# 華中科技大学

# 研究生课程报告

## 并发队列的实现实验报告

学　　号　M202470678

姓　　名　梅硕

专　　业　机械工程

课程指导老师　周健

院（系、所）　机械科学与工程学院

2024 年 12 月 28 日

# 目　　录

# 1 并发队列的实现方法

本次实践中，基于数组的无锁队列使用老师上课所介绍的方法实现，使用预分配的指针来控制对数组的读写不发生在同一位置（数组的循环复用问题后面讨论）。课后的资料查询中我推测该方法是 Michael and Scott 无锁队列算法的基于数组情况下的变体，原文[1] 给出的实现为基于链表的实现，细节上存在一定的差别，基于链表的实现方案中，使用 CAS 操作，首先将尾指针指向的后续为空的元素链接到新元素，然后调整尾指针指向新元素，因为这两步是分开进行的，因此需要保调整尾指针指向新元素时，尾指针原本指向的元素没有改变，为一开始指向的后续为空，当前指向新元素的节点。这一点是通过此两步操作前的初始化赋值确保的，并在 CAS 操作中进行比较判定而确认生效。基于数组的实现则是首先将预分配指针增加，然后在预分配位置写入数据，最后判断当前尾指针是否达到当前分配节点的前一步，如果成立则将尾指针指向当前节点。

上述操作是不难于理解的，但是具体实现上存在有一定的问题。使用循环数组的情况下，当入队元素超过数组大小，此时如何控制入队元素不覆写之前写入的元素。实际上，在无锁队列的实现中，向一个线程提供队列是否为空/满是没有意义的，即使提供的信息是正确的，在线程收到信息与调用出队或者入队操作间，已经可能有其他线程对队列进行了操作，队列的空/满信息变得无效了。

为解决上述问题，在队列操作内部实现对不合法操作的处理。基于需要与 TBB 中 concurrentqueue 队列对比的考虑，对于出队操作，参考 TBB::concurrent-queue 的实现，当尝试出队时如果队列为空，跳过操作，返回 bool 值为假，否则将出队元素写入传入引用变量。对于入队操作，由于 TBB::concurrentqueue 动态分配内存，不会遇到队列为满的问题，因此合理解决即可。受 TBB 中队列空/满判断的启发，可以设置一个 size 变量，其在进行入队和出队操作前先变化，每个入队操作前，都需要保证 size 变量增加后不超过队列容量，如果不满足的该条件，则在始终尝试入队操作直到队列产生空位。

**算法 1.1.** NonBlockingQueue-enqueue

**Algorithm 1.1** Enqueue

**Input:** nonblockqueue class member $arr[N], size, head, prehead, tail$
enqueue value $v$

1: **while** true **do**
2: $t\_size = size$(atomically)
3: **while** $t\_size >= N - 1$ **do**
4: $t\_size = size$(atomically)
5: **end while**
6: **if** ***CAS***$(size, t\_size, t\_size + 1)$ **then**
7: $allo\_pos =$ ***fetchadd***$(prehead, 1)$
8: $arr[allo\_pos\%N] = v$(atomically)
9: **while** !***CAS***$(head, allo\_pos, allo\_pos + 1)$ **do**
10: pass
11: **end while**
12: **break**
13: **end if**
14: **end while**

算法 1.1 代码中 ***fetchadd*** 为 C++ std::atomic 变量的一个原子级操作方式，将原子变量加上一个值，并返回原子变量增加前的值。

**算法 1.2.** NonBlockingQueue-dequeue

**Algorithm 1.2** Dequeue

**Input:** nonblockqueue class member $arr[N], size, head, prehead, tail$
dequeue write address $a$
**Output:** flag

1: Initial $flag = false$
2: $t\_size = size$(atomically)
3: **while** $t\_size > 0$ **do**
4: **if** ***CAS***$(size, t\_size, t\_size - 1)$ **then**
5: $allo\_pos =$ ***fetchadd***$(prehead, 1)$
6: $a = arr[allo\_pos\%N]$(atomically)
7: $flag = true$
8: **end if**
9: $t\_size = size$(atomically)
10: **end while**
11: **return** flag

算法 1.2 表示出队逻辑，实现上述的队列不空执行出队操作的行为。

同时需要解释的是，为何入队操作算法 1.1 第 3 行比较的为 N-1，由于 size 的限制只能保证队列满后不能再添加，而产生的一个情况是，队列满时，出队操

作使 size-1，此时一个入队进程触发，并在读出位置写入新数据，导致读出出错，因此需要间隔一位空位。

# 2 并发队列的正确性验证

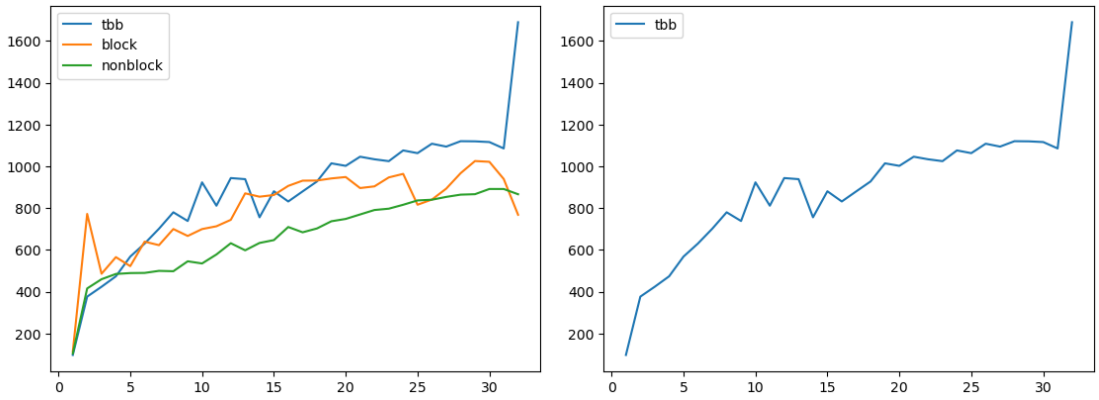
对于并发队列正确性的验证，参考了 TBB 文档中的叙述：”In a single-threaded program, a queue is a first-in first-out structure. But if multiple threads are pushing and popping concurrently, the definition of ”first” is uncertain. Use of concurrent_queue guarantees that if a thread pushes two values, and another thread pops those two values, they will be popped in the same order that they were pushed.”（在单线程程序中，队列是一种先进先出的结构。但是，如果多个线程同时进行推入和弹出操作，那么“先进”的定义就不确定了。使用 concurrent_queue 可以保证，如果一个线程推入两个值，而另一个线程弹出这两个值，那么它们将以与推入顺序相同的顺序被弹出。）

那么实际上，验证实验可以设计为，使用 N 个生产者线程，每个线程将对 N 求余等于线程 id（0~N-1) 的一定范围内的数按序加入队列，使用一个消费线程，进行出队操作，并在出队时，判断该数对 N 求余的组别，验证该组中的数是否单调上升即可。实验中的有锁队列和无锁队列均能通过该测试。

# 3 性能对比

下载并编译 tbb 源码，编译环境 Windows11 wsl2 Ubuntu20.04，编译工具 GNU gcc，cmake release 默认优化选项。

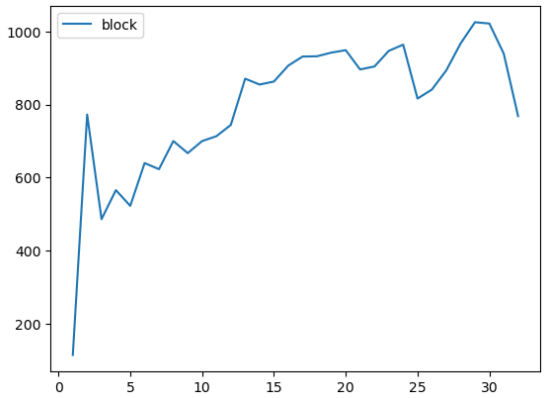
测试中使用 1~32 个线程操作并发队列，需要完成 $10^7$ 次操作，操作使用随机数随机选择，提前测试过随机数的随机效果能够达到出现 0，1 概率接近 1/2。由于基于数组的并发队列实现队列容量有限，当检测到队列为满时，修改输入操作为输出。但由于数组分配空间较大 $10^6$，因此该情况基本不会发生（概率上相当小）。



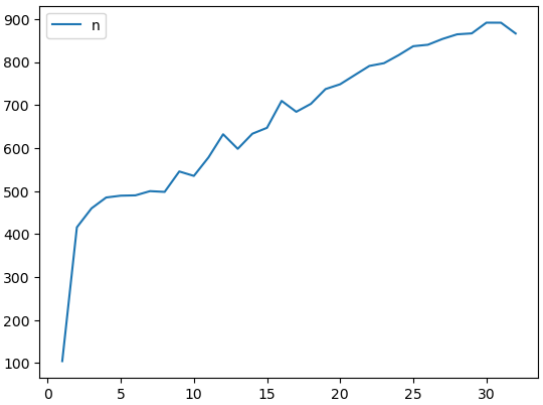

(a) 并发队列对比




(b) tbb 并发队列实现




(c) 有锁队列实现




(d) 无锁队列实现

图 3-1　并发队列实验结果. 图片纵坐标单位 (ms)

由图 3-1 可以看出，三种实现方式在线程数增加时，均出现了时间的增长，即由于线程间的竞争出现了性能的下降。其中 tbb 并发队列在 0~10 线程过程中经历一段接近线性的较为剧烈的上升，在 10~30 线程的阶段则为较为缓慢的性能下降，直到 32 线程时，再次出现了性能的剧烈下降。而实验中基于数组实现的有所队列和无锁队列，主要的特点是在线程数增加到 2 个时，性能大幅下降，5 线程以上进入一段接近线性的性能减小阶段。

有锁的并发队列实现在大部分情况下性能与 tbb 基本一致，无锁队列则能够在性能上略微由于 tbb 库中的并发队列，原因推测为 tbb 中并发队队列需要动态分配内存，同时，作为一款面向使用的程序，其中存在大量的错误处理逻辑，从而性能上较裸的并发队列实现有所下降理所应当。

# 4 调试记录

为了找到第一章中所提到的出队操作与入队操作相互干扰的问题，花了不少时间，直到测试正确性时将多线程生产单线程消费改为单线程生产单线程消费才终于在调试中发现了问题，同时使用命令行调试在多线程程序调试中发挥了关键作用，学到了很多。

# 参考文献

[1] MM. NBCAQ[EB/OL]. .
https://lrita.github.io/images/posts/datastructure/Simple_Fast_and_Practical_Non-Blocking_and_Blocking_Concurrent_Queue_Algorithms.pdf.

[2] INTEL. TBBCQC[EB/OL]. .
https://www.intel.com/content/www/us/en/docs/onetbb/developer-guide-api-reference/2022-0/concurrent-queue-classes.html.