Algorithms in the Real World (15-853), Spring 2018
Assignment #6 Due: Tuesday, April 10, 2018

Complete all problems.

You are not permitted to look at solutions of previous years' assignments. You can work together in groups, but all solutions must be written up individually, and please list your collaborators. If you get information from sources other than the course notes and slides, please cite the information, even if from Wikipedia or a textbook.

For algorithm design problems, you should try to provide pseudocodes or pseudocode-like well organized verbal description. Then you should analyze the complexity (work and span) of your algorithms based on the pseudocodes. Please make your solutions as clear as possible, and they will be graded by the correctness and clearness. If only some "high-level" concepts are provided, you will get no more than half of the credits even if they are correct.

1 Buffered B-tree

(a) The nodes split in the same way as in a B-tree or buffer tree, and hence all leaves are at the same height. Since each internal node (except the root) has $\Theta(B^\epsilon)$ children, the height is $O(\log_{B^\epsilon} N) = O(\frac{\log_B N}{\epsilon})$. Observing that each leaf contains $\Theta(B)$ objects, and hence there are only N/B leaves, yields a bound of $O(\frac{\log_B(N/B)}{\epsilon})$ on the height. These two bounds are asymptotically equivalent if $N > B^2$, and hence either is acceptable.

(b) The cost of inserting into a leaf that is already in memory is $O(1/B)$. The analysis for this is the same as in a B-tree.

(c) Insertions into the root is free unless the root buffer overflows and things move down the tree. Whenever objects are moved from a parent node down to its child node, which costs $\Theta(1)$ block transfers to load both nodes, $\Omega(B^{1-\epsilon})$ objects are moved. Thus, the amortized cost of moving each object is $O(1/B^{1-\epsilon})$. Since an object moves once at each height in the tree, the total cost is $O(\frac{\log_B N}{\epsilon B^{1-\epsilon}})$.

(d) An inserted object first moves through buffers (counted in b) then eventually gets inserted into a leaf (counted in c). The cost of flushing the buffers already counts for the cost of loading the leaf, so the assumption in (b) that the leaf be in memory is fair.

Combining (b) and (c) yields $O(\frac{\log_B N}{\epsilon B^{1-\epsilon}})$.

(e) A search need only look in a single root-to-leaf path and the corresponding buffers. Since each buffer and node fits in a single block, the cost is 1 block transfer per height of the tree, for a total of $O(\frac{\log_B N}{\epsilon})$.

(f) The buffer tree does not allow for efficient searches — each buffer is unordered, so a search may have to load the entire buffer at each node, yielding a search cost of $O(\frac{M}{B}\log_{M/B} N) = \omega(M/B)$ for a buffer tree. The search cost for the buffered B-tree, on the other hand, is logarithmic.

(g) The buffered B-tree is worse than the B-tree for searches by a $1/\epsilon$ factor, but it is better for insertions by a $\epsilon B^{1-\epsilon}$ factor. If insertions are more frequent than searches, the buffered B-tree may be much better.

2 Row-to-column Update on Matrices

Part (a):

Similar to the cache-oblivious algorithms we discussed in the lecture on matrix transpose and matrix multiplication, we can use a divide-and-conquer approach. We partition the input row and output column into halves (4 subproblems in total), and solve them individually and recursively.

We note that if A is stored in row-major order, then the accesses of the output array are not cache-efficient. So we copy A to A' and store A' in column-major order, and an access to both the input row and the output column can cover B elements. The base case is when the input and output in a recursive subtask has an overall size of $O(M)$, which uses $O(M/B)$ I/Os. The overall cost is $Q(n) = 4Q(n/2)$, which solves to be $O(n/B + n^2/BM)$.

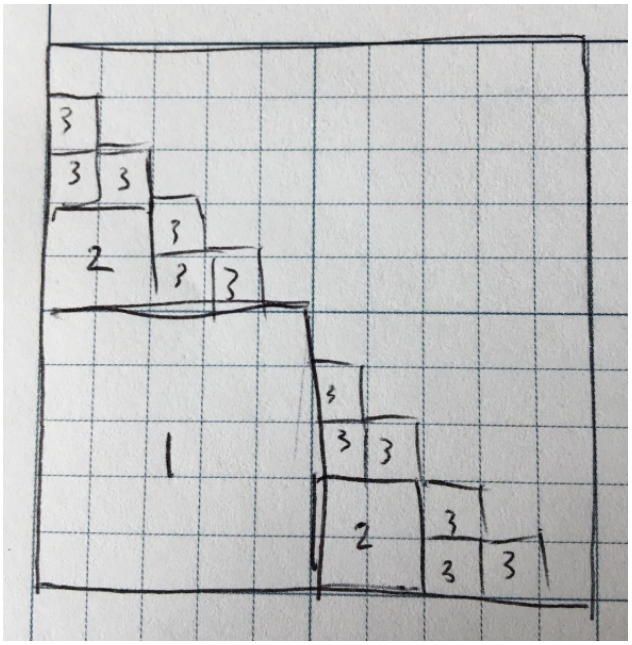
Part (b):

We can run the algorithm in part (a) for n rounds, except that the values of $a_{k,i}$ for $i < k$ need to be copied back to A and used as the input value for the k-th update. These cells in the array A look like a staircase, shown in the figure below. (The rest of elements can be copied back to A' at the end using the matrix transpose algorithm we discussed in the lecture. Or we can just use A' for output, since all the values in A' are up-to-date.)

The value of $a_{k,i}$ need to be copied back after the i-th update, and before the k-th update. This restriction allows us to "group" more elements and copies them back together to save some I/Os. Our algorithm works as follow.

After the $(n/2)$-th round, we transpose and copy all elements in $A_{\{n/2+1,\cdots,n\},\{1,\cdots,n/2\}}$ from A', and recurse on the two halves (i.e., copy $A_{\{n/4+1,\cdots,n/2\},\{1,\cdots,n/4\}}$ after $(n/4)$-th round, $A_{\{3n/4+1,\cdots,n\},\{n/2+1,\cdots,3n/4\}}$ after $(3n/4)$-th round, etc.). An illustration of this process is given in the following figure (thanks to Rohan Yadav!). We leave the proof of correctness to the readers.

It is easy to check that using the matrix transpose algorithm discussed in the lecture, the cost of transposing a square matrix of size n' is $O(n'/B)$ when $n' \geq B$. For the transposing for $n' < B$, based on the tall-cache assumption that $M \gg B^2$, we can assume and afford to



hold such a small submatrix in the cache using n' cache-lines, and the cost of loading them and writing them back is already charged in part (a).

In total, the overall additional cost to copy the data from A' to A is $O(n^2/B)$, which does not change the bound shown in part (a).