并发算法与理论——并发 SkipList 设计与思考

王紫萁* 学号：201220154
ziqiwang@smail.nju.edu.cn
南京大学计算机科学与技术系

2023 年 12 月 19 日

摘要

作为《并发算法与理论》的课程设计，我们选取了 [1, Herlihy] 的经典并发 SkipList 算法进行复现，并对论文中的概念与算法进行了分析与思考。最后，讨论模糊搜索 HNSW 算法 [2, Malkov] 是否能够采用本文的思想。

1 引言

Skip List 是一种基于链表的数据结构，其在插入、删除、查找等操作上的时间复杂度均为 $O(\log n)$，并且相比于红黑树等其他数据结构，SkipList 的实现更加简单，且在并发环境下具有较好的性能。Skip List 的并发算法在原论文截止时的相关工作有 [3, Pugh] 的算法，该算法由于使用了反转指针因此实现非常复杂；Lea, D. 在前人工作上进行了改进并把算法应用到了 Java 的并发库中，该算法的性能达到了 sota，缺点时实现复杂，并且在某些特定的插入与删除交错执行的情况下，跳表的不变性（invariant）会暂时或永久被违背。

Herlihy 提出了一种基于锁的算法，该算法特点是：乐观锁（optimistic lock），延迟删除（lazy remove）。论文对该算法的实现非常详细，并且使用 Java 实现并比较了该算法与 Lea 的算法的性能，结果表明该算法的性能和 Lea 的算法相当，但是实现更加简单，有利于并发环境下的调试与维护。

本报告尝试使用 c++ 复现了朴素与并发跳表算法，并在 Add，Remove，Contains 三个操作总共 200,000 条指令上进行了压力测试，采用不同线程进行性能对比，在实验部分会给出具体的数据与分析。

*邮编：210046，系楼 412，Github：ziqiwww

2 背景与相关工作

2.1 跳表

跳表是一种基于链表的线性数据结构，也是一种典型的分层索引算法。在插入删除的平均时间复杂度为 $O(\log n)$。核心思想为：使用空间换取时间，一个元素在跳表中可能出现在多个层级，每个层级都是一个有序链表，最底层的链表包含所有元素，查找时从最上层开始查找，如果当前层级的下一个元素大于目标元素，则下降到下一层级继续查找，直到找到目标元素或者到达最底层。对于每个节点的最高层数通过参数 P 来控制，P 越大，跳表的层数越高，查找的效率越高，但是空间开销也越大。因此选择合适的 P 是也一个开放问题。

2.2 并发跳表

并发跳表的实现有很多种，其中最简单的一种是使用锁来保证并发安全，但是这种方法的性能并不好，因为在并发环境下，锁的开销很大。Herlihy 提出了一种基于乐观锁的算法，该算法的核心思想是：在插入和删除时，不立即修改数据结构，而是在查找时进行修改，如果发现数据结构被修改，则重新查找，直到成功为止。该算法的优点是实现简单，缺点是在高并发的情况下，会有大量的重试，导致性能下降。

3 算法描述

和普通的 wait-free 的链表算法相比，跳表的算法更加复杂，因为跳表的每个节点都有多个层级，所以相比单链表的并发插入，跳表只有等所有层级都被链入才被认为插入成功，该时刻也是 linearization

```cpp
template<typename T>
auto ConSkipList<T>::FindNode(
  T key,
  std::shared_ptr<ConSkipListNode<T>> *preds,
  std::shared_ptr<ConSkipListNode<T>> *succs) -> int {
  int layer = -1;
  std::shared_ptr<ConSkipListNode<T>> pred = LSentinel_;
  for (int i = this->max_layer_ - 1; i >= 0; --i) {
      std::shared_ptr<ConSkipListNode<T>> curr = pred->next_[i];
      // if we add RSentinel_ to the end of the list, curr is never nullptr
      while (curr != nullptr && curr->key_ < key) {
          pred = curr;
          curr = curr->next_[i];
      }
      if (layer == -1 && curr != nullptr && curr->key_ == key) {
          layer = i;
      }
      preds[i] = pred;
      succs[i] = curr;
  }
  return layer;
}
```

图 1: FindNode 函数

point。因此，跳表的并发插入算法需要保证所有层级都被链入，否则需要回滚。Herlihy 的算法使用了乐观锁，即在插入时不立即修改数据结构，而是在查找时进行修改，如果发现数据结构被修改，则重新查找，直到成功为止。

其次，跳表在插入之前的查找也很有讲究。如果像链表一样只记录一个前驱节点和后继节点，就只有该邻居的指针能被成功链入。因此，跳表的查找需要记录每一层的前驱节点和后继节点，这样才能保证所有层级都被成功链入。查找时原论文使用使用辅助函数`FindNode`（图1），该函数使用`preds`和`succs`两个数组来记录每一层的前驱节点和后继节点，并且不做任何前驱后继的 validation，从整个实验过程来看，该辅助函数被频繁调用，因此不做额外检查可以提高性能。

3.1 Add 函数

Add 主要被分为三个过程：查找目标位置并验证，加锁并插入节点，解锁并返回。使用 FindNode 查找插入位置，由于 FindNode 并不加锁，因此在函数返回后为了确保正确需要进行两步检查：[1]. 待插入节点在跳表中是否存在；[2]. 如果存在，是否被 mark。在确定未被 mark 并且存在后，该线程需要等到 key 完全被链入跳表再返回 false。

当 key 不在跳表中时，对 preds 和 succs 中暂存的每一层的前驱和后继节点，从下而上扫描到准备给新加入节点分配的高度值 h，并依次给每一层的 pred 加锁。或者找到最低的不满足条件的层，这里的条件是指该层的 pred 和 succ 都没有被 mark 且 pred 的后继的确是 succ，即没有其他线程在 pred 和 succ 之间插入。

如果在对 h 之下的每一层检查都成立，该节点可以被安全的加入到跳表中，链入后设置`fully_linked`为 true，否则解锁并从头开始尝试。

有关 Add 和 Remove 的具体实现见附录。

3.2 Remove 函数

Remove 与 Add 类似，首先调用 FindNode 看看目标节点是否存在，然后依次检查以下条件是否成立：[1]. 当前节点是否需要被删除，该判断为了检查上一次失败的删除操作是否有需要删除的节点；或者：[2]. 当前节点是否被完全链入跳表（`fully_linked`）；[3]. 节点的最高层是否和从上到下寻找的最高层一致；[4]. 节点是否还没被 mark。

当上述条件满足时，节点的`marked`被设置为 true，并执行与`Add`对 preds 和 succs 数组从下而

上做 validation 同样的检查。检查成功，每一层的 pred 直接连接到 succ 即可，`shared_ptr`会自动回收被删除的节点。

3.3 Contains

contains 直接调用`FindNode`，并检查找到的节点是否未被 marked 并且是否完全链入跳表。

4 算法正确性

4.1 Linearizability

Linearizability 的正确性由 lazy list [3] 的正确性保证，即对于算法的每一个可达状态，节点的前驱和后继关系在之后的任何过程中都能得到保证，直到某个节点被移除。在移除过程中，被物理移除的节点必须被标记为 marked，并且由于`fully_linked`只会从`false`标记为`true`，于是一个节点只可以通过标记 marked 来移除，而设置 marked 的指令是 linearization point。

对节点的加入操作只有在`fully_linked`被标记为 true 时才算做完。在此之前，Contains 和 Remove 即使真的找到该节点，也不会认为节点被完全加入到跳表，因为新节点的链接关系还未被完全设置。于是`fully_linked`也是 linearization point。

最后需要说明的是，某个节点的`marked`在程序执行的某个时刻被发现标记为 true，那么该 node 的 key 一定不在跳表中。一方面，被标记的节点在检查之前一定在跳表中，因为：[1]. 只有`marked`为 false 的节点才能被加入 list；[2].key 在跳表中唯一，不可能有 unmarked 的节点具有相同的 key。另一方面，如果看到节点被标记，说明在该线程之前一定有某个线程正在负责该节点的删除操作，只不过还未把节点从当前层摘下从而被当前线程发现。对于这种情况，当负责摘除的线程完成删除之后，只有当前线程的中的某个临时指针指向该节点，临时指针释放后，该节点被自动物理释放。

4.2 不变性保证

在加入和删除时之前都会对前驱和后继的有效性做检查，检查的同时会给每一层加锁。加入时从上到下将节点链入，删除时从下到上将节点摘除。假设在某一时刻两个线程希望同时加入一个相同的 key，在 validation 时，两个线程在竞争第 0 层的锁时只有一个线程能够成功，而另一线程在第 0 层锁上等待。等到竞争成功的线程完全持有每一层的锁并物理加入节点后，等待的线程被唤醒并发现记录的前驱和后继不匹配，于是重新 `FindNode`，发现节点已经被加入，返回 false。

删除节点时同理。

对于朴素跳表算法，由于不加锁，在两个线程加入相同 key 的节点时，最好的情况是可能导致具有重复 key 的节点被加入，删除时只删除其中一个，破坏了不变性。最坏的情况是每一层的前驱指针分别随机指向具有相同 key 的不同节点，破坏了链表的结构。

4.3 deadlock-free & wait-free

由于`FindNode`按照从小到大的顺序查找目标位置，算法总是优先操作较小的 key，较大 key 在较低层等待。因此能够保证到来的每个 key 都能按顺序被操作。不会出现死锁。同理 Contains 操作也是 wait-free 的，首先他不持有任何锁，并且只对跳表做一次扫描。

5 实验结果

本次实验机器的 CPU 是 Apple M1 (8 cores, 8 threads)，测试一执行 200,000 条 Add 指令，测试二总共执行 200,000 条指令操作，其中 90% Add 指令，7.5% 的 Remove 指令以及 2.5% 的 Contains 指令。

表 1: 测试 1：200,000 条 Add 指令

线程数	指令数	时间 (ms)	每秒指令数
1	200,000	304,560	656
2	200,000	196,218	1,019
4	200,000	115,205	1,735
8	200,000	96,267	2,076

实验结果如表1和表2所示，可以看到随着线程数的增加，每秒指令数也随之增加，但是增加的幅度逐渐减小。由于生成 Remove 和 Contains 指令的 key 在 0-200,000 的`int`型数据中采样，所以执行中可能由于节点未被加入而提前返回，并且 contains 只包含简单的查找与谓词逻辑，因此吞吐量会稍高于测试一。

表 2: 测试 2：200,000 条指令，其中 90% Add 指令，7.5% 的 Remove 指令以及 2.5% 的 Contains 指令

线程数	指令数	时间 (ms)	每秒指令数
1	200,000	279,979	714
2	200,000	167,576	1,193
4	200,000	94,557	2,115
8	200,000	79,929	2,501

6 讨论

6.1 模糊搜索 HNSW 算法

HNSW（Hierarchical Navigable Small World）[2] 是一种基于图的模糊搜索算法，其核心思想是：在高维空间中，每个节点都有一些邻居，这些邻居的距离都比较近，但是不一定是最近的。这样的邻居被称为“小世界”。该索引已经作为向量数据库的基础索引被广泛使用，其性能已经超过了 sota 的 KD-Tree 算法。HNSW 索引第 0 层是一个关于数据集邻接关系的图，以上每一层都是下一层的子图。对于一个向量搜索请求，从高层向下搜索，与跳表不同的是，由于采用基于最近邻的模糊搜索，HNSW 需要从最高层一直找到最底层。

和跳表不同的是，HNSW 对 Remove 的支持并不友好，图中节点的删除可能会导致图不连通，因此只能重新构建索引。因此我们在此仅讨论针对 HNSW 的 Add 和 Contains 的并发化的一些可能思路。

在插入节点之前，节点被赋予`top_layer`（与跳表含义相同）。搜索时采用不加锁的贪心搜索，从当前层的起点开始找到 k 个最近邻（k 为当前层的超参数）。如果当前层需要与邻居节点加边（$layer \leq top\_layer$），那么将所有邻居加锁，并把当前节点加入到邻居的邻居列表中，解锁并返回，这里可能需要检查邻居节点的邻居数是否与进入临界区前记录的邻居数是否相同，防止在相邻位置有新加入的节点。如果有新加入的节点，可能需要重新执行当前层的搜索（不需要从最高层开始查找）。

Contains 操作被重新定义为找到当前查询的 k 个最近邻，可以不加锁地从高到低搜索。

6.2 跳表的优化

`FindNode`函数被三个函数都会被调用一次，因此该操作具有很大的优化空间。比如可以通过记录跳表当前具有的最高层数，从而不必每次从最高层开始扫描，由于每一层节点数相比下一层指数级降低，因此并没有很多节点能够拥有跳表允许的最高层，当层数增加时该现象更加明显。另外，可以通过记录每一层的最后一个节点，从而在查找时可以从上一次查找的位置开始，而不必从头开始扫描。

7 总结

本次课程设计复现了 Herlihy 的并发跳表算法，并对算法的正确性进行了分析。实验结果表明该算法在高并发环境下有很好的性能提升，且实现简单，易于调试与维护。最后，讨论了 HNSW 算法的并发化可能的思路，总结了跳表的一些优化空间。

参考文献

[1] M. Herlihy, Y. Lev, V. Luchangco, and N. Shavit, “A Simple Optimistic Skiplist Algorithm,” in *Structural Information and Communication Complexity*, G. Prencipe and S. Zaks, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, vol. 4474, pp. 124–138, series Title: Lecture Notes in Computer Science. [Online]. Available: http://link.springer.com/10.1007/978-3-540-72951-8_11

[2] Y. A. Malkov and D. A. Yashunin, “Efficient and Robust Approximate Nearest Neighbor Search Using Hierarchical Navigable Small World Graphs,” *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 42, no. 4, pp. 824–836, Apr. 2020. [Online]. Available: https://ieeexplore.ieee.org/document/8594636/

[3] W. Pugh, “Skip lists: a probabilistic alternative to balanced trees,” *Communications of the ACM*, vol. 33, no. 6, pp. 668–676, Jun. 1990. [Online]. Available: https://dl.acm.org/doi/10.1145/78973.78977

A 附录

A.1 Add 函数

```cpp
template<typename T>
auto ConSkipList<T>::Add(T key) -> bool {
  std::shared_ptr<ConSkipListNode<T>> preds[this->max_layer_];
  std::shared_ptr<ConSkipListNode<T>> succs[this->max_layer_];
  while (true) {
      int layer_check = FindNode(key, preds, succs);
      if (layer_check != -1) {
          std::shared_ptr<ConSkipListNode<T>> nodeFound = succs[layer_check];
          if (!nodeFound->marked_) {
              // wait until node is fully linked, i.e. node is visible.
              while (!nodeFound->fully_linked_) {}
              return false;
          }
          continue;
      }
      int highestLocked = -1;
      int top_layer = this->RandomLayer();
      std::shared_ptr<ConSkipListNode<T>> pred, succ, prevPred = nullptr;
      bool valid = true;
      for (int layer = 0; valid && layer <= top_layer; ++layer) {
          pred = preds[layer];
          succ = succs[layer];
          // check if pred and succ are still valid
          if (pred != prevPred) {
              pred->lock_.lock();
              highestLocked = layer;
              prevPred = pred;
          }
          valid = !pred->marked_ && !succ->marked_ && pred->next_[layer] == succ;
      }
      if (!valid) {
          for (int layer = 0; layer <= highestLocked; ++layer) {
              preds[layer]->lock_.unlock();
          }
          continue;
      }
      std::shared_ptr<ConSkipListNode<T>> newNode = std::make_shared<ConSkipListNode<T>>(
          key, top_layer);
      for (int layer = 0; layer <= newNode->top_layer_; ++layer) {
          newNode->next_[layer] = succs[layer];
          preds[layer]->next_[layer] = newNode;
      }
      // linearization point
      newNode->fully_linked_ = true;
      for (int layer = 0; layer <= highestLocked; ++layer) {
          preds[layer]->lock_.unlock();
      }
      return true;
  }
}
```

A.2 Remove 函数

```
template<typename T>
auto ConSkipList<T>::Remove(T key) -> bool {
    std::shared_ptr<ConSkipListNode<T>> victim = nullptr;
    bool isMarked = false;
    std::shared_ptr<ConSkipListNode<T>> preds[this->max_layer_];
    std::shared_ptr<ConSkipListNode<T>> succs[this->max_layer_];
    while (true) {
        int layer_check = FindNode(key, preds, succs);
        if (layer_check != -1) {
            victim = succs[layer_check];
        }
        if (isMarked ||
            (layer_check != -1 &&
              victim->fully_linked_ && victim->top_layer_ == layer_check && !victim->
                  marked_)) {
            if (!isMarked) {
                victim->lock_.lock();
                if (victim->marked_) {
                    victim->lock_.unlock();
                    return false;
                }
                victim->marked_ = true;
                isMarked = true;
            }
            int highestLocked = -1;
            std::shared_ptr<ConSkipListNode<T>> pred, succ, prevPred = nullptr;
            bool valid = true;
            for (int layer = 0; valid && layer <= victim->top_layer_; ++layer) {
                pred = preds[layer];
                succ = succs[layer];
                if (pred != prevPred) {
                    pred->lock_.lock();
                    highestLocked = layer;
                    prevPred = pred;
                }
                valid = !pred->marked_ && pred->next_[layer] == succ;
            }
            if (!valid) {
                for (int layer = 0; layer <= highestLocked; ++layer) {
                    preds[layer]->lock_.unlock();
                }
                continue;
            }
            for (int layer = victim->top_layer_; layer >= 0; --layer) {
                preds[layer]->next_[layer] = victim->next_[layer];
            }
            victim->lock_.unlock();
            for (int layer = 0; layer <= highestLocked; ++layer) {
                preds[layer]->lock_.unlock();
            }
            return true;
        } else {
            return false;
        }
```

```
    }
}
```

A.3 Contains 函数

```
template<typename T>
auto ConSkipList<T>::Contains(T key) -> bool {
    std::shared_ptr<ConSkipListNode<T>> preds[this->max_layer_];
    std::shared_ptr<ConSkipListNode<T>> succs[this->max_layer_];
    int layer = FindNode(key, preds, succs);
    return (layer != -1 && succs[layer]->fully_linked_ && !succs[layer]->marked_);
}
```