```
template<typename queue_type>
void measure_serial(benchmark::State& state)
{
    const size_t queue_size = state.range(0);
    const size_t iters = 4 * queue_size;

    for (auto _: state)
    {
        // Initialise the queue half full
        queue_type queue(queue_size);

        for (size_t i = 0; i < (queue_size / 2); ++i)
            push (queue, i);

        stopwatch sw;

        for (size_t i = 0; i < iters; ++i)
        {
            if (! queue.try_push (i))
                continue;

            int v;
            if (! queue.try_pop (v))
                continue;

            benchmark::DoNotOptimize(v);
        }

        state.SetIterationTime(sw.get());
    }

    // Throughput is calculated as an item being enqueued and dequeued, so travelling fully through the queue
    state.SetItemsProcessed(int64_t(state.iterations()) * iters);
}
```

```
template<typename queue_type>
void measure_serial(benchmark::State& state)
{
    const size_t queue_size = state.range(0);
    const size_t iters = 4 * queue_size;

    for (auto _: state)
    {
        // Initialise the queue half full
        queue_type queue(queue_size);

        for (size_t i = 0; i < (queue_size / 2); ++i)
            push (queue, i);

        stopwatch sw;

        for (size_t i = 0; i < iters; ++i)
        {
            if (! queue.try_push (i))
                continue;

            int v;
            if (! queue.try_pop (v))
                continue;

            benchmark::DoNotOptimize(v);
        }

        state.SetIterationTime(sw.get());
    }

    // Throughput is calculated as an item being enqueued and dequeued, so travelling fully through the queue
    state.SetItemsProcessed(int64_t(state.iterations()) * iters);
}
```

```cpp
template<typename T>
class queue
{
public:
    queue (size_t capacity_)
        : capacity (capacity_)
    {}

    bool try_push (const T&);
    bool try_pop (T&);

private:
    size_t capacity = 0;
    std::vector<T> data { std::vector<T> (capacity) };
    size_t head { 0 }, tail { 0 };

    size_t next_index (size_t current) const
    {
        return (current + 1) % capacity;
    }
};
```

```cpp
bool try_push (const T& v)
{
    size_t current_tail = tail;
    size_t next_tail = next_index (current_tail);

    if (next_tail == head)
        return false;

    data[current_tail] = v;
    tail = next_tail;
    return true;
}
```

```cpp
bool try_pop (T& v)
{
    size_t current_head = head;

    if (current_head == tail)
        return false;

    v = data[current_head];
    head = next_index (current_head);
    return true;
}
```