# MultiList - A MPSC Wait-Free Queue

Andreia Correia
Concurrency Freaks
andreiacraveiroramalhete@gmail.com

Pedro Ramalhete
Cisco Systems
pramalhe@gmail.com

## ABSTRACT

Memory-unbounded linearizable non-blocking queues have at its core a singly-linked list, with one or multiple items per node. Constructing a non-blocking queue backed by multiple lists is certainly possible, but until now, none existed that was capable of providing First-In-First-Out (FIFO) order with linearizable consistency.

We present MultiList queue, the first Multi-Producer-Single-Consumer wait-free queue backed by multiple singly-linked lists, capable of providing FIFO order with linearizable consistency and with fast enqueues. Results from our benchmarks show that MultiList surpasses the current fastest wait-free and lock-free linearizable queues for enqueueing.

## Categories and Subject Descriptors

D.4.1.f [**Operating Systems**]: Synchronization

## Keywords

queues; non-blocking; wait-free

## 1. INTRODUCTION

Several linearizable lock-free and wait-free queues exist in the literature. Of the memory-unbounded ones, all of them are backed by a singly-linked list, where each node may contain one item or an array with multiple items. The simplest of these lock-free queues is undoubtedly the algorithm created by Maged Michael and Michael Scott (MS queue), which is used as the basis for fast lock-free queues like LCRQ [5], and the basis of several wait-free queues [3, 6].

Queues backed by a singly-linked list have an inherent contention point on their enqueues, where multiple threads attempting an enqueue will repeatedly do a Compare-And-Swap (CAS) on the last node of the list. Fast queues go around this issue by either allowing multiple items per node [5], or by inserting multiple nodes with a single CAS [2].

Memory-unbounded queues backed by multiple lists exist in the literature [4], though none provides First-In-First-Out (FIFO) order and linearizable consistency and non-blocking progress.

We present `MultiList` queue, the first memory-unbounded Multi-Producer-Single-Consumer (MPSC) wait-free queue backed by multiple singly-linked lists, with fast enqueueing even under high contention, and linearizability for `enqueue()` and `dequeue()` methods.

ACM ISBN 978-1-4503-2138-9.
DOI: 10.1145/1235

## 2. ALGORITHM

**Algorithm 1** Variables of the MultiList queue class

```
std::atomic<uint64_t> currTS {0};
Node* heads[MAX_THREADS];
Node* tails[MAX_THREADS];

struct Node {
  T* item;
  Node* next;
  std::atomic<uint64_t> ts;
  Node() : ticket{MAX_UINT} { }
};
```

The main idea of the MultiList queue is that a thread executing an enqueue has its own single-linked on which it will insert its node. A node in one of these singly-linked lists has: a pointer to the item, a pointer to the next node, and an atomic integer to represent the round. Algorithm 1 shows C++ code for the node and class variables of MultiList.

Algorithm 2 shows the C++ code for `enqueue()`. Because each enqueuer has its own list, the only synchronization needed is the store with `memory_order_release` on the `ts` (line 8 of Algorithm 2) making visible to the dequeuer the `item` and the new node, and a Fetch-And-Add to increment the global counter, named `currTS` The `ts` represents the *timestamp* in which the item was inserted in the queue. The `ts` of each node needs not be unique, and other nodes in other lists may have the same value, as long as it is unique per list and increases

**Algorithm 2** Enqueue algorithm

```
void enqueue(T* item , const int tid) {
  if (item == nullptr) throw std::invalid_argument("null_item");
  uint64_t ts = currTS.load();
  Node* ltail = tails[tid];
  tails[tid] = new Node;
  ltail->item = item;
  ltail->next = tails[tid];
  ltail->ts.store(ts, std::memory_order_release);
  if (currTS.load() == ts) currTS.fetch_add(1);
}
```

for successive nodes in the same list. If multiple nodes in different lists have the same timestamp, the dequeuer can dequeue them in any order it chooses to, because these node's enqueuing windows overlapped in time.

Each enqueuer has a thread-local tail with a pointer to the last enqueued node, which in our implementation is an entry in the `tails` array at position `tid`. Although the enqueuer creates a new node, it stores its item and timestamp on the previous node. This is relevant for the dequeuer because it will increase throughput by reducing the number of de-references.

In contrast with the simplicity of the enqueue operation, the dequeue has to determine in which list is the oldest item in the queue, to guarantee it is the first to be dequeued. The dequeuer scans the current head nodes and chooses to dequeue the node with the lowest `ts`. For this, the dequeue will scan the head of each of the singly-linked lists and check if any of them has an item to be dequeued, i.e. if its timestamp is lower than `MAX_UINT` (line 8 of Algorithm 3).

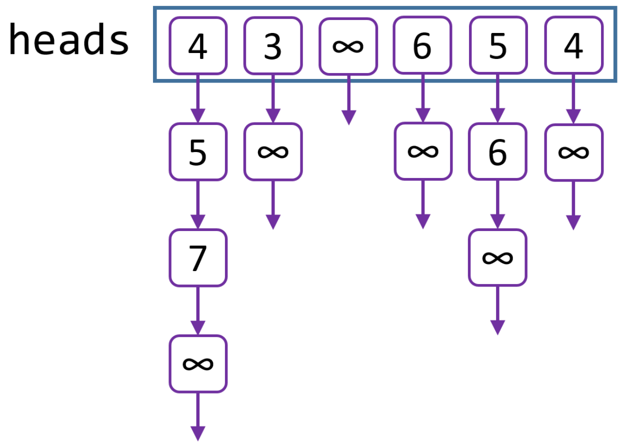
**Algorithm 3** Dequeue algorithm for single consumer

```
1  T* dequeue(const int tid) {
2    int prevIdx = -2;
3    while (true) {
4      uint64_t minTS = MAX_UINT;
5      int minIdx = -1;
6      for (int idx = 0; idx < maxThreads; idx++) {
7        uint64_t lts = heads[idx]->ts.load();
8        if (lts < minTS) {
9          minTS = lts;
10         minIdx = idx;
11       }
12     }
13     if (minIdx == -1 && prevIdx == minIdx) return nullptr;
14     if (prevIdx == minIdx) {
15       Node* lhead = heads[minIdx];
16       T* item = lhead->item;
17       heads[minIdx] = lhead->next;
18       delete lhead;
19       return item;
20     }
21     prevIdx = minIdx;
22   }
23 }
```

Naively, it is tempting to assume a single scan would be sufficient to find the lowest oldest item in the queue, but unfortunately this is not true. Consider the scenario where the dequeuer is scanning the array of heads and that when it scans entry $i$, its entry `heads[i].ts` is `MAX_UINT`. Immediately after, enqueuer $i$ will add a new node, and returns. Upon returning, thread $i$ sets an atomic flag to indicate that its enqueuing has completed and this flag is seen by thread $i+1$, triggering an enqueing operation by it. A new item is inserted in the $i+1$ list, with a timestamp higher than `heads[i].ts`, however, now that the dequeuer continues its scan, it will chose $i+1$ as being the next node to be dequeued because there was no other node. Such behavior would **not** provide FIFO order.

There is a simple way to solve this problem, the answer is to execute *two consecutive scans with the same result*. Scanning once for the node with the lowest timestamp is not enough, but if there are two consecutive scans that indicate the same node as being the lowest, then the linearization point becomes the load on the node done on the first scan. If the second scan gives a different answer, then the scan must be repeated until two consecutive scans yield the same node as having the lowest timestamp. The maximum number of scans that will have to be done on the `heads` array is `maxThreads+1` because each node can only change its timestamp one single time from `MAX_UINT` to a valid timestamp, and becomes immutable after changing it. This means the `dequeue()` method has wait-free bounded progress, with a bound on the number of steps proportional to `maxThreads`.

Figure 2 shows an example state for a queue with six possible enqueuers, in which the dequeue order for each of the indexes will be the following (assuming no further enqueues): 1, 0, 5, 0, 4, 3, 4, 0.



There are two innovations in MultiList that allow the usage of multiple lists with linearizable consistency. The first innovation is the adding of the `ts` in the node. This provides a FIFO order among nodes of different lists. The second innovation is the execution of two consecutive scans to determine the node with the lowest timestamp. The two scans prevent an earlier enqueuer, with a lower timestamp, of being missed in favor of a later enqueuer, with a higher timestamp.

In our MPSC queue algorithm each enqueuer only dereferences nodes from its own list, and therefore, the single consumer can delete the dequeued nodes immediately, without the need for memory reclamation techniques like Hazard Pointers.

## 3. DISCUSSION AND IMPROVEMENTS

It is possible to adapt this algorithm to be Multi-Producer-Multi-Consumer with lock-free or wait-free `dequeue()`. The array of `heads` must be `std::atomic<>` and the advance of each head has to be done with a CAS, using the same algorithm as the MS queue. This modification requires the usage of a Garbage Collector or a technique for memory reclamation, like Hazard Pointers.

Having one list uniquely assigned to a thread is not always possible or easy in practice, therefore, the MultiList queue can be modified to have multiple threads enqueueing in the same list, for example by using the MS queue algorithm for enqueueing, where the list is chosen based on a random number or hashing of the thread's id over the $k$ available lists. Again here we need to handle memory reclamation.

When the queue is mostly filled in all its lists, it can be interesting to add a `no_max_uint` flag in the `dequeue()` method, that is set to true when *all* of the heads timestamps are lower than `MAX_UINT`. If this flag is set, then a single scan of the for lowest timestamp is enough, because there is the guarantee that no other node can be inserted during the scan. Even better, the number of nodes with non-`MAX_UINT` timestamps can be saved for the next call to `dequeue()` and if it is the same on the first scan of the following `dequeue()`, that single scan is enough to provide linearizable consistency.

One modification that can be done on `enqueue()` with great potencial increase in throughput, is to replace the load on line 3 (and subsequent FAA on line 9) with a single load from a hardware register that provides a monotonic timestamp with linearizable properties across threads/cores, or alternatively, a function call that emulates such a behavior. Such a modification requires hardware support (like recent x86's tsc) or OS specific support (like the Java HotSpot JVM `System.currentTimeMillis()`) [1].

## 4. THROUGHPUT

The typical benchmark for MPMC queues is for each thread to do a single enqueue followed by a single dequeue, however, our implementation is MPSC (single consumer), instead, we do a burst of $10^8$ enqueue operations, measure the time it took and divide by the total number of operations. The median of 5 runs is shown in figure 1 for our wait-free MultiList queue, the lock-free MS queue, the lock-free LCRQ, and the wait-free SimQueue. LCRQ is one of the fastest MPMC lock-free queues and SimQueue is one of the fastest MPMC wait-free queues. The LCRQ implementation is using an array of 1024 entries, while the other queues have one item per node. There is no random sleep between each enqueue, so as to maximize the effects of the high contention.

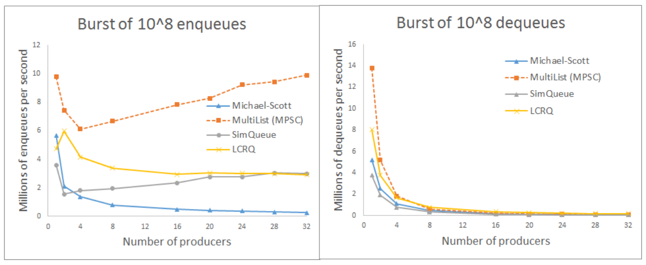


**Figure 1: Burst of $10^8$ enqueues and dequeues with single consumer. Higher is better.**

The left plot of figure 1 shows the total number of enqueue operations as the number of enqueuers increases. Both SimQueue and MultiList have a drop with two threads but they eventually recover and MultiList starts to have positive scalability with 32 enqueuer threads, being more than 2x faster than SimQueue and 40x faster than MS under high contention.

The right plot of figure 1 shows the number of `dequeue()` per second done by a single consumer as the number of producers increases, normalized to Michael-Scott. Due to cache effects there is a decrease as the number of producers increases on all queues, and on MultiList there is a higher decrease due to having to scan more lists to determine the next node to dequeue.

## 5. CONCLUSION

Results from our microbenchmark show that MultiList queue is the fastest wait-free MPSC queue, surpassing the currently fastest wait-free queue SimQueue, and the fastest lock-free MPMC queue LCRQ. The high throughput of `enqueue()` is a trade-off resulting from a design choice that leaves most of the work to be done by the `dequeue()`.

With MultiList it is now possible to have a multi-list backed queue with linearizable operations. Accomplishing this, required two tricks: adding a variable in the node that represents the timestamp of insertion, and doing two consecutive scans to determine what is the node with the lowest timestamp, so that it is the next item to be dequeued. The outcome of this design is a fast (for enqueues) memory-unbounded wait-free MPSC queue with safe memory reclamation.

## 6. REFERENCES

[1] Click, C. A JVM Does That? https://www.youtube.com/watch?v=-vizTDSz8NU&t=14m22s, 2016.

[2] Fatourou, P., and Kallimanis, N. D. A highly-efficient wait-free universal construction. In *Proceedings of the twenty-third annual ACM symposium on Parallelism in algorithms and architectures* (2011), ACM, pp. 325–334.

[3] Kogan, A., and Petrank, E. Wait-free queues with multiple enqueuers and dequeuers. In *ACM SIGPLAN Notices* (2011), vol. 46, ACM, pp. 223–234.

[4] Mellor-Crummey, J. M. Concurrent queues: Practical fetch-and-phi algorithms. Tech. rep., DTIC Document, 1987.

[5] Morrison, A., and Afek, Y. Fast concurrent queues for x86 processors. In *ACM SIGPLAN Notices* (2013), vol. 48, ACM, pp. 103–112.

[6] Ramalhete, P., and Correia, A. A Wait-Free Queue with Wait-Free Memory Reclamation. https://github.com/pramalhe/ConcurrencyFreaks/papers/crturnqueue-2016.pdf, 2016.