A Fast General Purpose Lock-Free Queue for C++

Posted November 06, 2014

So I've been bitten by the lock-free bug! After finishing my single-producer, single-consumer lock-free queue, I decided to design and implement a more general multi-producer, multi-consumer queue and see how it stacked up against existing lock-free C++ queues. After over a year of spare-time development and testing, it's finally time for a public release. TL;DR: You can grab the C++11 implementation from GitHub (or jump to the benchmarks).

The way the queue works is interesting, I think, so that's what this blog post is about. A much more detailed and complete (but also more dry) description is available in a sister blog post, by the way.

Sharing data: Oh, the woes

At first glance, a general purpose lock-free queue seems fairly easy to implement. It isn't. The root of the problem is that the same variables necessarily need to be shared with several threads. For example, take a common linked-list based approach: At a minimum, the head and tail of the list need to be shared, because consumers all need to be able to read and update the head, and the producers all need to be able to update the tail.

This doesn't sound too bad so far, but the real problems arise when a thread needs to update more than one variable to keep the queue in a consistent state -- atomicity is only ensured for single variables, and atomicity for compound variables (structs) is almost certainly going to result in a sort of lock (on most platforms, depending on the size of the variable). For example, what if a consumer read the last item from the queue and updated only the head? The tail should not still point to it, because the object will soon be freed! But the consumer could be interrupted by the OS and suspended for a few milliseconds before it updates the tail, and during that time the tail could be updated by another thread, and then it becomes too late for the first thread to set it to null.

The solutions to this fundamental problem of shared data are the crux of lock-free programming. Often the best way is to conceive of an algorithm that doesn't need to update multiple variables to maintain consistency in the first place, or one where incremental updates still leave the data structure in a consistent state. Various tricks can be used, such as never freeing memory once allocated (this helps with reads from threads that aren't up to date), storing extra state in the last two bits of a pointer (this works with 4-byte aligned pointers), and reference counting pointers. But tricks like these only go so far; the real effort goes into developing the algorithms themselves.

My queue

The less threads fight over the same data, the better. So, instead of using a single data structure that linearizes all operations, a set of sub-queues is used instead -- one for each producer thread. This means that different threads can enqueue items completely in parallel, independently of each other.

Of course, this also makes dequeueing slightly more complicated: Now we have to check every sub-queue for items when dequeuing. Interestingly, it turns out that the order that elements are pulled from the sub-queues really doesn't matter. All elements from a given producer thread will necessarily still be seen in that same order relative to each other when dequeued (since the sub-queue preserves that order), albeit with elements from other sub-queues possibly interleaved. Interleaving elements is OK because even in a traditional single-queue model, the order that elements get put in from from different producer threads is non-deterministic anyway (because there's a race condition between the different producers). [Edit: This is only true if the producers are independent, which isn't necessarily the case. See the comments.] The only downside to this approach is that if the queue is empty, every single sub-queue has to be checked in order to determine this (also, by the time one sub-queue is checked, a previously empty one could have become non-empty -- but in practice this doesn't cause problems). However, in the non-empty case, there is much less contention overall because sub-queues can be "paired up" with consumers. This reduces data sharing to the near-optimal level (where every consumer is matched with exactly one producer), without losing the ability to handle the general case. This pairing is done using a heuristic that takes into account the last sub-queue a producer successfully pulled from (essentially, it gives consumers an affinity). Of course, in order to do this pairing, some state has to be maintained between calls to dequeue -- this is done using consumer-specific "tokens" that the user is in charge of allocating. Note that tokens are completely optional -- the queue merely reverts to searching every sub-queue for an element without one, which is correct, just slightly slower when many threads are involved.

So, that's the high-level design. What about the core algorithm used within each sub-queue? Well, instead of being based on a linked-list of nodes (which implies constantly allocating and freeing or re-using elements, and typically relies on a compare-and-swap loop which can be slow under heavy contention), I based my queue on an array model. Instead of linking individual elements, I have a "block" of several elements. The logical head and tail indices of the queue are represented using atomically-incremented integers. Between these logical indices and the blocks lies a scheme for mapping each index to its block and sub-index within that block. An enqueue operation simply increments the tail (remember that there's only one producer thread for each sub-queue). A dequeue operation increments the head if it sees that the head is less than the tail, and then it checks to see if it accidentally incremented the head past the tail (this can happen under contention -- there's multiple consumer threads per sub-queue). If it did over-increment the head, a correction counter is incremented (making the queue eventually consistent), and if not, it goes ahead and increments another integer which gives it the actual final logical index. The increment of this final index always yields a valid index in the actual queue, regardless of what other threads are doing or have done; this works because the final index is only ever incremented when there's guaranteed to be at least one element to dequeue (which was checked when the first index was incremented).

So there you have it. An enqueue operation is done with a single atomic increment, and a dequeue is done with two atomic increments in the fast-path, and one extra otherwise. (Of course, this is discounting all the block allocation/re-use/referencing counting/block mapping goop, which, while important, is not very interesting -- in any case, most of those costs are amortized over an entire block's worth of elements.) The really interesting part of this design is that it allows extremely efficient bulk operations -- in terms of atomic instructions (which tend to be a bottleneck), enqueueing X items in a block has exactly the same amount of overhead as enqueueing a single item (ditto for dequeueing), provided they're in the same block. That's where the real performance gains come in :-)

I heard there was code

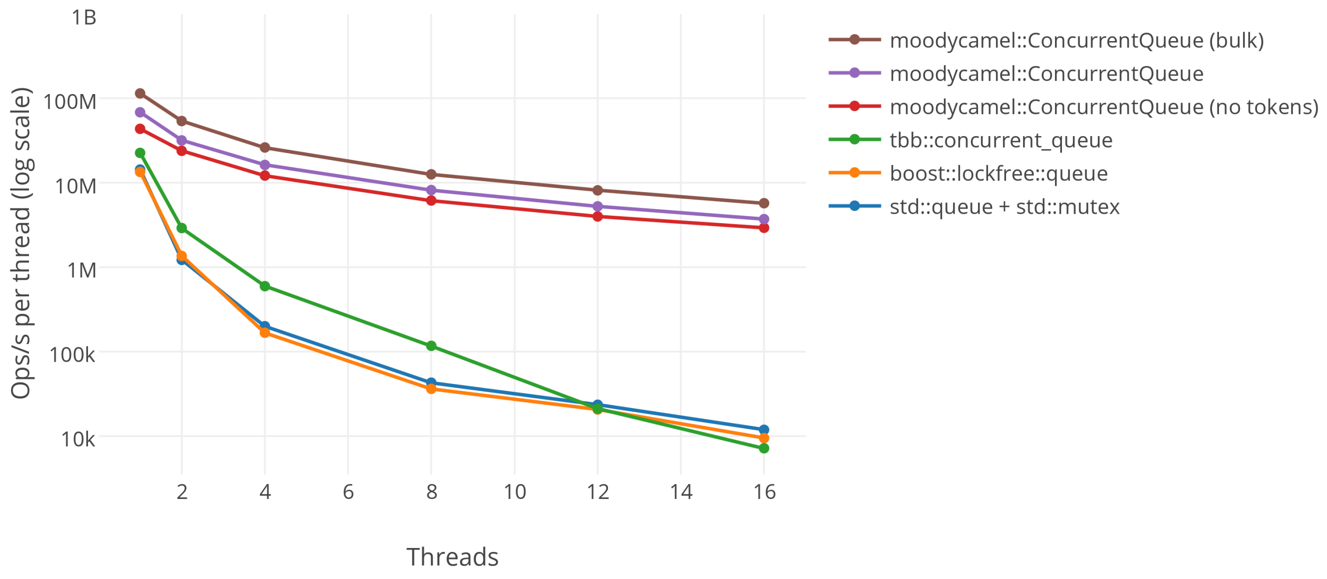
Since I thought there was rather a lack of high-quality lock-free queues for C++, I wrote one using this design I came up with. (While there are others, notably the ones in Boost and Intel's TBB, mine has more features, such as having no restrictions on the element type, and is faster to boot.) You can find it over at GitHub. It's all contained in a single header, and available under the simplified BSD license. Just drop it in your project and enjoy!

Benchmarks, yay!

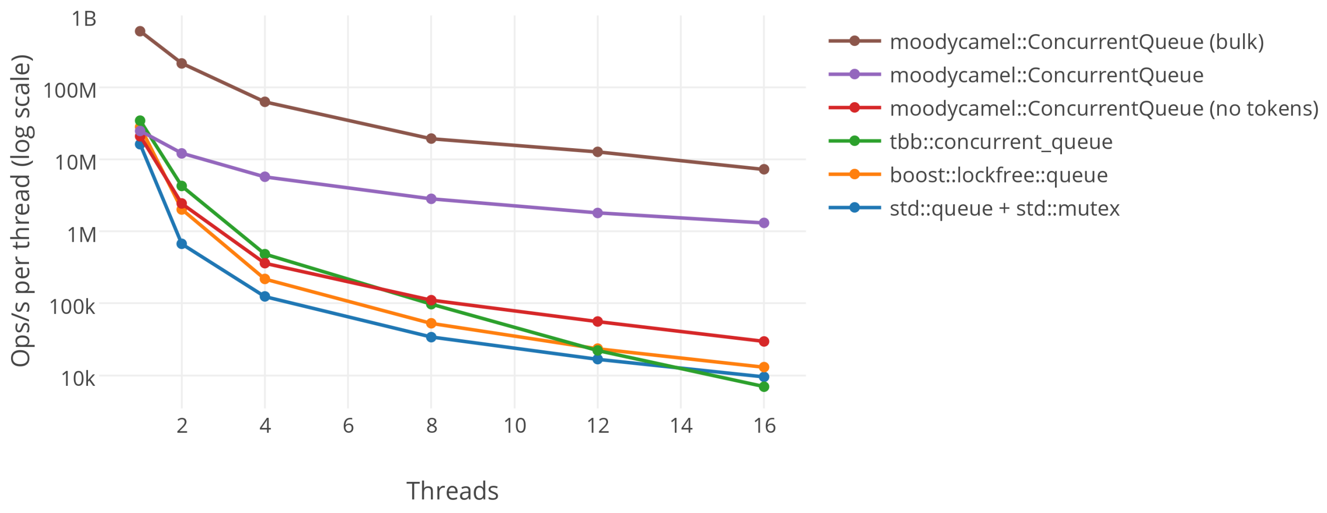
So, the fun part of creating data structures is writing synthetic benchmarks and seeing how fast yours is versus other existing ones. For comparison, I used the Boost 1.55 lock-free queue, Intel's TBB 4.3 `concurrent_queue`, another linked-list based lock-free queue of my own (a naïve design for reference), a lock-based queue using `std::mutex`, and a normal `std::queue` (for reference against a regular data structure that's accessed purely from one thread). Note that the graphs below only show a subset of the results, and omit both the naïve lock-free and single-threaded `std::queue` implementations.

Here are the results! Detailed raw data follows the pretty graphs (note that I had to use a **logarithmic scale** due to the enormous differences in absolute throughput).

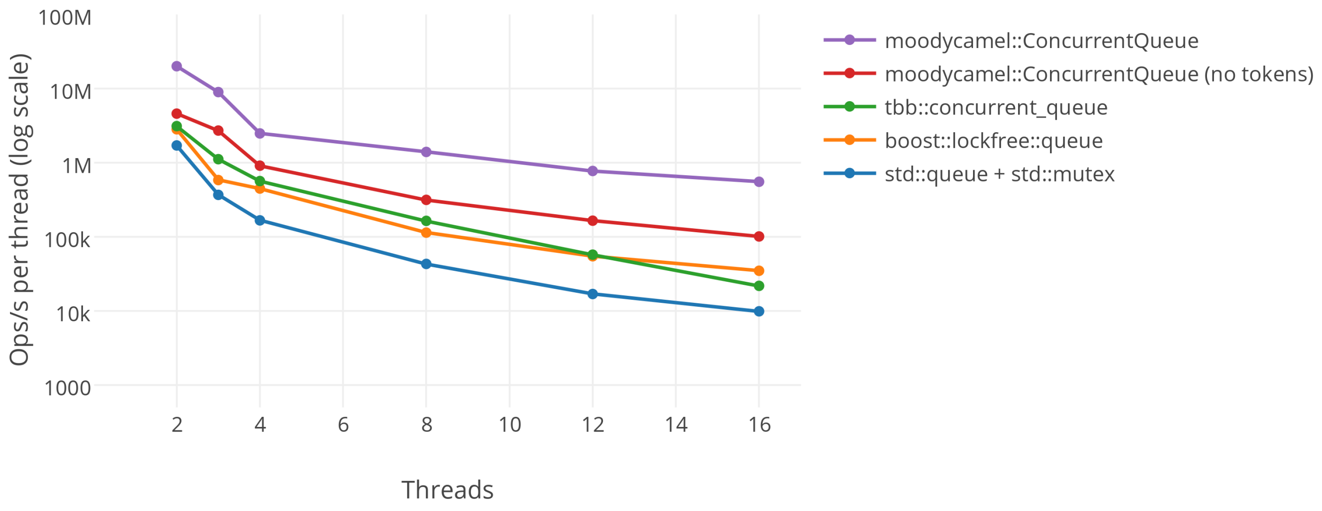
Enqueue Performance (AWS 8-core)



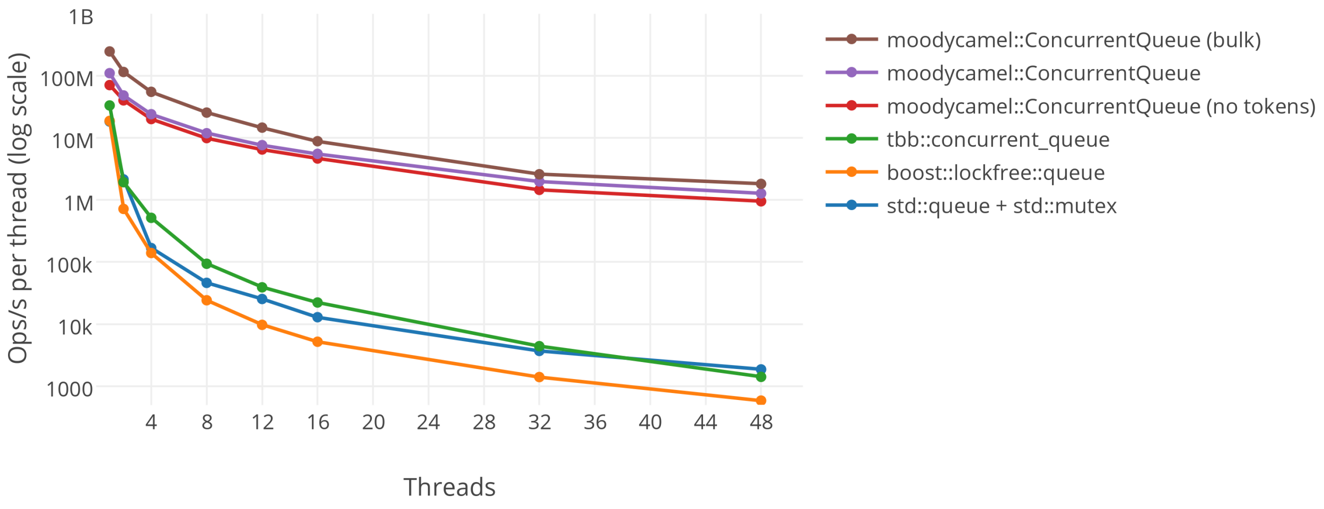
Dequeue Performance (AWS 8-core)

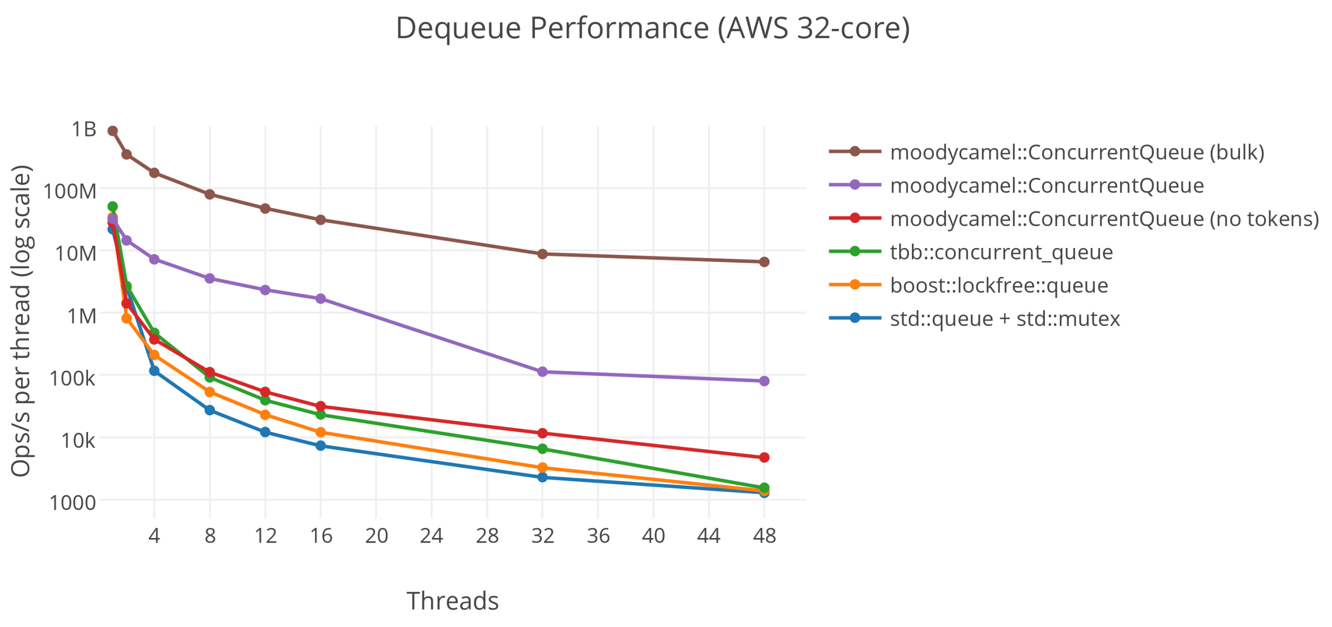


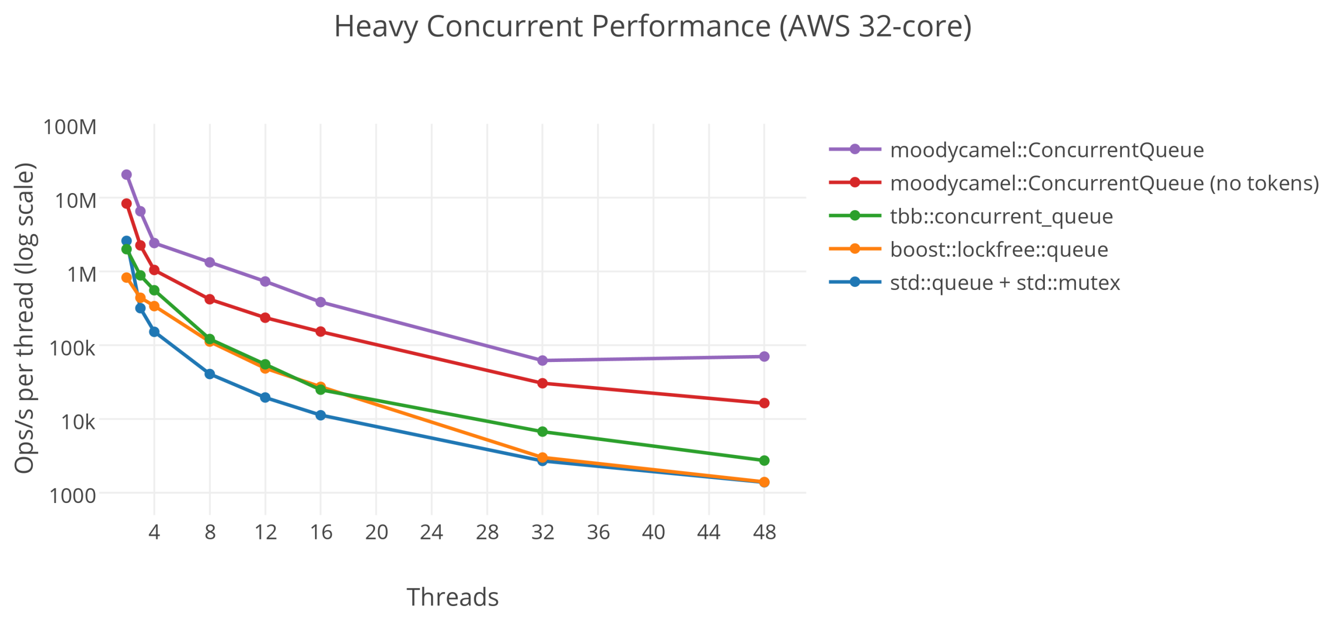
Heavy Concurrent Performance (AWS 8-core)



Enqueue Performance (AWS 32-core)







64-bit 8-core AWS instance (c1.xlarge), Ubuntu, g++

```
Running 64-bit benchmarks on an Intel(R) Xeon(R) CPU           E5506  @ 2.13GHz
    (precise mode)
Note that these are synthetic benchmarks. Take them with a grain of salt.

Legend:
    'Avg':     Average time taken per operation, normalized to be per thread
    'Range':   The minimum and maximum times taken per operation (per thread)
    'Ops/s':   Overall operations per second
    'Ops/s/t': Operations per second per thread (inverse of 'Avg')
    Operations include those that fail (e.g. because the queue is empty).
    Each logical enqueue/dequeue counts as an individual operation when in bulk.

balanced:
  (Measures the average operation speed with multiple symmetrical threads
  under reasonable load -- small random intervals between accesses)
  > moodycamel::ConcurrentQueue
      Without tokens
         2   threads:  Avg: 2.1439us  Range: [2.0316us, 2.1834us]  Ops/s: 932.90k  Ops/s/t
         3   threads:  Avg: 3.8096us  Range: [3.3842us, 4.1643us]  Ops/s: 787.49k  Ops/s/t
         4   threads:  Avg: 5.5406us  Range: [5.0081us, 5.6616us]  Ops/s: 721.94k  Ops/s/t
         8   threads:  Avg: 0.0120ms  Range: [0.0118ms, 0.0123ms]  Ops/s: 664.05k  Ops/s/t
         12  threads:  Avg: 0.0189ms  Range: [0.0188ms, 0.0191ms]  Ops/s: 633.72k  Ops/s/t
         16  threads:  Avg: 0.0246ms  Range: [0.0240ms, 0.0248ms]  Ops/s: 650.46k  Ops/s/t
         Operations per second per thread (weighted average): 133.15k

      With tokens
         2   threads:  Avg: 2.0623us  Range: [2.0364us, 2.0712us]  Ops/s: 969.77k  Ops/s/t
         3   threads:  Avg: 3.1724us  Range: [3.1542us, 3.2327us]  Ops/s: 945.67k  Ops/s/t
         4   threads:  Avg: 4.4476us  Range: [4.2467us, 4.6332us]  Ops/s: 899.36k  Ops/s/t
```

64-bit 32-core AWS instance (cc2.8xlarge), Ubuntu, g++

```
Running 64-bit benchmarks on a       Intel(R) Xeon(R) CPU E5-2670 0 @ 2.60GHz
    (precise mode)
Note that these are synthetic benchmarks. Take them with a grain of salt.

Legend:
    'Avg':     Average time taken per operation, normalized to be per thread
    'Range':   The minimum and maximum times taken per operation (per thread)
    'Ops/s':   Overall operations per second
    'Ops/s/t': Operations per second per thread (inverse of 'Avg')
    Operations include those that fail (e.g. because the queue is empty).
    Each logical enqueue/dequeue counts as an individual operation when in bulk.

balanced:
  (Measures the average operation speed with multiple symmetrical threads
  under reasonable load -- small random intervals between accesses)
  > moodycamel::ConcurrentQueue
      Without tokens
         2   threads:  Avg: 0.9490us  Range: [0.9444us, 0.9512us]  Ops/s:   2.11M  Ops/s/t
         3   threads:  Avg: 1.6217us  Range: [1.6120us, 1.6288us]  Ops/s:   1.85M  Ops/s/t
         4   threads:  Avg: 2.5471us  Range: [2.5150us, 2.5578us]  Ops/s:   1.57M  Ops/s/t
         8   threads:  Avg: 8.7667us  Range: [8.7260us, 8.8109us]  Ops/s: 912.54k  Ops/s/t
         12  threads:  Avg: 0.0194ms  Range: [0.0186ms, 0.0196ms]  Ops/s: 618.06k  Ops/s/t
         16  threads:  Avg: 0.0347ms  Range: [0.0343ms, 0.0348ms]  Ops/s: 461.73k  Ops/s/t
         32  threads:  Avg: 0.1103ms  Range: [0.1095ms, 0.1110ms]  Ops/s: 290.14k  Ops/s/t
         Operations per second per thread (weighted average): 190.15k

      With tokens
         2   threads:  Avg: 0.6626us  Range: [0.6322us, 0.6736us]  Ops/s:   3.02M  Ops/s/t
         3   threads:  Avg: 1.1325us  Range: [1.1000us, 1.1511us]  Ops/s:   2.65M  Ops/s/t
```

64-bit 8-core, Fedora 19, g++ 4.81 (a Linode VM)

```
Running 64-bit benchmarks on a      Intel(R) Xeon(R) CPU E5-2650L 0 @ 1.80GHz
    (precise mode)
Note that these are synthetic benchmarks. Take them with a grain of salt.

Legend:
    'Avg':     Average time taken per operation, normalized to be per thread
    'Range':   The minimum and maximum times taken per operation (per thread)
    'Ops/s':   Overall operations per second
    'Ops/s/t': Operations per second per thread (inverse of 'Avg')
    Operations include those that fail (e.g. because the queue is empty).
    Each logical enqueue/dequeue counts as an individual operation when in bulk.

balanced:
  (Measures the average operation speed with multiple symmetrical threads
  under reasonable load -- small random intervals between accesses)
  > moodycamel::ConcurrentQueue
      Without tokens
         2   threads:  Avg: 3.9753us  Range: [3.9540us, 3.9879us]  Ops/s: 503.11k  Ops/s/t
         3   threads:  Avg: 6.0741us  Range: [6.0383us, 6.1039us]  Ops/s: 493.90k  Ops/s/t
         4   threads:  Avg: 8.3842us  Range: [8.3230us, 8.4119us]  Ops/s: 477.09k  Ops/s/t
         8   threads:  Avg: 0.0199ms  Range: [0.0196ms, 0.0200ms]  Ops/s: 402.49k  Ops/s/t
         12  threads:  Avg: 0.0290ms  Range: [0.0288ms, 0.0292ms]  Ops/s: 414.03k  Ops/s/t
         16  threads:  Avg: 0.0403ms  Range: [0.0394ms, 0.0408ms]  Ops/s: 397.36k  Ops/s/t
         Operations per second per thread (weighted average):  80.36k

      With tokens
         2   threads:  Avg: 3.8409us  Range: [3.8075us, 3.8688us]  Ops/s: 520.71k  Ops/s/t
         3   threads:  Avg: 5.8761us  Range: [5.8086us, 5.9162us]  Ops/s: 510.55k  Ops/s/t
         4   threads:  Avg: 7.9556us  Range: [7.8602us, 7.9969us]  Ops/s: 502.79k  Ops/s/t
```

64-bit dual-core, Windows 7, MinGW-w64 g++ 4.81 (a netbook)

```
      3   threads:  Avg: 1.0917us  Range: [0.9830us, 1.1773us]  Ops/s:   2.75M  Ops/s/t: 91
      4   threads:  Avg: 1.1793us  Range: [1.0690us, 1.2686us]  Ops/s:   3.39M  Ops/s/t: 84
      Operations per second per thread (weighted average):   1.10M

    > tbb::concurrent_queue
      2   threads:  Avg: 0.5967us  Range: [0.4621us, 0.6802us]  Ops/s:   3.35M  Ops/s/t:
      3   threads:  Avg: 0.9434us  Range: [0.3221us, 1.1513us]  Ops/s:   3.18M  Ops/s/t:
      4   threads:  Avg: 1.3151us  Range: [0.4924us, 1.7355us]  Ops/s:   3.04M  Ops/s/t: 76
      Operations per second per thread (weighted average):   1.11M

    > SimpleLockFreeQueue
      2   threads:  Avg: 0.7336us  Range: [0.6230us, 0.8236us]  Ops/s:   2.73M  Ops/s/t:
      3   threads:  Avg: 0.6925us  Range: [0.4716us, 0.8721us]  Ops/s:   4.33M  Ops/s/t:
      4   threads:  Avg: 0.6623us  Range: [0.5031us, 0.8796us]  Ops/s:   6.04M  Ops/s/t:
      Operations per second per thread (weighted average):   1.45M

    > LockBasedQueue
      2   threads:  Avg: 0.0210ms  Range: [0.0127ms, 0.0273ms]  Ops/s:  95.11k  Ops/s/t:  4
      3   threads:  Avg: 0.0370ms  Range: [9.9256us, 0.0798ms]  Ops/s:  81.04k  Ops/s/t:  2
      4   threads:  Avg: 0.0970ms  Range: [0.0147ms, 0.1225ms]  Ops/s:  41.22k  Ops/s/t:  1
      Operations per second per thread (weighted average):  26.17k

    > std::queue
      (skipping, benchmark not supported...)

 Overall average operations per second per thread (where higher-concurrency runs have more
 (Take this summary with a grain of salt -- look at the individual benchmark results for a
 better idea of how the queues measure up to each other):
    moodycamel::ConcurrentQueue (including bulk):  16.69M
```

As you can see, my queue is generally at least as good as the next fastest implementation, and often much faster (especially the bulk operations, which are in a league of their own!). The only benchmark that shows a significant reduction in throughput with respect to the other queues is the 'dequeue from empty' one, which is the worst case dequeue scenario for my implementation because it has to check *all* the inner queues; it's still faster on average than a successful dequeue operation, though. The impact of this can also be seen in certain other benchmarks, such as the single-producer multi-consumer one, where most of the dequeue operations fail (because the producer can't keep up with demand), and the relative throughput of my queue compared to the others suffers slightly as a result. Also note that the LockBasedQueue is extremely slow on my Windows netbook; I think this is a quality-of-implementation issue with the MinGW `std::mutex`, which does not use Windows's efficient `CRITICAL_SECTION` and seems to suffer terribly as a result.

I think the most important thing that can be gleaned from these benchmarks is that the total system throughput *at best* stays roughly constant as the number of threads increases (and often falls off anyway). This means that even with the fastest queue of each benchmark, the amount of work each thread accomplishes individually declines with each thread added, with approximately the same total amount of work being accomplished by e.g. four threads and sixteen. And that's the best case. There is no linear scaling at this level of contention and throughput; the moral of the story is to stay away from designs that require intensively sharing data if you care about performance. Of course, real applications tend not to be purely moving things around in queues, and thus have far lower contention and *do* have the possibility of scaling upwards (at least a little) as threads are added.

On the whole, I'm very happy with the results. Enjoy the queue!

21 Comments **Login**

Join the discussion…

LOG IN WITH

OR SIGN UP WITH DISQUS

Name

Sort by Newest

Evgeny • a year ago
Hi, could you elaborate on the following problem. Since producer and consumer are paired this queue design may not adress the high rate by one of producers. Its consumer may fail to cope and there will be no other to help. In theory writing data (what producer does) is slower than read (what consumer does) so it may be balanced provided that consumer does not do anything else, but this is a massive limitation for the consumer. Usually consumer implements some business logic, otherwise it would need to pass the data farther down the line and encounter the same issue.
Reply • Share ›

Evgeny Evgeny • a year ago
Ah, actually I read more and found that the queue is not a set of spsc queues, but spmc ones. That solves the problem.
Reply • Share ›

Jinkun Geng • a year ago
How much is the performance difference between the blocking version and the unblocking version of the concurrent queue?
Hi, Cameron. Since you provide both the blocking version and unblocking version of concurrent queue, I am wondering how much the performance gap will be between them. (I expect the unblocking version will be higher-performance, after wall, it just busy polls the queue, whereas the blocking version is event-driven by some signals)
Reply • Share ›

Cameron Mod Jinkun Geng • a year ago
Hi, it depends how it's used. Running the benchmarks on my laptop shows the non-blocking version is somewhere between 30%-80% faster for most operations (except checking if the queue is empty, where the blocking version is much faster). See https://gist.github.com/cam.... I suggest building and running the benchmarks on your own system, or (even better!) testing performance directly in the context of your particular application.
Reply • Share ›

VaMa SMS • 2 years ago • edited

Hi,
I have requirement for a single producer and multiple consumer . where in a producer generates new data and the same is not deleted , but overwritten by same producer in a circular buffer fashion.

consumer can read the produced data as many times as possible without destroying it.

can i use this library for this purpose?

^ | ˅ • Reply • Share ›

Cameron Mod ➦ VaMa SMS • 2 years ago • edited

Hi, no, this queue implementation is not based on a circular buffer and does not allow unread data to be overwritten. Additionally, data can only be read/popped once by a given consumer.

^ | ˅ • Reply • Share ›

bhargav M.P • 2 years ago • edited

Is this queue(concurrentqueue) tested on x86 processor on debian machine.? When i tested with a sample program(that deques the element from the queue when the program allocates the memory for an object i,e "new" and enqueues the element when the program de-allocates the memory i.e delete) it shows better results when compared with boost lock free queue but when used with the production software the cpu % is more (2% to 3%) when compared to boost::lockfree queue. The production software also use the concurrent queue for memory allocation an de-allocation purpose (like the sample program) but it shows performance degradation when compared with boost::lockfree queue. Is it got something to do with the size of the object that production software is using.? Or i am missing something here. ?
Thanks.

^ | ˅ • Reply • Share ›

Cameron Mod ➦ bhargav M.P • 2 years ago

Tested on x86, sure, Debian, no, but that shouldn't matter.

Your code could be faster with the boost queue for any number of reasons. It's possible that your use case better fits the boost implementation.

^ | ˅ • Reply • Share ›

bhargav M.P ➦ Cameron • 2 years ago • edited

Ok. Do you see any obvious reasons that why it can be faster with boost.? We use jemalloc for memory allocation does it have any effect on the queue.? Thanks for your help

^ | ˅ • Reply • Share ›

Cameron Mod ➦ bhargav M.P • 2 years ago

No. You'll need to investigate yourself with the specifics of your use case.

^ | ˅ • Reply • Share ›

bhargav M.P ➦ Cameron • 2 years ago

Ok. Thank you.

^ | ˅ • Reply • Share ›

NA_2020 • 2 years ago

I am trying to replace and existing WorkQueue as your lock free and at first look it seems that both lock and lock free giving same CPU usage. My doubt is : Is this because of my Queue structure (class of native types such as int, char will consume less CPU usage , for user defined types (for instance size >70 bytes) will take more cpu time to deque or process)

^ | ⌄ • Reply • Share ›

Cameron Mod ➦ NA_2020 • 2 years ago

CPU usage is not a very good measure of throughput.

^ | ⌄ • Reply • Share ›

NA_2020 ➦ Cameron • 2 years ago

Thank you, overall i am seeing 8% improvement in BW, Is this Max i can expect.

^ | ⌄ • Reply • Share ›

Cameron Mod ➦ NA_2020 • 2 years ago

It depends. Have you tried using the explicit consumer/producer tokens? Bulk methods? Is your element type POD or does a constructor/method have to be called on every move/assign? Are they marked nothrow?

Also, as you mentioned, a structure of 70+ bytes is going to be slower to move than a structure of a couple dwords. Keep in mind that each element has to move between different CPU caches in the worst case. A cacheline is typically only 64 bytes, so you'd have two cache misses per element instead of one cache miss for multiple elements (they are stored mostly contiguously).

^ | ⌄ • Reply • Share ›

NA_2020 ➦ Cameron • 2 years ago

Thanks for your reply. Yet to try on explicit consumer/producer tokens. Bulk is not applicable for my case as it is MPSC (n producer , 1 consumer thread) then again processed data given to other queue as SPMC (8 threads). It is NOT POD, Yes constructor/method invloved and it is not marked noexcept/notthrow , Is it have any impact on performance. Completely agreed with you on cache misses.

Do you have any suggestion.

^ | ⌄ • Reply • Share ›

Cameron Mod ➦ NA_2020 • 2 years ago

For best performance, definitely use producer/consumer tokens. Marking your move ctor and assignment operators as nothrow (if possible) results in faster/smaller code since all the exception handling code can be dropped.

^ | ⌄ • Reply • Share ›

NA_2020 ➦ Cameron • 2 years ago

Thanks Cameron for the advice. Tried producer/consumer token observed little improvement in B/W but doesn't meet target. Trying figure out am i missing anything.

^ | ⌄ • Reply • Share ›

Richard Whitehead • 3 years ago • edited

I'm just evaluating this code, which looks awesome!

Please tell me: Is it possible to limit the size of the queue, so that pushing will fail rather than allocating

more space? I'm intending this for a microcontroller with very limited memory.
Sorry, I don't really understand the stuff about Traits, perhaps you could point me at an example that uses it?
Thanks so much for contributing this incredibly useful code.

^ | ˅ • Reply • Share ›

Cameron Mod ➦ Richard Whitehead • 3 years ago • edited

You can use try_enqueue instead of enqueue, and it won't allocate (but you'll need to ue the traits to set up good defaults, and pass parameters to the constructor).

I would not suggest using this on a microcontroller though. It's far too memory hungry (it often trades memory for speed, plus the code size is huge).

^ | ˅ • Reply • Share ›

Richard Whitehead ➦ Cameron • 3 years ago

Thanks Cameron. I tried using the queue on Windows and it worked great (and the code was not too large at all). However, my real target is a microcontroller programmed using IAR compiler, and although it's C++14, it has no std::thread. We use an RTOS but they have not wrapped that into <thread>. So unfortunately, I can't use your lovely library (or anyone else's that I can find) on my target.

^ | ˅ • Reply • Share ›