布谷鸟哈希作业

> 这个作业让我切身感受到，并行并不是一定就快，并行的能够加速计算的条件有很多，譬如：
>
> - 机器的环境、配置、条件
> - 程序的锁的粒度、加锁的位置、加锁的数量
> - 数据量的大小，并行程序的时间

一、死循环检测和ReHash

关于陷入死循环之后的 Rehash 的内容，我是这么设计的：

- 用一个变量统计while的次数，如果超过了次数，就需要rehash

```
        std::unique_lock<std::mutex> lck(mtx);

        // two place for one certain key has been occupied, need evict others
        KeyType evicted = key;

        // wich is 0 or 1, 0 means evict key in hash1, 1 means evict key in hash2
        int which = 0;

        // first evict key in hash1
        int idx = hash1(evicted);

        int pre_pos = -1;

        int whileTimes = 0;
        // evict key in hash1
        while(T[idx] != 0) {
            swap(&evicted, &T[idx]);

            pre_pos = idx;
            which = 1 - which;
            idx = (which == 0)?hash1(evicted):hash2(evicted);

            whileTimes++;
            if (whileTimes > Limit) {
                printf("dead loop, rehash ing\n");
                // 加锁mutex
                // std::unique_lock<std::mutex> lck(mtx);
                rehash(evicted);
                // 解锁mutex
                // lck.unlock();
                return;
            }
        }
```

```
34
35              T[idx] = evicted;
36          }
```

- 由于我用到的Hash函数，是和Canvas 上的 第十章Cuckoo hash.pdf 的10.3-10.5节一样的，也就是说根**T数组的容量一样**，我在ReHash的时候执行下面的操作：
- 扩容之前，保存所有的旧的元素，然后把Vector容器清空
- 数组扩容（也就是Vector的Resize）
- 然后把旧的元素全部执行一遍重新插入。
- 此外ReHash也涉及到加锁，因为改了共享变量

```
1   // ========================================
2   void Cuckoo::rehash(int evicted) {
3       // TODO
4       std::vector <KeyType> oldT;
5
6       // 从T中拷贝数据到oldT
7       oldT.assign(T.begin(), T.end());
8
9       // 重新分配空间
10      this->cuckooSize *= 4;
11
12      // 清空T
13      T.clear();
14
15      // resizeT
16      T.resize(this->cuckooSize, 0);
17
18      // 重新插入数据
19      for (int i = 0; i < oldT.size(); i++) {
20          if (oldT[i] != 0) {
21              put(oldT[i]);
22          }
23      }
24
25      // 重新插入evicted
26      put(evicted);
27  }
```

二、实验结果

- 实验结果和理想的结果差距比较大，当然也在我的预估之内，毕竟并行的程序收到的影响很多。
- 我们分为PUT和GET两个来看实验结果
- 所有的实验结果参考本目录下面的**CSV文件**，里面包含了输出的时间、加速比。

1）put操作

PUT的操作的代码最核心的区域涉及到加锁，如下图所示这就导致，哪怕是多线程的执行，几乎等同于单线程。

```
        std::unique_lock<std::mutex> lck(mtx);
        KeyType evicted = key;
        int which = 0;
        int idx = hash1(evicted);
        int pre_pos = -1;
        int whileTimes = 0;
        // evict key in hash1
        while(T[idx] != 0) {
            swap(&evicted, &T[idx]);
            pre_pos = idx;
            which = 1 - which;
            idx = (which == 0)?hash1(evicted):hash2(evicted);
            whileTimes++;
            if (whileTimes > Limit) {
                printf("dead loop, rehash ing\n");
                rehash(evicted);
                return;
            }
        }
        T[idx] = evicted;
```

- 先来看看实验结果，这个是数据量50000的时候的

```
threadNum,insertSize,time,rate
1,50000,5840786,0.995964
2,50000,5523192,0.941809
3,50000,5258677,0.896704
4,50000,5201388,0.886935
5,50000,5366976,0.915171
6,50000,5262064,0.897281
7,50000,5276552,0.899752
8,50000,5284380,0.901087
9,50000,5403510,0.921401
10,50000,5357557,0.913565
11,50000,5303994,0.904431
12,50000,5398018,0.920464
13,50000,5363005,0.914494
14,50000,5336380,0.909954
15,50000,5263718,0.897563
16,50000,5282132,0.900703
17,50000,5346729,0.911718
18,50000,5354375,0.913022
19,50000,5364288,0.914712
20,50000,5379309,0.917274
```

```
21,50000,5364208,0.914699
22,50000,5429666,0.925861
23,50000,5381559,0.917657
24,50000,5378466,0.91713
25,50000,5394187,0.919811
26,50000,5358596,0.913742
27,50000,5309309,0.905337
28,50000,5376395,0.916777
29,50000,5445778,0.928608
30,50000,5474283,0.933469
31,50000,5495286,0.93705
32,50000,5408820,0.922306
```

- 这个是数据量10000的时候的

```
threadNum,insertSize,time,rate
1,10000,4854270,1.00326
2,10000,4796547,0.991334
3,10000,4819067,0.995989
4,10000,4832586,0.998783
5,10000,4913565,1.01552
6,10000,4904224,1.01359
7,10000,4913455,1.0155
8,10000,4958372,1.02478
9,10000,4957093,1.02452
10,10000,4927950,1.01849
11,10000,4892651,1.0112
12,10000,4946859,1.0224
13,10000,4966211,1.0264
14,10000,4938718,1.02072
15,10000,4924393,1.01776
16,10000,4974742,1.02816
17,10000,5006768,1.03478
18,10000,4974268,1.02807
19,10000,4987619,1.03082
20,10000,4974479,1.02811
21,10000,4995195,1.03239
22,10000,4906828,1.01413
23,10000,5016143,1.03672
24,10000,5037351,1.0411
25,10000,5030797,1.03975
26,10000,4964841,1.02612
27,10000,5028242,1.03922
28,10000,5049367,1.04359
29,10000,5034333,1.04048
30,10000,5059466,1.04567
31,10000,5057799,1.04533
32,10000,5033247,1.04025
```

- 基本上就可以看到，和1大差不差，等同于串行的效果，并行并没什么加速可以说，因为锁的粒度太大了

- **于是乎，我就想锁粒度大，我可以减小锁的粒度啊！**但是事与愿违，当我把原来的一个锁变成了三个共享变量操作附近的单独的锁的时候，发现**时间反而变成十倍**，这说明加锁的数量对于时间也有影响，所以不能凭借锁的粒度来简单的分析问题。毕竟加锁、解锁这个过程本身也是有开销的！不能忽略
- 总之就是，**由于PUT的函数并不适合做并行，而是写操作本身就是一个需要注意隔离的事情，所以并行效果约等于串行，效果不好**

2）get操作

> get是读取操作，因为读操作不需涉及到锁，所以我本来也认为会更快的。但是实际上并不是。我发现并行根数据量的大小也有关系

下面的数据是10000数据量的时候的

```
threadNum,insertSize,time,rate
1,10000,163541,1.48124
2,10000,163781,1.48342
3,10000,157684,1.42819
4,10000,136182,1.23344
5,10000,157835,1.42956
6,10000,164856,1.49315
7,10000,178411,1.61592
8,10000,193182,1.74971
9,10000,206166,1.86731
10,10000,219299,1.98626
11,10000,244968,2.21875
12,10000,248346,2.24935
13,10000,271747,2.4613
14,10000,275919,2.49909
15,10000,304822,2.76087
16,10000,323877,2.93346
17,10000,341218,3.09052
18,10000,354166,3.20779
19,10000,399091,3.61469
20,10000,405575,3.67342
21,10000,533062,4.82811
22,10000,444210,4.02335
23,10000,435419,3.94373
24,10000,484489,4.38817
25,10000,514234,4.65758
26,10000,518541,4.69659
27,10000,544427,4.93105
28,10000,562734,5.09686
29,10000,578385,5.23861
30,10000,586153,5.30897
31,10000,574416,5.20267
32,10000,610783,5.53205

```

下面的数据是50000数据量的时候的

```
threadNum,insertSize,time,rate
```

```
1,50000,572070,1.07588
2,50000,381455,0.717394
3,50000,315953,0.594206
4,50000,261656,0.492091
5,50000,284893,0.535792
6,50000,271138,0.509923
7,50000,278249,0.523297
8,50000,291809,0.548799
9,50000,261776,0.492316
10,50000,309518,0.582104
11,50000,285666,0.537246
12,50000,299799,0.563826
13,50000,304783,0.573199
14,50000,301159,0.566383
15,50000,312838,0.588348
16,50000,323028,0.607512
17,50000,329328,0.61936
18,50000,349169,0.656675
19,50000,381658,0.717776
20,50000,396052,0.744846
21,50000,410421,0.77187
22,50000,439413,0.826395
23,50000,461106,0.867192
24,50000,484815,0.911781
25,50000,526435,0.990055
26,50000,508104,0.95558
27,50000,509020,0.957303
28,50000,551533,1.03726
29,50000,541031,1.01751
30,50000,552400,1.03889
31,50000,558953,1.05121
32,50000,610346,1.14786

```

我们来仔细分析一下：

- 首先，线程数量为1的时候，是怎么回事呢，我运行的时候额外创建了一个线程，然后等他执行完毕，这显然比单线程要开销多出来了创建线程，所以加速比大于1（开销更大，也没有毛病）
- 然后问题就出来了，数据量为10000的时候，哪怕是双线程、三线程，基本上也没有看出来任何的加速，反而速度更慢，这是为什么呢，我认为有下面的原因：
    - 并行计算里面，拆分了N线程，每个线程只需要做原来1/N的工作，也就是说，节约了(1-1/N)T的时间
    - 但是并行计算里面，创建线程也是有开销的！创建线程、回收线程（等待join）都是时间开销，这个我暂时不太会计算
    - 前者的节约的时间和任务的复杂度有关，假如数据量越大，也就是T越大（消耗的时间越多）多线程节约的时间就越多，反之如果任务本身就很简单的话，可能创建线程的开销反而带来的更多，导致整个时间延迟
- 所以我发现数据量小的时候，是**不适合做并行计算的**，因为本身就很快，并行计算节约的时间也不多，反而小于创建线程，回收线程的开销，导致总体加速并不明显

- 数据量比较大的时候50000，可以看到加速比还是有的，大概线程的数量在9的时候，节约的时间最明显（我电脑是20核心的mac，有点怪，理论是应该20核心都跑满的话那就是20线程的时候时间比最小，但是可能根操作系统调度线程有关，所以和实际的也有差距）
- 但是当线程数量太多的时候，显然上下文切换带来的时间就越来越多了，反而效果不好，也可以看到加速比在变大，这也是符合理论的。

三、实验总结

并行不一定比串行快，总结一下并行的影响因素：

- 跟机器有关，比如一个单核心的机器，并行效果肯定比不上多核心的
- 跟操作系统的调度有关，核心数量再多，一核有难，八核围观照样没有用。
- 跟并行程度代码有关，假如一个并行的函数写的里面全是锁，那就根白写了一样，等于串行
- 跟并行加锁的数量有关，假设我放小粒度的锁，但是带来反复的加锁、解锁，反而也会带来更多的时间开销
- 跟并行任务的执行时间有关，假设我一个很简单的任务（本来要T时间完成），拆分为N个线程，节约的时间是 $T - T/N$ 但是创建线程回收线程都是有开销的，所以两者权衡之后，再做出是否并行的选择

所以，并行还是串行，需要实践解决，观察到效果确实存在、明显，使用多线程并行就很好，但是如果发现效果反而不如人意，不如使用串行的策略。