数据结构与算法 I 作业 20

2019201409 于倬浩

2020 年 12 月 21 日

设计外存选择算法

考虑改进最坏线性时间复杂度的 `Select` 算法。首先，假设共有 `N` 个要处理的元素，内存可以容纳 `M` 个元素，每块存储块可容纳 `B` 个元素。

和原始算法类似，假设我们当前处理 `Select(L, R, k)`，先把这 $\Theta(N)$ 个元素分成 $\frac{N}{M}$ 块，每次读入 M 个元素至内存中，使用内存中的中位数算法以 $\Theta(M)$ 的时间复杂度找到中位数并存下这些中位数，最后使用内存中位数算法算出这 $\frac{N}{M}$ 个中位数的中位数 P。接下来进行 `partition` 操作，每次读入 B 个元素至内存中，并维护三个缓冲块。块 `A` 维护了小于等于 P 的元素，块 B 维护大于 P 的元素，块 C 维护从外存中读取的当前要处理的 B 个元素。当前两个块有一个填满时，写入外存。写入策略和内存的 `partition` 算法一致，因此每次块满后，最好情况只需要一次写入，最坏情况还需要加一次整块交换。最终，当所有块已经处理完毕，只需将缓冲区中剩余的元素写入外存即可。因此，每次递归处理 `Select(L,R,k)`，需要 $\Theta(\frac{N}{B})$ 次 I/O 操作。

根据朴素算法的时间复杂度分析，我们只需保证每次分的块 $\frac{N}{M}$ 大于 5 即可保证总时间复杂度为线性，而由于元素的访问次数和时间复杂度又成线性关系，因此可以保证总的元素读写次数为 $\Theta(n)$，且由于保证了连续 I/O，我们总共对外存的 I/O 次数也是 $\Theta(\frac{n}{B})$。

然而，如果 $\frac{N}{M}$ 过大，即要处理的元素数目远大于内存可以容纳的数量，就不再能使用内存的中位数算法算出 $\frac{N}{M}$ 个数字的的中位数。因此，还需要按照朴素算法，五个数分一组，然后之后进行和上述描述的算法相同的操作。这样做的优点是不再需要关注内存大小，但是缺点是对外存的 I/O 次数增加了一倍（每次算出中位数后，都需要写入外存）, 但仍能达到要求的 $\Theta(\frac{n}{B})$ 次 I/O。

Cache-Oblivious 算法

- a. 证明 Quicksort 不是一个 Cache-Oblivious 算法

  `QuickSort` 算法的主要时间消耗、数据读写主要集中于 `partition` 操作，因此我们只需考虑 `partition` 操作带来的影响。`partition` 操作实际上是对要分区的区间，将大于主元的元素放在尾部，将小于主元的元素放在中部，不断迭代。虽然最终 `partition` 操作的区间将被不断细分，直到 cache 中可以存下整个当前操作的区间，但是 `partition` 操作本身并不是递归算法，即使当前处理的区间大于 cache，依然会使用同样的方式处理。

  举例而言，假设当前 `partition` 的区间长度为 $\Theta(N)$ 级别，不能放入 cache 中，该算法就会对区间内的数据不断进行无缓冲的读写，因此最坏情况下，例如给出一个极大元素、极小元素交替的数组，那么每次都需要往数组的中部/尾部交替写入，因此需要 $\Theta(N)$ 次 I/O，高于 $\Theta(N/B)$。

  因此，最坏情况下，I/O 的操作数至少可以达到 $O(N + \frac{N}{B} log_{\frac{M}{B}} \frac{N}{B})$，高于 $\frac{N}{B} log_{\frac{M}{B}} \frac{N}{B}$。

- b. 考虑如下算法，证明其在不知道 B 和 M 的情况下，I/O 复杂度可达到 $O(\frac{N}{B} log_{\frac{M}{B}} \frac{N}{B})$ 个 I/O

  该算法整体上是一个递归的过程，直到要处理的区间可以完全被放到 cache 中，接下来的读写均在 cache 中进行，因此只需考虑大于 cache 大小的区间，在计算的过程中是否可以保证连续 I/O。

  观察算法流程，发现实际上只有前两步 `partition` 和 `distribution` 会对数据产生实际的操作。如果可以保证 `partition` 和 `distribution` 的读写连续性，即使不知道 B 依然可以做到单次调用产生的 I/O 次数为 $O(N/B)$。

  因此，如果我们保证了 `distribution` 的读写连续性，每层递归的区间长度都是上一层的 $\frac{1}{\sqrt{N}}$，而当递归深度到 $O(log_{\frac{M}{B}} \frac{N}{B})$ 时，显然此时的区间已经足够小，可以直接放在 cache 中了，因此更深层次的递归不会带来外存 I/O 操作数目的增加。因此，每一层都需要对整个区间进行读写，而最多有 $O(log_{\frac{M}{B}} \frac{N}{B})$ 层，因此该算法的 I/O 操作数为 $O(\frac{N}{B} log_{\frac{M}{B}} \frac{N}{B})$。

- c. 给出符合上述要求的 distribution 过程伪代码

```
void distribute(int L, int R, int P) {
    // 将编号为 [L, R] 范围内的子数组，插进从 P 开始的若干个桶
    if(L == R) {
        Load Array[L] into RAM;
        Initialize RAM buffer Buf[];
        for(auto i: Array[L]) { //连续访问第 i 个子数组，
            if(i < B[P].pivot) Add i to Buf[];
        }
        //使用一次外存读写将整个数组合并起来
        Concat(B[P], Buf);
```

```
            // 如果大小不满足要求，split 操作将一个桶分裂成两个。
        if(B[P].size > 2 * sqrt(N)) split(B[P]);
        // split() 使用线性中位数算法和 partition 算法分割区间，
        // 在分裂后，重新计算两个桶 pivot，
        // 较大的桶 pivot 为 B[P].pivot，较小的为找到的中位数。
    }
    else {
        int M = (L + R) / 2;
        //递归处理，将前一半/后一半的子数组中较大/较小元素分成两部分
        distribute(L, M, P);
        distribute(M + 1, R, P);
        distribute(L, M, P + (R - L + 1) / 2);
        distribute(M + 1, R, P + (R - L + 1) / 2);
    }

}
```

这道题有阅读参考文献，自己真的是想不到这么妙的东西，但是发现参考文献上的伪代码并没有给出关键细节（如何保证连续读写），于是补上了文献中缺失的部分。

参考资料：Cache-Oblivious Algorithms. Harald Prokop. MIT