CS 229r: Algorithms for Big Data Fall 2013

Lecture 21 — November 12, 2013

Prof. Jelani Nelson *Scribe: Perry Green*

1 Introduction

This week we begin a new topic - the external memory model, also called disk access model (DAM), or the I/O model [AV88].

1.1 Setup

All of our data is stored on some external storage (disk), which is connected to a much smaller memory (RAM), of size M. The disk is divided into sectors, so that it is quicker to read from sequential sectors than from many arbitrarily located sectors. As a result, all data on disk is stored in blocks of size B, and when data needs to be moved from disk to memory, the entire block is moved.

When you need some data, you first check if it is already in memory. If it is, then you can use it immediately. If it isn't, then the block is moved from disk to memory. If memory is full, then one of the $\frac{M}{B}$ blocks already in memory must be evicted. **The cost of an algorithm in the DAM model is the number of reads/writes to and from disk**. Operations over the content of memory do not count towards the cost.

1.2 Motivation

Some numbers from online (possibly out of date, but should give some idea):

1. Sequential reads from disk: $\sim 2 - 300$ times slower than RAM
2. Random reads from disk: $\sim 10^6$ times slower than RAM

So, suppose you are operating over a large amount of data, such that it can fit onto a single disk, but not within memory. As long as the work being done within memory is not extremely computationally expensive, this is a appropriate model, and can predict real life performance of algorithms.

2 Algorithms

Some standard (or somewhat modified) algorithms and their respective costs within DAM.

2.1 Scanning an array

Cost: $O(\frac{N}{B})$. Since the array is of length N, it is stored in no more than $\frac{N}{B}+1$ blocks, and so takes $O(\frac{N}{B})$ IOs to move into memory.

2.2 Multiplying XY, two $N \times N$ matrices

Store each matrix by blocks of size $\sqrt{\frac{M}{2}} \times \sqrt{\frac{M}{2}}$. Then you can store one block from X and one block from Y in memory at all times.

Moving each block into memory takes $O(\frac{M}{B})$ IOs, and the standard matrix multiplication algorithm is cubic, so $O(\frac{N}{\sqrt{M}}^3)$ block×block multiplications are required. So, the whole algorithm is $O(\frac{N^3}{B\sqrt{M}})$

3 Linked Lists

We want our linked list to support:

1. Insertions
2. Deletions
3. Traversal of k items

DAM implementation: Each node is instead a block-size array of elements, and a pointed to the next node (block). Maintain the invariant that each block (except possibly the last) is at least half full. To maintain invariants:

1. Insertions: Add to the correct node if possible (rearranging the internal elements within memory if needed). If the node is already full, allocate a new node and move half of the elements there.
2. Deletions: If the node is greater than half full, then you're done. If it is exactly half full, and it's the last node, then that is fine. If it is exactly half full and not the last node, then it must have some neighbor which is at least half full. If it is more than half full, then take elements from there. If it is also exactly half full, the combine the two nodes.

4 Binary Search

Naive solution: Do the normal $\log(N)$ solution. But, at some point, the window of possible locations will be of size $\leq B$, and you can just look within memory without any additional IOs. So, the total cost is $\log(N) - \log(B) = \log(\frac{N}{B})$

But, it should be possible to do better. In the comparison model, it should take $\log(N)$ bits of information to find the location of an item within a sorted list of size N. And, we learn $\log B$ bits of information in each IO, because I learn the comparison between the desired item and each item within the block. So, it should be possible to do this in $\frac{\log(N)}{\log(B)} = \log_B(N)$ IOs. To achieve this performance, we need some new data structures.

4.1 (a,b)-Trees / B-Trees

Every internal node of an (a,b)-tree has between a and b (inclusive) children. Leaves each have between a and b elements (unless there is only one node). To insert into an (a,b) tree, navigate as you would a BST (except possibly choosing between more than 2 intervals at each node) until you get to a leaf. Then, insert the value at that leaf. If this causes the leaf to have more than b elements, then split the leaf into two at the middle, and add the dividing element to the parent, recursing upwards if that node is also full, or making a new node if needed. A B-tree is just a $(\frac{B}{2}, B)$-tree and was invented in [BM72]. For more information, diagrams, etc. see:

`http://en.wikipedia.org/wiki/B_tree`

`http://en.wikipedia.org/wiki/2-3-4_tree`

Observe that new nodes are only ever added to the tree as roots, and so that the path from the root to any leaf is always the same. So, (a,b) trees are always balanced. For a B-tree, there are at most $\frac{N}{B/2} = \frac{2N}{B}$ leafs. The branching factor is always at least $\frac{B}{2}$. So, the height of the tree is always at most $\log_{B/2}(\frac{2N}{B}) = O(\log_B N)$. So, using B-trees, binary search can be solved in $O(\log_B N)$ IOs: just traverse the tree starting at the root.

But, there is also a downside to using B-trees. What if we just kept a big log of every time we add/remove an element. This would be $O(\frac{N}{B})$ lookup, since you would need to traverse all the elements to find one. But, it would also be only $O(\frac{1}{B})$ amortized insertion, since only $\frac{1}{B}$ of the time do you need to allocate a new block to insert an element - otherwise you can just add it to a block already in memory. So, there is a lot of room to improve on insertion time, even if it costs a little in query time.

4.2 Buffer Repository Trees

These were invented in [BGVW00]. A buffer repository tree is an augmented (2,4)-tree. Every node now has a buffer of size B associated with it, so that again each node takes up one block. Leaves are now just buffers of size B.

1. Query: To query a BRT, start by searching the root's buffer. If it isn't there, then recurse to the appropriate child, as in a normal (2,4)-tree. This has cost $O(\log \frac{N}{B})$, since that is the depth of the tree.

2. Insertion: To insert into a BRT, try insert into the root's buffer. Then, if it is full, flush the buffer to its children (recursively), by iterating through the buffer, and sending each element to the appropriate child's buffer. If a leaf has a full buffer and an element gets added to it, then it splits, promoting one of its elements upwards (recursively if needed), just as in a normal (2,4)-tree.

   This has amortized cost of $O(\frac{1}{B} \log \frac{N}{B})$ - The root is always in memory, and all but $\frac{1}{B}$ of the time, there is an open spot. When there isn't, you need to move B items down the tree at some constant cost per move. Each item can be moved at most the depth of the tree, so $O(\log(\frac{N}{B}))$.

Note, however, that this can have poor worst-case time for insertions, since if all child buffers are full you may need to essentially remake the entire tree. This worst case performance can be limited to $O(\log \frac{N}{B})$ by instead only flushing downwards the elements associated with a single child of the root (the one which would get the most elements).

5 Sorting

Multi-way merge sort: Split the list into $\frac{M}{nB}$ groups for some constant n. Sort each group recursively, then merge them.

Given the groups already sorted, it has cost $O(\frac{N}{B})$ to merge them: you just keep one block from each group in memory at once, merge elements in order into some output buffer in memory, which you write to disk when full. Throughout the whole merge you end up reading all of the list into memory at some point, and writing it back one block at a time, for $\approx 2\frac{N}{B}$ IOs.

So, a recurrence relation for the cost is $T(N) = O(\frac{N}{B}) + \frac{M}{B}T(\frac{N}{\frac{M}{B}})$, $T(B) = 1$, since once the list is down to size B, you can just sort it in memory at no cost. Solving this recurrence gives:

$$T(N) = O(\frac{N}{B} \log_{\frac{M}{B}} \frac{N}{B})$$

5.1 Proof of optimality

Just as in the normal setting $O(n \log n)$ is provably optimal, this is optimal in the DAM model by a very similar method.

Consider sorting over the restricted class of inputs, where each block is already sorted internally. So, you want to sort a list of size N, broken up into $\frac{N}{B}$ sorted pieces. There are $\frac{N!}{(B!)^{N/B}}$ possible permutations within this restricted set, so sorting requires $N \log N - \frac{N}{B} B \log B = N \log \frac{N}{B}$ bits of information.

Whenever a block is read, you learn the position of each element of that block relative to every other element already in memory. This gives you $\log \binom{M+B}{B} \approx \log(\frac{M+B}{B})^B \approx B \log \frac{M}{B}$ bits of information.

So, the cost of sorting must be $\Omega(\frac{N \log \frac{N}{B}}{B \log \frac{M}{B}}) = \Omega(\frac{N}{B} \log_{M/B} \frac{N}{B})$

References

[AV88] Alok Aggarwal, Jeffrey Scott Vitter. The input/output complexity of sorting and related problems *Commun. ACM*, 31(9):1116–1127, 1988.

[BM72] Rudolf Bayer, Edward M. McCreight. Organization and Maintenance of Large Ordered Indices. *Acta Inf.* 1: 173–189, 1972.

[BGVW00] Adam Buchsbaum, Michael H. Goldwasser, Suresh Venkatasubramanian, Jeffery Westbrook. On external memory graph traversal. In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 2000.