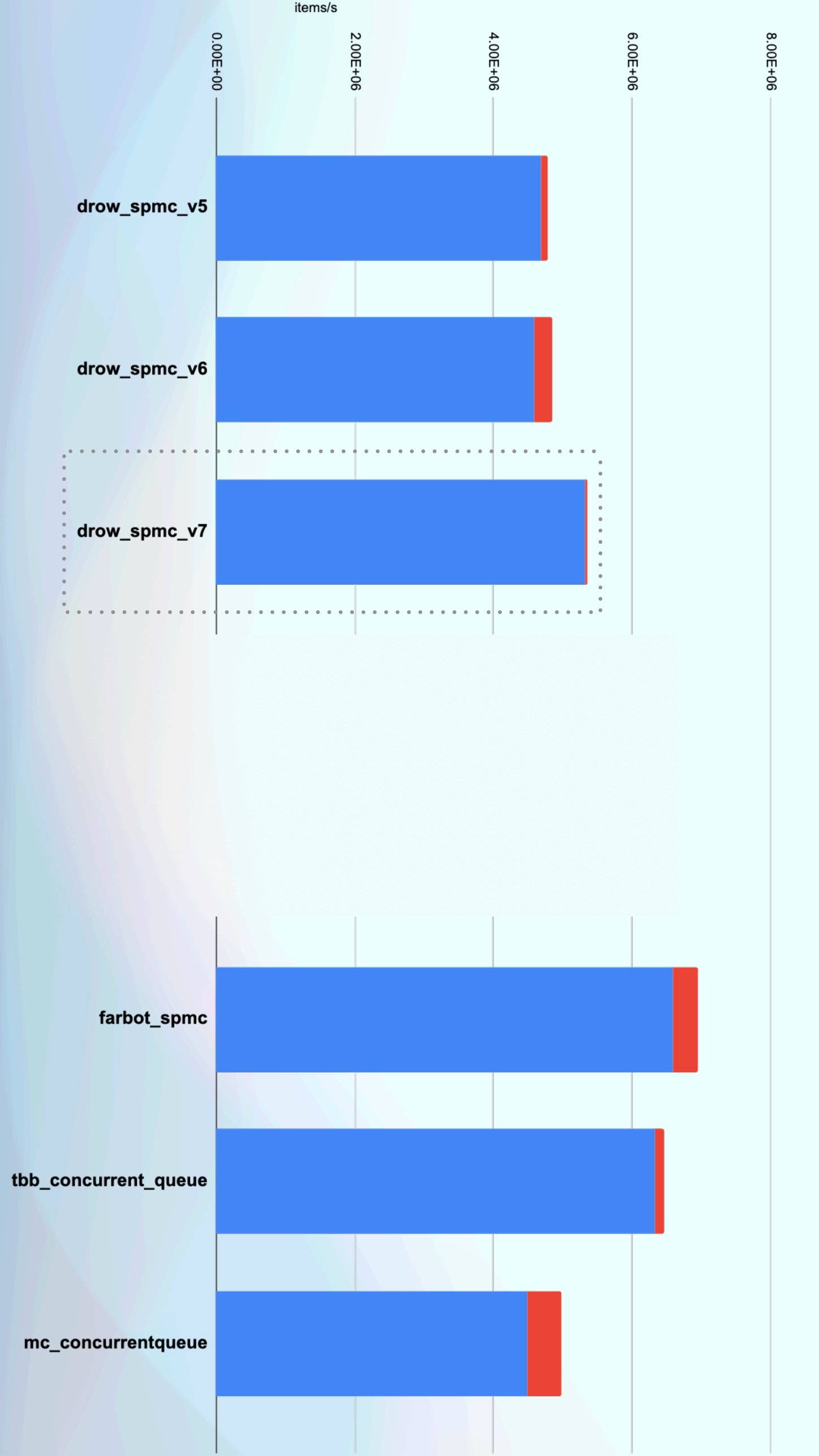
```cpp
template<typename T>
class drow_spmc_v7
{
public:
    drow_spmc_v7 (size_t capacity_)
        : capacity (std::bit_ceil (capacity_))
    {
        // Initialise each slot's sequence number to its index
        for (size_t i = 0; i < capacity; ++i)
            slots[i].sequence.store (i, std::memory_order_relaxed);
    }

    bool try_push (const T& v)
    {
        size_t current_tail = cached_tail;
        size_t index = current_tail & (capacity - 1);
        auto& slot = slots[index];

        size_t seq = slot.sequence.load (std::memory_order_acquire);

        if (size_t diff = seq - current_tail;
            diff == 0)
        {
            slot.data = v;

            // Mark slot as ready for reading
            cached_tail = current_tail + 1;
            slot.sequence.store (cached_tail, std::memory_order_release);
            tail.store (cached_tail, std::memory_order_relaxed);

            return true;
        }

        // Queue is full (slot hasn't been consumed yet)
        return false;
    }

private:
    struct slot
    {
        std::atomic<size_t> sequence;
        T data;
    };

    size_t capacity = 0;
    alignas(hardware_destructive_interference_size) std::atomic<size_t> head { 0 };
    alignas(hardware_destructive_interference_size) size_t cached_tail { 0 };
    alignas(hardware_destructive_interference_size) std::atomic<size_t> tail { 0 };
    std::vector<slot> slots { std::vector<slot> (capacity) };
};
```

```
bool try_pop (T& v)
{
    for (;;)
    {
        size_t current_head = head.load (std::memory_order_relaxed);
        size_t index = current_head & (capacity - 1);
        auto& slot = slots[index];

        size_t seq = slot.sequence.load (std::memory_order_acquire);
        ssize_t diff = seq - (current_head + 1);

        if (diff == 0) // Slot is ready for reading
        {
            // Try to claim this slot
            if (head.compare_exchange_weak(current_head, current_head + 1,
                                           std::memory_order_relaxed,
                                           std::memory_order_relaxed))
            {
                // Successfully claimed, read data
                v = slot.data;

                // Mark slot as consumed and ready for writing
                slot.sequence.store (current_head + capacity, std::memory_order_release);

                return true;
            }

            // CAS failed, another consumer claimed it, retry
        }
        else if (diff < 0) // Queue is empty
        {
            return false;
        }

        // If diff > 0, slot isn't ready yet, retry
    }
}
```

```cpp
template<typename T>
class drow_spmc_v7
{
public:
    drow_spmc_v7 (size_t capacity_)
        : capacity (std::bit_ceil (capacity_))
    {
        // Initialise each slot's sequence number to its index
        for (size_t i = 0; i < capacity; ++i)
            slots[i].sequence.store (i, std::memory_order_relaxed);
    }

    bool try_push (const T& v)
    {
        size_t current_tail = cached_tail;
        size_t index = current_tail & (capacity - 1);
        auto& slot = slots[index];

        size_t seq = slot.sequence.load (std::memory_order_acquire);

        if (size_t diff = seq - current_tail;
            diff == 0)
        {
            slot.data = v;

            // Mark slot as ready for reading
            cached_tail = current_tail + 1;
            slot.sequence.store (cached_tail, std::memory_order_release);
            tail.store (cached_tail, std::memory_order_relaxed);

            return true;
        }

        // Queue is full (slot hasn't been consumed yet)
        return false;
    }

private:
    struct slot
    {
        std::atomic<size_t> sequence;
        T data;
    };

    size_t capacity = 0;
    alignas(hardware_destructive_interference_size) std::atomic<size_t> head { 0 };
    alignas(hardware_destructive_interference_size) size_t cached_tail { 0 };
    alignas(hardware_destructive_interference_size) std::atomic<size_t> tail { 0 };
    std::vector<slot> slots { std::vector<slot> (capacity) };
};
```

```cpp
    bool try_pop (T& v)
    {
        for (;;)
        {
            size_t current_head = head.load (std::memory_order_relaxed);
            size_t index = current_head & (capacity - 1);
            auto& slot = slots[index];

            size_t seq = slot.sequence.load (std::memory_order_acquire);
            ssize_t diff = seq - (current_head + 1);

            if (diff == 0) // Slot is ready for reading
            {
                // Try to claim this slot
                if (head.compare_exchange_weak(current_head, current_head + 1,
                                               std::memory_order_relaxed,
                                               std::memory_order_relaxed))
                {
                    // Successfully claimed, read data
                    v = slot.data;

                    // Mark slot as consumed and ready for writing
                    slot.sequence.store (current_head + capacity, std::memory_order_release);

                    return true;
                }

                // CAS failed, another consumer claimed it, retry
            }
            else if (diff < 0) // Queue is empty
            {
                return false;
            }

            // If diff > 0, slot isn't ready yet, retry
        }
    }
```


items/s
0.00E+00
2.00E+06
4.00E+06
6.00E+06
8.00E+06
drow_spmc_v5
drow_spmc_v6
drow_spmc_v7
farbot_spmc
tbb_concurrent_queue
mc_concurrentqueue