Advanced Data Structures Jan-Apr 2012

Lecture 25 — April 27, 2012

Lecturer: Venkatesh Raman *Scribe: Sudeshna Kolay*

1 Overview

So far, in the algorithms and data structures we have looked at, we assumed a RAM model where accessing any location takes a constant amount of time. However, in real computers, usually there are more than one layer of memory, with varying access costs. In modern computers, the CPU operates on values in its registers. These values are fetched from the main memory, usually through several layers of caches. The main memory may not have enough space to store large files. Therefore, such files are stored on a disk, and when necessary data is transferred from disk to main memory. The main memory is the fastest but also the smallest. As we go along the layers of memory while the available space increases the access speed goes down. Moreover, it takes about as long to fetch a consecutive kilobyte from a disk as it does to fetch a single byte, since most of the cost is time for the disk to find the right location. Therefore, we transfer a full block of data at a time from the slower memory layers. We also need to device algorithms and data structures that exploit the fact that contiguous data can be accessed at one go, arranging the data in the memory such that those which are usually accessed together are placed in the same blocks, thereby minimizing the number of blocks that need to be accessed. In this lecture we will look at two models of computation that take this memory hierarchy into account: the external-memory model and its refinement, the cache-oblivious model.

2 External-Memory Model (EMM)

The external-memory model or the disk access model was defined by Aggarwal and Vitter [1]. In this model, a CPU is connected directly to a fast cache of size M, which in turn is connected to a much larger but slower disk. The block size in both the layers of memory is prespecified to be B. The cache thus holds $\frac{M}{B}$ blocks, while the disk can hold many more. Algorithms can do memory transfer operations, ie, read a block from disk to cache, or write a block from cache to disk. The cost of an algorithm is the number of memory transfer operations required. In this model, operations executed on the data stored in the cache are considered to be for free. Notice that any algorithm with a running time of $T(N)$ in the cell probe model can be trivially converted into an *EMM* algorithm that requires no more than $T(N)$ memory transfers. Ideally, in *EMM*, we would like to achieve $\frac{T(N)}{B}$ running time, but this optimum is often hard to achieve.

2.1 Basic Results in the EMM

1. Scanning : scanning N items costs $O(\lceil \frac{N}{B} \rceil)$ memory transfers.
2. Searching : we use B-trees of $O(B)$ branching to do efficient searches. We split the data into consecutive parts of length $\frac{N}{B}$. The B endpoints of the parts are stores in the root node

of the B-tree. The structure is then recursively built. Everytime a node is cached, we find the useful range to which the searched item could belong and cache the corresponding child node. Finally we can return an output once the entire current useful range lies in a single block(node of B-tree). This takes $O(\log_{B+1} N)$ time. In fact there is an information theoretic argument that shows that this is the optimal bound for searching.

3. Sorting : For sorting, instead of a 2-way merge sort, we do a $\frac{M}{B}$-way merge sort. This has running time $O(\frac{N}{B} \log_{\frac{M}{B}} \frac{N}{B})$ memory transfers. This can be shown to be optimal [1].

3 Cache-Oblivious Model (COM)

In *EMM*, we assume that we known M and B from beforehand. But in reality, this is not a good assumption. To deal with this the cache-oblivious model was introduces and explored by Frigo et. al [2, 3].

3.1 Results in the COM

1. Scanning : again scanning takes $O(\lceil \frac{N}{B} \rceil)$ memory transfers, no matter what the M and B are.

2. Searching : This will be done using cache-oblivious B-trees to achieve a running time of $O(\log_{B+1} N)$ amortised [6, 7, 4].

3. Sorting : Aagin sorting N elements can be done cache-obliviously by $O(\frac{N}{B} \log_{\frac{M}{B}} \frac{N}{B})$ memory transfers [2, 5].

3.2 Cache-Oblivious B-tree

3.2.1 Static search trees : vEB layout

In the van Emde Boas layout, for simplicity we assume N to be a power of 2 [3]. The more general case can also be analysed [6]. In the considered case , we first build a complete binary tree on N vertices. The tree is then broken along half its height. The upper subtree is of height $\frac{1}{2} \log n$ and is analogous to the Top structure of the *vEB* trees. This is linked to $\sqrt{N}$ subtrees, each of height $\sqrt{N}$. Each of these subtrees is again recursively broken up. When a recursion reaches a size less than B then we can stop. On the disk, the upper subtree is stored first, followed by a sequential layout of the lower subtrees.

The *straddling* level is the recursive level where we first come upon a tree where, on cutting into half, we get the height of each subtree to be $< \log B$. This also means that the height of each subtree is $> \frac{1}{2} \log B$. Thus, the size of each subtree is between B and $\sqrt{B}$ and the number of elements in this level is between B^2 and B.

Claim : Performing a search on a vEB layout requires $O(\log_{B+1} N)$ memory transfers.

Proof : By the recursive structure, for any root to leaf search path we pass through at most $\frac{\log n}{\frac{1}{2}\log B}$ subtrees of the *straddling* level. Each subtree of this level requires at most 2 memory transfers(as worked out above). So the entire searchc requires $O(4\log_{B+1} N)$ memory transfers.

3.2.2 Dynamic search trees

Problem *Ordered File Maintenance (OFM)* : Involves storing N elements in an $O(N)$ array in a certain ordering. Constant sized gaps are thus allowed between elements. This data structure supports *INSERT* between consecutive elements and *DELETE*. Each such operation may involve an amortised moving of an array interval of $O(\log^2 N)$ elements. We will use this as a black box.

Given such an N element input , we build a static binary search tree over the *OFM* of size $O(N)$, where the key of a node in the complete binary tree is the maximum key of its subtrees. Now we will look at the time for supporting the following operations:

1. *SEARCH* : By default we look at the key of the left child to find the potential subtree that may contain the searched element as a leaf. So this takes $O(\log_{B+1} N)$ memory transfers.
2. *INSERT* : First we use *SEARCH* to find the predecessor and successor of the searched element X. Then we insert X in the *OFM*, which may need to change $O(\log^2 N)$ elements. Finally, we need to make changes in the binary search tree built over the *OFM* to maintain the properties of construction.
3. *DELETE* : We first *SEARCH* in the BST; *DELETE* the searched element from the *OFM*; make necessary changes in the BST afterwards.

Claim : If the *OFM* changes k leaves, then updating the BST requires $O(\frac{k}{B} + \log_{B+1} N)$ memory transfers.

Proof : Consider the *straddling* level. Here again there are: 1) chunks of size $\leq B$; and 2) chunks of size $> B$. Type 2 chunks contain $O(B^2 + 1)$ type 1 chunks. We want to count the group of type 2 chunks spanning the k leaves that the *OFM* changes. As long as $M \geq 4B^5$, the cost of updating one type 2 chunk (equal to the cost of scanning it) is $O(\frac{|type2|}{B})$ memory transfers. Summing across all the type 2 chunks, these bottom-most updates cost $O(\frac{k}{B})$ memory transfers.

Suppose each type 2 chunk has size J. After the above series of updates for the chunks of the *straddling* levels, each type 2 chunk has been reduced to one node. Now, the larger subtrees composed of those type 2 chunks also have size $J > B$ So there are $O(J) = O(B)$ BST nodes to traverse before the LCA of the k changed elements is reached. Above the LCA, there are $O(\log_{B+1} N)$ elements remaining on the k path to the root, each of which may require a memory transfer. This gives us a total of $O(\frac{k}{B} + \log_{B+1} N)$ memory transfers.

3.2.3 Indirection

We see that for *OFMs* this k is amortised $O(\log^2 N)$. We can remove the $\frac{k}{B}$ from the memory tranfer bound by using indirection, similar to constructions for y-fast tries.

Again we group the elements into $O(\frac{N}{\log N}$ groups of size $O(\log N)$ elements each. Then we select the set of minimums from each group and build an *OFM* for these elements. Then, the vEB-BST over the *OFM* will be built over $O(\frac{N}{\log N})$ leaves. The vEB storage allows us to *SEARCH* the top structure in $O(\log_{B+1} N)$.

At most one lower group also has to be scanned, at cost $O(\frac{\log N}{B})$. So the total search cost is $O(\log_{B+1} N)$ memory transfers. *INSERT* and *DELETE* will require us to change an entire group at a time, but this costs $O(\frac{\log N}{B}) = O(\log_B N)$ memory transfers, which is okay. Once groups become too small or too full can be merged or split by destroying or forming new groups. We need $\Omega(\log N)$ updates to cause a merge or split. The merge and split costs can be charged to the updates, and their amortized cost becomes $O(1)$. The minimum element only needs to be updated when a merge or split occurs. So expensive updates to the vEB structure only occur every $O(\log N)$ updates at cost $O(\frac{\log_{B+1} N + \frac{\log^2 N}{B}}{\log N}) = O(\frac{\log N}{B}) = O(\log_{B+1} N)$. Thus all operations are supported in $O(\log_{B+1} N)$ time.

References

[1] A. Aggarwal and J.S. Vitter. *The input/outut complexity of sorting and related problems*, Commun. ACM, 31(9):1116-1127, 1988.

[2] M. Frigo, C.E. Leiserson, H. Prokop, S. Ramachandran. *Cache-oblivious Algorithms*, Proc. FOCS, 285-298, 1999.

[3] H Prokop. *Master's thesis*, Massechusetts Institute of Technology, June 1999.

[4] G.S. Brodal, R. Fagerberg, R. Jacob. *Cache-oblivious search trees via binary trees of small height*, Proc. SODA, 39-48, 2002.

[5] G.S. Brodal, R. Fagerberg. *Cache-oblivious distribution sweeping*, Proc. ICALP, 4-26, 2003.

[6] M.A. Bender, E.D. Demaine, M. Farach-Colton. *cache-oblivious B-trees*, Proc. FOCS, 399-409, 2000.

[7] M.A. Bender, Z. Duan, J. Iacono, J. Wu. *A locality-preserving cache-oblivious dynamic dictionary*, Proc. SODA, 29-38, 2002.