CS 229r: Algorithms for Big Data Fall 2013

Lecture 22 — November 14, 2013

Prof. Jelani Nelson *Scribe: Arpon Raksit*

1 The cache oblivious model

Last time we started talking about external memory algorithms, in particular in the disk access model (DAM). Today we're going to look the following related model, called the cache-oblivious model [FLPR99].

Definition 1. Keep precisely the same setup as DAM—in particular we have memory of size M and blocks on disk of size B. But for the *cache-obvlious model* we make the following changes.

1. The algorithm or data structure is *not allowed to know or use the parameters M and B.* Note, however, our goal in constructing and analysing these algorithms will be to *still get good I/O performance in terms of M and B*, as in DAM.
2. In DAM we had full control over the system, i.e., what to evict from memory. Here we assume *the system evicts memory automatically and optimally.*

Example 2. From last time: array traversal was clearly cache oblivious, but our algorithm for multiway merge sort, for instance, critically used the value of M, and hence was not cache-oblvious.

The second assumption of the cache oblivious model might seem problematic or unrealistic, but in fact it is not. We'll justify it by mentioning a few definitions and theorems about online algorithms.

1.1 Online algorithms

Definition 3. An *online algorithm* receives a sequence of inputs—basically, a stream—without knowing the entire sequence in advance, and must make decisions on each input.

At the end of the day, we compare the cost of its decisions with the cost of an *omniscient/optimal algorithm* OPT, which does see the entire input in advance and then makes optimal decisions.

Example 4. The canonical example is ski rental. Here renting skis for a day costs one dollar, and buying skis costs ten dollars. Each day your friend says one of two things:

A: Let's go skiing.

B: I'm done skiing for the rest of my life. There's no more skiing, ever.

So the input is a sequence of **A**s, terminated at some point by a **B**. We want to minimise how much we pay for skiing.

Of course OPT, being omniscient, would just count the number a of **A**s which will occur in the input, and buys the skis on day one if $a \geq 10$, or rents skis daily if $a < 10$.

To be 2-competitive (i.e., be within a factor of 2 of the cost of OPT) as a real person, simply rent for the first 10 days then buy on the 11th day (if it occurs).

In our situation, the online problem is choosing how to evict memory. The omniscient algorithm would evict the block that will be fetched again farthest in the future (you can prove this is in fact optimal). But we don't know the future, so what to do in the real system? Two commonly used strategies/algorithms are:

FIFO: first-in / first-out

LRU: least recently used

They're exactly what they sound like. These strategies are nice because of the following fact.

Theorem 5 (Sleator-Tarjan [ST85])**.** *FIFO and LRU are:*

1. M*-competitive against* OPT.
2. 2*-competitive against* OPT *when* OPT *is given* $M/2$ *memory.*

Why does this justify the cache-oblivious model? Well, as long as $T(N, M, B) \approx T(N, M/2, B)$, where $T(-, -, -)$ is the cost given by the analysis of our cache-oblivious algorithm, then Theorem 5 implies that using FIFO or LRU instead of the assumed OPT results in no (asymptotic) loss in performance.

2 Some cache oblivious algorithms

2.1 Array {tra, re}versal

For traversal, as mentioned in Example 2, the DAM algorithm from last time was actually cache oblivious: we just scan the array in blocks of size B at a time. I/O cost is still at most $\lceil N/B \rceil + 1$.

For reversal, we just by traverse the array backwards and forwards and swap along the way. So by traversal cost, cost for reversal is $O(N/B)$.

2.2 Square matrix multiplication

Here our DAM algorithm from last time doesn't carry over to the cache-oblivious model, since we explicitly broke up the matrix into blocks of size $\sqrt{M}$ by $\sqrt{M}$. But we'll still be able to do something simple. We split our matrices into four blocks as follows:

$$\begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix} = \begin{pmatrix} A_{11}B_{11} + A_{12}B_{21} & A_{11}B_{12} + A_{12}B_{22} \\ A_{21}B_{11} + A_{22}B_{12} & A_{21}B_{12} + A_{22}B_{22}, \end{pmatrix}$$

reducing multiplication of $n \times n$ matrices to eight multiplications and four additions of $n/2 \times n/2$ matrices. Moreover we'll store our matrices A and B on disk as follows.

A_{11}	A_{12}	A_{21}	A_{22}

B_{11}	B_{12}	B_{21}	B_{22}

Then we'll apply this construction recursively for each A_{ij} and B_{ij}. E.g., A_{11} will be further decomposed (within the decomposition of A above) as follows.

$$\begin{pmatrix} (A_{11})_{11} & (A_{11})_{12} \\ (A_{11})_{21} & (A_{11})_{22} \end{pmatrix} \qquad \boxed{(A_{11})_{11}} \boxed{(A_{11})_{12}} \boxed{(A_{11})_{21}} \boxed{(A_{11})_{22}}$$

This gives us a recursive algorithm for matrix multiplication.

Let's analyse number $T(n)$ of I/Os. We 8 recursive multiplications, and the additions just require scans over $O(n^2)$ entries. Thus recurrence is given by $T(n) \leq 8\,T(n/2) + O(n^2/B + 1)$. The base case is $T(\sqrt{M}) = O(M/B)$, since we can read an entire $\sqrt{M} \times \sqrt{M}$ matrix into memory (due to the recursive data layout!). Solving this gives $T(n) = O(n^2/B + n^3/(M\sqrt{B}))$.

Remark 6. This technique of recursively laying out data to get locality, and then using M and B to get a good base case of our analysis, will be quite useful in many situations.

2.3 Linked lists

Don't know who to credit this construction to, but you can find it in the survey [D02].

Shooting for: $O(1)$ insertion and deletion, and $O(1 + k/B)$ to traverse k elements (amortised).

Data structure: just an array where each element has pointers to the next and previous locations that contain an element of the list. But it'll be self-organising (like union-find, if you remember that).

Insertion: append element to end of array. Costs $O(1)$ I/Os.

Deletion: mark the array location as empty. Costs $O(1)$ I/Os.

But now elements might be far apart in the array, so on traversal queries we're going to fix up the data structure (this is the self-organising part).

Traverse k elements starting at x: we traverse as usual using the pointers. But in addition, afterwards we delete the k elements we traversed from their locations and append them to the end of the list.

Rebuild: finally, every $n/2$ updates we completely rebuild the linked list by putting all elements (contiguously) in a new array.

OK, let's analyse this. Note first that rebuilding costs $O(n/B)$, which amortises to $O(1/B)$ over the $n/2$ updates.

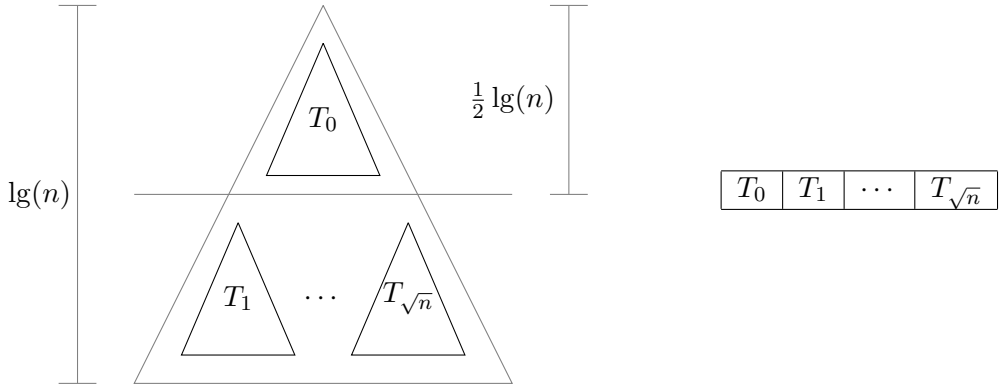
Now, what does traversal cost? When we do a traversal, we touch r contiguous runs of elements. Thus the number of I/Os in the traversal is $O(r+k/B)$—1 for each run, and the cost of a scan of the k elements. But there must have been r updates to cause the gaps before each run. We amortise the $O(r)$ over these r updates, so that traversal costs $O(k/B)$ amortised. (To be more precise: any sequence of a insertions, b deletions, and a total of k items traversed costs $O(a+b+1+k/B)$ total I/Os.) And since after a traversal we consolidate all of the runs, the r updates we charged here won't be charged again.

> "One money for me means one I/O." – Jelani Nelson

That's amortised linked lists. If you want worst-cased bounds see [BCD+02].

2.4 Static B-tree

We're going to build a B-tree (sort of) without knowing B. The data structure will only support queries, that is, no insertions [FLPR99]. We'll use another recursive layout strategy, except with binary trees. It looks as follows (conceptual layout on left, disk layout on right). Keep in mind this picture is recursive again.



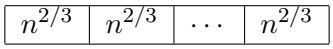
We query as usual on a binary search tree. To analyse the I/O cost, consider the first scale of recursion when the subtrees/triangles have at most B elements. Reading in any such triangle is $O(1)$ I/Os. But of course there are at least $\sqrt{B}$ elements in the tree, so in the end traversal from root to leaf costs $O(\log(n)/\log(\sqrt{B})) = O(\log_B(n))$ I/Os.

2.5 Lazy funnel sort

Original funnel sort by [FLPR99], simplified by [BFJ02]. Yet another recusrive layout strategy, but a lot funkier. Assume we have the following data structure.

Definition 7. A *K-funnel* is an object which uses $O(K^2)$ space and can merge K sorted lists of total size K^3 with $O((K^3/B)\log_{M/B}(K^3/B) + K)$ I/Os.

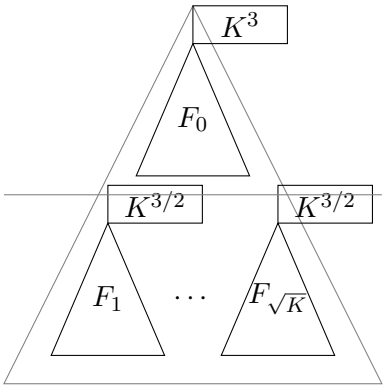
Lazy funnel sort splits the input into blocks of size $n^{2/3}$, recuseively sorts each block, and merges blocks using the K-funnel, with $K = n^{1/3}$.

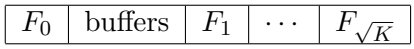


When analysing this, we'll make the following *tall cache assumption*. Unfortunately this assumption is required to get the desired $O((n/B)\log_{M/B}(n/B))$ I/Os for sorting [BFJ02].

Assumption 8 (Tall cache)**.** Assume $M = \Omega(B^2)$. But note that this can be relaxed to $M = \Omega(B^{1+\gamma})$ for any $\gamma > 0$.

We're running low on time so let's just see what the K funnel is. It's another recursive, built out of $\sqrt{K}$-funnels. The funnels are essentially binary trees, except with buffers (the rectangles, with labelled sizes) attached.





At each level the $\sqrt{K}$ buffers use $O(K^2)$ space. Then the total space used is given by the recurrence $S(K) = (1 + \sqrt{K})S(\sqrt{K}) + S(K^2)$. Solving this gives $S(K) \leq O(K^2)$.

How do you use a K-funnel to merge? Every edge has some buffer on it (which all start off empty). The root node tries to merge the contents of the buffers of the two edges to its children. If they are empty, the root recursively asks its children to fill their buffers, then proceeds to merge them. The recursion can go all the way down to the leaves, which are either connected to the original K lists to be merged, or are connected to the output buffers of other funnels created at the same level of recursion (in which case you recursively ask them to fill their output buffers before merging).

Theorem 9. *As described, lazy funnel sort costs $O((n/B)\log_{M/B}(n/B))$ I/Os (under the tall cache assumption).*

Proof sketch. The recurrence above proved funnel property (1). For funnel property (2), look at the coarsest scale of recursion where we have J funnels, with $J \ll \sqrt{M}$. The details are in [D02]. □

References

[BCD+02] Michael A Bender, Richard Cole, Erik D Demaine, Martin Farach-Colton, and Jack Zito, *Two simplified algorithms for maintaining order in a list*, ESA 2002, Springer, 2002, pp. 152–164.

[BFJ02] Gerth Stølting Brodal, Rolf Fagerberg, and Riko Jacob, *Cache oblivious search trees via binary trees of small height*, Proceedings of the thirteenth annual ACM-SIAM symposium on Discrete algorithms, Society for Industrial and Applied Mathematics, 2002, pp. 39–48.

[D02] Erik D Demaine, *Cache-oblivious algorithms and data structures*, Lecture Notes from the EEF Summer School on Massive Data Sets (2002), 1–29.

[FLPR99] Matteo Frigo, Charles E Leiserson, Harald Prokop, and Sridhar Ramachandran, *Cache-oblivious algorithms*, Foundations of Computer Science, 1999. 40th Annual Symposium on, IEEE, 1999, pp. 285–297.

[ST85] Daniel D Sleator and Robert E Tarjan, *Amortized efficiency of list update and paging rules*, Communications of the ACM **28** (1985), no. 2, 202–208.