Detailed Design of a Lock-Free Queue

Posted November 06, 2014

This post outlines, in quite some detail, my design for an efficient lock-free queue that supports multiple concurrent producers and consumers (an MPMC queue). My C++11 implementation of this design can be found on GitHub. A more high-level overview of the queue, including benchmark results, can be found in my initial blog post introducing my implementation.

One key insight is that the total order that elements come out of a queue is irrelevant, as long as the order that they went in on a given thread matches the order they come out on another thread. This means that the queue can be safely implemented as a set of independent queues, one for each producer; writing a single-producer, multi-consumer lock free queue is much easier than writing a multi-producer, multi-consumer one, and can be implemented much more efficiently too. The SPMC queue can be generalized to an MPMC one by having the consumers pull from different SPMC queues as needed (and this can be done efficiently as well with some cleverness). A heuristic is used to speed up dequeueing in the typical case, pairing consumers with producers as much as possible (which drastically reduces the overall contention in the system).

Apart from the high-level set-of-SPMC-queues design, the other key part of the queue is the core queueing algorithm itself, which is brand new (I devised it myself) and unlike any other that I've heard of. It uses an atomic counter to track how many elements are available, and once one or more have been claimed (by incrementing a corresponding elements-consumed counter and checking whether that increment turned out to be valid), an atomic

index can be safely incremented to get the actual IDs of the elements to reference. The problem is then reduced to mapping integer IDs to individual elements, without having to worry about other threads referencing the same objects (each ID is given to only one thread). Details are below!

System Overview

The queue is composed of a series of single-producer, multi-consumer (SPMC) queues. There is one SPMC queue per producer; the consumers use a heuristic to determine which of these queues to consume from next. The queue is lock-free (though not quite wait-free). It is designed to be robust, insanely fast (especially on x86), and allow bulk enqueueing and dequeueing with very little additional overhead (vs. single items).

Some thread-local data is required for each producer, and thread-local data can also optionally be used to speed up consumers. This thread-local data can be either associated with user-allocated *tokens*, or, to simplify the interface, if no token is provided by the user for a producer, a lock-free hash table is used (keyed to the current thread ID) to look up a thread-local producer queue: An SPMC queue is created for each explicitly allocated producer token, and another *implicit* one for each thread that produces items without providing a token. Since tokens contain what amounts to thread-specific data, they should never be used from multiple threads simultaneously (although it's OK to transfer ownership of a token to another thread; in particular, this allows tokens to be used inside thread-pool tasks even if the thread running the task changes part-way through).

All the producer queues link themselves together into a lock-free linked list. When an explicit producer no longer has elements being added to it (i.e. its token is destroyed), it's flagged as being unassociated to any producer, but it's kept in the list and its memory

is not freed; the next new producer reuses the old producer's memory (the lock-free producer list is add-only this way). Implicit producers are never destroyed (until the high-level queue itself is) since there's no way to know whether a given thread is done using the data structure or not. Note that the worst-case dequeue speed depends on how many producers queues there are, even if they're all empty.

There is a fundamental difference in the lifetimes of the explicit versus implicit producer queues: the explicit one has a finite production lifetime tied to the token's lifetime, whereas the implicit one has an unbounded lifetime and exists for the duration of the high-level queue itself. Because of this, two slightly different SPMC algorithms are used in order to maximize both speed and memory usage. In general, the explicit producer queue is designed to be slightly faster and hog slightly more memory, while the implicit producer queue is designed to be slightly slower but recycle more memory back into the high-level queue's global pool. For best speed, always use an explicit token (unless you find it too inconvenient).

Any memory allocated is only freed when the high-level queue is destroyed (though there are several re-use mechanisms). Memory allocation can be done up front, with operations that fail if there's not enough memory (instead of allocating more). Various default size parameters (and the memory allocation functions used by the queue) can be overridden by the used if desired.

Full API (pseudocode)

```
# Allocates more memory if necessary

enqueue(item) : bool

enqueue(prod_token, item) : bool
```

```
    enqueue_bulk(item_first, count) : bool

    enqueue_bulk(prod_token, item_first, count) : bool


    # Fails if not enough memory to enqueue

    try_enqueue(item) : bool

    try_enqueue(prod_token, item) : bool

    try_enqueue_bulk(item_first, count) : bool

    try_enqueue_bulk(prod_token, item_first, count) : bool


    # Attempts to dequeue from the queue (never allocates)

    try_dequeue(item&) : bool

    try_dequeue(cons_token, item&) : bool

    try_dequeue_bulk(item_first, max) : size_t

    try_dequeue_bulk(cons_token, item_first, max) : size_t


    # If you happen to know which producer you want to
dequeue from

    try_dequeue_from_producer(prod_token, item&) : bool

    try_dequeue_bulk_from_producer(prod_token, item_first,
max) : size_t


    # A not-necessarily-accurate count of the total number
of elements

    size_approx() : size_t
```

Producer Queue (SPMC) Design

Shared Design Across Implicit and Explicit Versions

The producer queue is made up of blocks (both the explicit and implicit producer queues use the same block objects to allow better memory sharing). Initially, it starts out with no blocks. Each block can hold a fixed number of elements (all blocks have the same capacity which is a power of 2). Additionally, blocks contain a flag indicating whether a filled slot has been completely consumed or not (used by the explicit version to determine when a block is empty), as well as an atomic counter of the number of elements completely dequeued (used by the implicit version to determine when a block is empty).

The producer queue, for the purposes of lock-free manipulation, can be thought of as an abstract infinite array. A *tail index* indicates the next available slot for the producer to fill; it also doubles as the count of the number of elements ever enqueued (the *enqueue count*). The tail index is written to solely by the producer, and always increases (except when it overflows and wraps around, which is still considered "increasing" for our purposes). Producing an item is trivial owing to the fact that only a single thread is updating the variables involved. A *head index* indicates what element can be consumed next. The head index is atomically incremented by the consumers, potentially concurrently. In order to prevent the head index from reaching/passing the perceived tail index, an additional atomic counter is employed: the *dequeue count*. The dequeue count is optimistic, i.e. it is incremented by consumers when they speculatively believe there is something to dequeue. If the value of the dequeue count after being incremented is less than the enqueue count (tail), then there is guaranteed to be

at least one element to dequeue (even taking concurrency into account), and it is safe to increment the head index, knowing that it will be less than the tail index afterwards. On the other hand, if after being incremented the dequeue count exceeds (or is equal to) the tail, the dequeue operation fails and the dequeue count is logically decremented (to keep it eventually consistent with the enqueue count): this could be done by directly decrementing the dequeue count, but instead (to increase parallelism and keep all variables involved increasing monotonically) a *dequeue overcommit* counter is incremented instead. To get the *logical* value of the dequeue count, we subtract the dequeue overcommit value from the dequeue count variable.

When consuming, once a valid index is determined as outlined above, it still needs to be mapped to a block and an offset into that block; some sort of indexing data structure is used for this purpose (which one depends on whether it's an implicit or explicit queue). Finally, the element can be moved out, and some sort of status is updated so that it's possible to eventually know when the block is completely spent. Full descriptions of these mechanisms are provided below in the individual sections covering the implicit- and explicit-specific details.

The tail and head indices/counts, as previously mentioned, will eventually overflow. This is expected and taken into account. The index/count is thus considered as existing on a circle the size of the maximum integer value (akin to a circle of 360 degrees, where 359 precedes 1). In order to check whether one index/count, say `a`, comes before another, say `b`, (i.e., logical less-than) we must determine whether `a` is closer to `b` via a clockwise arc on the circle or not. The following algorithm for circular less-than is used (32-bit version): `a < b` becomes `a - b > (1U << 31U)`. `a <= b` becomes `a - b - 1ULL > (1ULL << 31ULL)`. Note that circular subtraction "just works" with normal unsigned integers

(assuming two's complement). Care is taken to ensure that the tail index is not incremented past the head index (which would corrupt the queue). Note that despite this, there's still technically a race condition in which an index value that the consumer (or producer, for that matter) sees is so stale that it's almost a whole circle's worth (or more!) behind its current value, causing the internal state of the queue to become corrupted. In practice, however, this is not an issue, because it takes a while to go through 2^31 values (for a 32-bit index type) and the other cores would see something more up to date by then. In fact, many lock-free algorithms are based on a related tag-pointer idiom, where the first 16 bits is used for a tag that gets repeatedly incremented, and the second 16 bits for a pointer value; this relies on the similar assumption that one core can't increment the tag more than 2^15 times without the other cores knowing so. Nevertheless, the default index type for the queue is 64 bits wide (which, if even 16-bits seems to be enough, should prevent any potential races even in theory).

Memory allocation failure is also handled properly and will never corrupt the queue (it is simply reported as failure). However, the elements themselves are assumed never to throw exceptions while being manipulated by the queue.

Block Pools

There are two different pools of blocks that are used: First, there is the initial array of pre-allocated blocks. Once consumed, this pool remains empty forever. This simplifies its wait-free implementation to a single fetch-and-add atomic instruction (to get the next index of a free block) with a check (to make sure that that index is in range). Second, there is a lock-free (though not wait-free) global free list ("global" meaning global to the high-level queue) of spent blocks that are ready to be re-used, implemented as a lock-free singly linked list: A head pointer initially points to

nothing (null). To add a block to the free list, the block's *next* pointer is set to the head pointer, and the head pointer is then updated to point at the block using compare-and-swap (CAS) on the condition that the head hasn't changed; if it has, the process is repeated (this is a classic lock-free CAS-loop design pattern). To remove a block from the free list, a similar algorithm is used: the head block's next pointer is read, and the head is then set to that next pointer (using CAS) conditional to the fact that the head hasn't changed in the meantime. To avoid the ABA problem, each block has a reference count which is incremented before doing the CAS to remove a block, and decremented afterwards; if an attempt is made to re-add a block to the free list while its reference count is above 0, then a flag indicating that the block should be on the free list is set, and the next thread that finishes holding the last reference checks this flag and adds the block to the list at that time (this works because we don't care about order). I've described the exact design and implementation of this lock-free free list in further detail in another blog post. When a producer queue needs a new block, it first checks the initial block pool, then the global free list, and only if it can't find a free block there does it allocate a new one on the heap (or fail, if memory allocation is not allowed).

Explicit Producer Queue

The explicit producer queue is implemented as a circular singly-linked list of blocks. It is wait-free on the fast path, but merely lock-free when a block needs to be acquired from a block pool (or a new one allocated); this only happens when its internal cache of blocks are all full (or there are none, which is the case at the beginning).

Once a block is added into an explicit producer queue's circular linked list, it is never removed. A *tail block* pointer is maintained by the producer that points to the block which elements are currently being inserted into; when the tail block is full, the next block is

checked to determine if it is empty. If it is, the tail block pointer is updated to point to that block; if not, a new block is requisitioned and inserted into the linked list immediately following the current tail block, which is then updated to point to this new block.

When an element is finished being dequeued from a block, a per-element flag is set to indicate that the slot is full. (Actually, all the flags start off set, and are only turned off when the slot becomes empty.) The producer checks if a block is empty by checking all of these flags. If the block size is small, this is fast enough; otherwise, for larger blocks, instead of the flag system, a block-level atomic count is incremented each time a consumer finishes with an element. When this count is equal to the size of the block, or all the flags are off, the block is empty and can be safely re-used.

To index the blocks in constant time (i.e. quickly find the block that an element is in given that element's global index from the dequeue algorithm), a circular buffer (contiguous array) is used. This index is maintained by the producer; consumers read from it but never write to it. At the *front* of the array is the most recently written-to block (the tail block). At the *rear* of the array is the last block that may have elements in it. It is helpful to think of this index (from a high-level perspective) as a single long ribbon of the history of which blocks have been used. The front is incremented whenever the producer starts on another block (which may be either newly allocated or re-used from its circular list of blocks). The rear is incremented whenever a block that was already in the circular list is re-used (since blocks are only re-used when they're empty, it is always safe to increment the rear in this case). Instead of storing the rear explicitly, a count of the number of slots used is kept (this avoids the need for a spare element in the circular buffer, and simplifies the implementation). If there is not enough room in the index to add a new item, a new index array is allocated which is twice the size of the previous array (obviously, this is only allowed if

memory allocation is allowed -- if not, the entire enqueue operation fails gracefully). Since consumers could still be using the old index, it is not freed, but instead simply linked to the new one (this forms a chain of index blocks that can be properly freed when the high-level queue is destructed). When the enqueue count of the producer queue is incremented, it releases all writes to the index; when a consumer performs an acquire (which it already needs for the dequeue algorithm), then from that point on whichever index the consumer sees will contain a reference to the block that the consumer is interested in. Since blocks are all the same size, and a power of 2, we can use shifting and masking to determine the number of blocks our target block is offset from any other block in the index (and the offset in the target block) provided we know the base index of a given block in the index. So, the index contains not just block pointers, but also the corresponding base index of each of those blocks. The block that is chosen as a reference point (to compute an offset against) in the index must be not be overwritten by the producer while it is being used -- using the (perceived) front of the index as the reference point guarantees this since (knowing the block index is at least as up to date as the enqueue count which is ahead the dequeue index we're looking up) the front of the index has to be at or ahead of the target block, and the target block will never be overwritten in the index until it (and all blocks before) is empty, and it can't be empty until after the dequeue operation itself finishes. The index size is a power of two, which allows for faster wrapping of the front/rear variables.

The explicit producer queue requires a user-allocated "producer token" to be passed when enqueueing. This token merely contains a pointer to a producer queue object. When the token is created, a corresponding producer queue is created; when the token is destroyed, the producer queue may still contain unconsumed elements, and hence the queue itself outlives the token. In fact, once allocated, a producer queue is never destroyed (until the high-

level queue is destructed), but it *is* re-used the next time a producer token is created (instead of resulting in a heap allocation for a new producer queue).

Implicit Producer Queue

The implicit producer queue is implemented as an un-linked set of blocks. It lock-free, but not wait-free, because the implementation of the main free block free-list is lock-free, and blocks are continuously acquired from and inserted back into that pool (resizing the block index is also not constant-time, and requires memory allocation). The actual enqueue and dequeue operations are still wait free within a single block.

A *current* block pointer is maintained; this is the block that is currently being enqueued in. When a block fills up, a new one is requisitioned, and the old one (from the producer's perspective) is forgotten about. Before an element is added to a block, the block is inserted in the block index (which allows consumers to find blocks the producer has already forgotten about). When the last element in a block is finished being consumed, the block is logically removed from the block index.

The implicit producer queue is never re-used -- once created, it lives throughout the lifetime of the high-level queue. So, in order to reduce memory consumption, instead of hogging all the blocks that it once used (like the explicit producer), it instead returns spent blocks to the global free list. To do this, an atomic dequeue count in each block is incremented as soon as a consumer is done dequeueing an item; when the counter reaches the size of the block, the consumer that sees this knows that it just dequeued the last item, and puts the block in the global free list.

The implicit producer queue uses a circular buffer to implement its block index, which allows constant-time searching for a block

given its base index. Each index entry is composed of a key-value pair representing the base index of the block, and a pointer to the corresponding block itself. Since blocks are always inserted in order, the base index of each block in the index is guaranteed to increase by exactly one block size's worth between adjacent entries. This means that any block that's known to be in the index can be easily found by looking at the last inserted base index, computing the offset to the desired base index, and looking up the index entry at that offset. Special care is taken to ensure that the arithmetic still works out when the block index wraps around (although it is assumed that at any given moment the same base index won't be present (or seen to be present) twice in the index).

When a block is spent, it is removed from the index (to make room for future block insertions); since another consumer could still be making use of that entry in the index (to calculate an offset), the index entry is not removed completely, but the block pointer is set to null instead, indicating to the producer that the slot can be re-used; the block base is left untouched for any consumers that are still using it to calculate an offset. Since the producer only re-uses a slot once all preceding slots are also free, and when a consumer is looking up a block in the index there's necessarily at least one non-free slot in the index (corresponding to the block it's looking up), and the block index entry the consumer is using to look up a block is at least as recently enqueued as that the one for that block, there is never a race condition between the producer re-using a slot and a consumer looking up a block using that slot.

When the consumer wishes to enqueue an item and there's no room in the block index, it (if permitted) allocates another block index (linked to the old one so that its memory can eventually be freed when the queue is destructed), which becomes the main index from then on. The new index is a copy of the old one, except twice as large; copying over all the index entries allows consumers

to only have to look in one index to find the block they're after (constant time dequeueing within a block). Because a consumer could be in the process of marking an index entry as being free (by setting the block pointer to null) when the new index is being constructed, the index entries themselves are not copied, but rather pointers to them. This ensures that any change to an old index by a consumer properly affects the current index as well.

Hash of Implicit Producer Queues

A lock-free hash-table is used to map thread IDs to implicit producers; this is used when no explicit producer token is provided for the various enqueueing methods. It is based on Jeff Preshing's lock-free hash algorithm, with a few adjustments: the keys are the same size as a platform-dependent numeric thread ID type; the values are pointers; when the hash becomes too small (the number of elements is tracked with an atomic counter) a new one is allocated and linked to the old one, and elements in the old one are transferred lazily as they are read. Since an atomic count of the number of elements is available, and elements are never deleted, a thread that wishes to insert an element in a hash table that is too small must attempt to resize or wait for another thread to finish resizing (resizing is protected with a lock to prevent spurious allocations). In order to speed up resizing under contention (i.e. minimize the amount of spin-waiting that threads do waiting for another thread to finish allocating), items can be inserted in the old hash table up to a threshold that is significantly greater than the threshold that triggers resizing (e.g. a load factor of 0.5 would cause a resize to commence, and in the meantime elements can be inserted in the old one up to a load factor of 0.75, say).

Linked List of Producers

As mentioned previously, a singly-linked (LIFO) list of all the producers is maintained. This list is implemented using a tail pointer, and intrusive *next* (really, "prev") pointers for each producer. The tail initially points to null; when a new producer is created, it adds itself to the list by first reading the tail, then using setting its next to that tail, then using a CAS operation to set the tail (if it hasn't changed) to the new producer (looping as necessary). Producers are never removed from the list, but can be marked as inactive.

When a consumer wants to dequeue an item, it simply walks the list of producers looking for an SPMC queue with an item in it (since the number of producers is unbounded, this is partly what makes the high-level queue merely lock-free instead of wait-free).

Dequeue Heuristics

Consumers may pass a token to the various dequeue methods. The purpose of this token is to speed up selection of an appropriate inner producer queue from which to attempt to dequeue. A simple scheme is used whereby every explicit consumer is assigned an auto-incrementing offset representing the index of the producer queue it should dequeue from. In this fashion, consumers are distributed as fairly as possible across the producers; however, not all producers have the same number of elements available, and some consumers may consume faster than others; to address this, the first consumer that consumes 256 items in a row from the same inner producer queue increments a global offset, causing all the consumers to rotate on their next dequeue operation and start consuming from the next producer. (Note that this means the rate of rotation is dictated by the fastest consumer.) If no elements are available on a consumer's designated queue, it moves on to the next one that has an element available. This simple heuristic is efficient and is able to pair consumers with producers with near-perfect scaling, leading to impressive dequeue speedups.

A Note About Linearizability

A data structure is *linearizable* (this paper has a good definition) if all of its operations appear to execute in some sequential (linear) order, even under concurrency. While this is a useful property in that it makes a concurrent algorithm obviously correct and easier to reason about, it is a very *strong* consistency model. The queue I've presented here is not linearizable, since to make it so would result in much poorer performance; however, I believe it is still quite useful. My queue has the following consistency model: Enqueue operations on any given thread are (obviously) linearizable on that thread, but not others (this should not matter since even with a fully linearizable queue the final order of elements is non-deterministic since it depends on races between threads). Note that even though enqueue operations are not linearizable across threads, they are still atomic -- only elements that are completely enqueued can be dequeued. Dequeue operations are allowed to fail if all the producer queues appeared empty *at the time they were checked*. This means dequeue operations are also non-linearizable, because the queue as a whole was not necessarily empty at any one point during a failed dequeue operation. (Even a single producer queue's emptiness check is technically non-linearizable, since it's possible for an enqueue operation to complete but the memory effects not yet be propagated to a dequeueing thread -- again, this shouldn't matter anyway since it depends on non-deterministic race conditions either way.)

What this non-linearizability means in practical terms is that a dequeue operation may fail before the queue is completely empty *if there are still other producers enqueueing* (regardless of whether other threads are dequeueing or not). Note that this this race condition exists anyway even with a fully linearizable queue. If the queue has stabilized (i.e. all enqueue operations have completed and their memory effects have become visible to any potential

consumer threads), then a dequeue operation will never fail as long as the queue is not empty. Similarly, if a given set of elements is visible to all dequeueing threads, dequeue operations on those threads will never fail until at least that set of elements is consumed (but may fail afterwards even if the queue is not completely empty).

Conclusion

So there you have it! More than you ever wanted to know about my design for a general purpose lock-free queue. I've implemented this design using C++11, but I'm sure it can be ported to other languages. If anybody does implement this design in another language I'd love to hear about it!

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).

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!