above, we observe performance improvement while using a better memory allocator. This shows that our hypothesis about the memory allocation is also correct. Currently we merely use the jemalloc for memory allocation. We believe a well-design fixed-size memory pool with thread-local cache can outperform jemalloc (designed for general variable-length allocation) in our implementation. However, due to the limited time, we don't include it in our project.

Execution time breakdown is difficult when handling the code that is sensitive to the cache coherence traffics. The average time for each operation varies from 100 ns to 1000 ns, which means that extra time measurement code inside the operation likely changes the numbers a lot. In addition, the succeed rate of the optimistic path in the transactional memory may change after inserting extra code for the time measurement. Hence, we don't try to breakdown the push and pop operations. However, from the analysis above, we think that we can try implementing a fixed-size memory pool to further improve the performance.

4.4   Distributed Extension

5   References

[1] Maged M. Michael and Michael L. Scott. 1996. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing (PODC '96). ACM, New York, NY, USA, 267-275.

[2] Google Benchmark library. https://github.com/google/benchmark/

[3] jemalloc. http://jemalloc.net/

[4] Boost C++ Libraries. https://www.boost.org/

[5] Folly: Facebook Open-source Library. https://github.com/facebook/folly/