6.851: Advanced Data Structures Spring 2012

Lecture 22 — May 19, 2012

Prof. Erik Demaine *Scribe: Abhijit Mudigonda (2017), Thuy Duong Vuong (2017), Ariel Jacobs (2017)*

Computers have been around for decades, and there's a lot of interest in modeling memory in a way that's simultaneously *clean* and *practically relevant*.

1 Idealized Two-Level Storage

In this model, we have a CPU and a RAM. RAM consists of blocks which store $\leq B$ items each. We'll *ignore the item order within each block*, only caring about which block an item is in.

Definition 1 (Block Operation)**.** *In a **block operation**, we read up to B items from two blocks i, j and copy them to a third block k.*

We'll assume that CPU operations are free and that items are indivisible.

Theorem 1. *Permuting N items into $\frac{N}{B}$ specified blocks (making sure there's a copy of each item in the item's target block) needs*

$$\Omega\left(\frac{N}{B}\log B\right)$$

block operations in the average case, if we make the tall disk assumption ($\frac{N}{B} > B$).

Note that we have an immediate lower bound of $\frac{N}{B}$ since we at least need to read all the blocks. The tall disk assumption (number of blocks exceeds block size) is reasonable here - if there were only one block, then we'd need to do no work!

We'll prove this using a simplified model - rather than our block operations copying elements, they'll **move** elements from two blocks into a third. This is equivalent in the case of permutation because we'd end up throwing away the copies anyways.

Proof. We define a potential function

$$\Phi = \sum_{i,j} n_{ij} \log n_{ij}$$

where n_{ij} is the number of items in block i destined for block j. This is maximized in the target configuration of full blocks, where $n_{ii} = B$ and thus $\Phi = N \log B$. To see this, notice that we can rewrite the potential function as $\log \prod_{ij} n_{ij}^{n_{ij}}$ and this is maximized when there are a few big n_{ij} rather than many small n_{ij}.

If we start with a random configuration (since we're working in the average case), and assume $\frac{N}{B} > B$, then

$$\mathbb{E}[n_{ij}] = O(1) \implies \mathbb{E}[\Phi] = O(N)$$

where the first equality follows from linearity of expectation.

We claim that each block operation increases Φ by at most B, and therefore the number of block operations is at most

$$\frac{N \log B - O(N)}{B}$$

The claim follows from the inequality

$$(x+y)\log(x+y) \leq x \log x + y \log y + x + y,$$

and therefore merging two clusters (from 2 different blocks) of x and y elements respectively into one blocks increase Φ by at most $x + y \leq B$.

□

Theorem 2. *There is an algorithm that permutes N items into $\frac{N}{B}$ specific blocks using*

$$O\left(\frac{N}{B} \log \frac{N}{B}\right)$$

block operations.

Proof. We use something similar to radix sort, where each item is keyed by the ID of its target block. Then, a sort puts everything in the right order. We require $\log \frac{N}{B}$ passes through the items (one per bit) and each pass takes $O\left(\frac{N}{B}\right)$ time because block operations. □

2 Red-Blue Pebble Game

We still have a CPU and a disk, but now we also have a **cache** in the middle. Furthermore, we work with items rather than blocks.

2.1 Single-Color Pebble Game

The motivation for the red-blue pebble game is a single-color pebble game that we can use to model computation. View the computation as a DAG of data dependencies (so each vertex corresponds to a computation that requires the values of its ancestors).

Definition 2 (Single-Color Pebble Game)**.**

- ***Start:*** *One pebble on each input vertex.*
- ***Moves:***
  - *Place a pebble on a node if all its predecessors have pebbles.*
  - *Remove pebble from node.*
- ***Goal:*** *Have pebbles on all output vertices.*

We are interested in ***minimizing*** *the number of pebbles that are ever on the board.*

The number of pebbles at any given point in the game is equivalent to the number of results we must store at any point in the computation.

Theorem 3. *Any DAG with n vertices can be executed using $O\left(\frac{n}{\log n}\right)$ pebbles.*

Corollary 4.

$$\mathsf{TIME}(t) \subseteq \mathsf{SPACE}\left(\frac{t}{\log t}\right).$$

2.2 Red-Blue Pebble Game

Now, to add in a notion of a cache, we have two colors of pebbles. **Red** pebbles correspond to items stored in cache, and **blue** pebbles correspond to items stored on disk. We'll let M be the size of the cache (the maximum number of red pebbles we can use).

Definition 3 (Red-Blue Pebble Game)**.**

- ***Start:*** *One blue pebble on each input vertex.*
- ***Moves:***
  - *Place a **red** pebble on a node if all its predecessors have **red** pebbles.*
  - *Remove pebble from node.*
  - ***Write:*** *- Convert a red pebble to a blue pebble.*
  - ***Read:*** *- Convert a blue pebble to a red pebble.*
- ***Goal:*** *Have **blue** pebbles on all output vertices while using at most M **red** pebbles. We are interested in **minimizing** the number of reads and writes we need to make.*

2.3 Red-Blue Pebble Game Results [1]

We can compare the number of memory transfers required in the RBPG model to the RAM runtime analysis for various problems. Note that in general, we expect the speedup to be at most M.

Comparison DAG	**Memory Transfers**	**Speedup**
Fast Fourier Transform	$\Theta(N \log_M N)$	$\Theta(\log M)$
Matrix-vector multiplication	$\Theta\left(\frac{N^2}{M}\right)$	$\Theta(M)$
Matrix-matrix multiplication	$\Theta\left(\frac{N^3}{\sqrt{M}}\right)$	$\Theta(\sqrt{M})$
Odd-even transposition sort	$\Theta\left(\frac{N^2}{M}\right)$	$\Theta(M)$
Lattice of size N^d	$\Omega\left(\frac{N^d}{M^{\frac{1}{d-1}}}\right)$	$\Theta(M^{\frac{1}{d-1}})$

3 I/O Models

The idealized two-level model has blocks but no cache. The RBPG model has a cache but no blocks. The I/O model combines these, and is also known as the **External Memory Model** (we studied these earlier). We have a cache of size M and an unlimited disk, both with block size B.

3.1 Scanning

A common technique in external memory is scanning. Visiting N elements of the disk in order costs $O\left(1 + \frac{N}{B}\right)$ memory transfers. We can also parallelize this, by running up to $\frac{M}{B}$ scans in parallel (one block per scan in cache). This lets us do things like merging $O\left(\frac{M}{B}\right)$ lists of size N in $O\left(1 + \frac{N}{B}\right)$ transfers.

3.2 Searching

Proposition 1. *We can search a sorted list in the external memory model in $\Theta(\log_{B+1} N)$ memory transfers.*

Proof. Consider the problem of searching in a sorted list. For a lower bound, we know that each block read gives us at most $\log B + 1$ bits of information (where our element fits among $B + 1$ elements), and we need $\log N + 1$ bits of information in order to find our element. Thus, we need to make at least $\frac{\log N+1}{\log B+1} = \log_{B+1} N$ reads.

For the upper bound, we use **B-Trees** with a block size of $B + 1$, which gives us exactly the bound we want! This also gives us insertions and deletions in $O\left(\log_{B+1} N\right)$, but this turns out to not be optimal for insertions and deletions. □

3.3 Sorting and Permutation

Proposition 2. *In the external memory model, we can sort in time*

$$\Theta\left(\frac{N}{B}\log_{\frac{M}{B}}\frac{N}{B}\right)$$

and can permute (if we already know where each element needs to go) in time

$$\Theta\left(\min\{N, \frac{N}{B}\log_{\frac{M}{B}}\frac{N}{B}\}\right).$$

Note that if N is lower than the sorting bound, permutation can be faster by just inserting each item where it needs to go, rather than by a sorting.

Proof of sorting bound. For a lower bound on sorting, we start by assuming that the cache is always sorted (this is free), and that every block in disk has been internally presorted. Now, when we read a block we learn how those B items fit within the M items in cache (at best), so we learn

$$\log\binom{M+B}{B} \approx B\log\frac{M}{B}$$

We need $\log N! \approx N\log N$ bits, but we already know $N\log B$ bits from the presorting. Thus, the number of transfers is at least

$$\frac{N(\log N - \log B)}{B\log\frac{M}{B}} = \frac{N}{B}\log_{\frac{M}{B}}\frac{N}{B}.$$

Note that we aren't counting any transfers needed for presorting but we don't need to because it's a lower bound.

For the upper bound, we use $O\left(\frac{M}{B}\right)$-way **merge sort**. The value $\frac{M}{B}$ is chosen so that we can do this all in parallel (this is the maximum number of "sorting threads" we can store in cache, one block per thread). The recurrence is

$$T(N) = \frac{M}{B}T\left(N/\frac{M}{B}\right) + O\left(1 + \frac{N}{B}\right)$$

with a base case of $T(B) = O(1)$ because we can sort a single block with one transfer. The recurrence works itself out with a recursion tree. □

Another approach that works is **distribution sort** [2] rather than merge sort. It's basically a $\sqrt{M/B}$-way **quick sort**. We find $\sqrt{M/B}$ partition elements that are evenly spaced, partition the array into $\sqrt{M/B}+1$ pieces using those partition elements in $O(N/B)$ memory transfers via scanning, and recurse. To find the partition elements, we scan each of the (N/M) M-elements intervals into memory, sort it in $O(M/B)$ time, then sample every $\frac{\sqrt{M}}{4\sqrt{B}}$th elements in the resulting sorted in-memory interval. In total, we get $\frac{4N}{\sqrt{M/B}}$ sampled elements in $O(\frac{N}{M}\frac{M}{B}) = O(N/B)$ memory transfers. For $i = 1, 2, .., \sqrt{M/B}$, we run linear-time selection (think median finding, turns out it also achieves linear mem. trans.) to find sample element at $i/\sqrt{M/B}$ fraction among the sampled elements in $O(\frac{4N}{B\sqrt{M/B}})$ memory transfers for each i, for a total of $O(\frac{N}{B})$ memory transfers. The recursive formula is similar to in merge sort, and we achieve the same number of memory transfers for sorting.

4 Sequential I/O

We want to capture the idea that accessing items in sequence on disk is faster than randomly accessing items. In other words, bulk-read/write $\Theta(M)$ items in sequence is faster than random access.
In this model, binary merge sort achieves $O(\frac{N}{B}\log\frac{N}{B})$ sequential memory transfers. This is optimal, because if we want the number of random memory transfers to be $o(\frac{N}{B}\log_{M/B} N/B)$ (a.k.a less than a constant fraction of sorting lower bound), then we need $\Omega(\frac{N}{B}\log\frac{N}{B})$ sequential memory transfers.

5 Hierarchical Memory Models

Two-levels are nice, but in reality, we often have many levels. We want to model this behavior, where we have increasingly large amounts of increasingly slow memory. One common assumption is that accessing memory location costs $f(x) = \lceil \log x \rceil$.

5.1 HMM Upper and Lower Bounds

Here are some results. Note that there is a **slowdown** from the "usual" runtime because we now have greater cost for accessing things. The slowdown can't be more than $\Theta(\log N)$ because every element can be accessed in $O(\log N)$ time.

Problem	**Time**	**Slowdown**
Semiring matrix multiplication	$\Theta(N^3)$	$\Theta(1)$
Fast Fourier Transform	$\Theta(N \log N \log\log N)$	$\Theta(\log\log N)$
Sorting	$\Theta(N \log N \log\log N)$	$\Theta(\log\log N)$
Scanning input	$\Theta(N \log N)$	$\Theta(\log N)$
Binary search	$\Theta(\log^2 N)$	$\Theta(\log N)$

However, this isn't that useful - $\log x$ is actually fairly arbitrary. Instead, we want a generalization.

5.2 $\text{HMM}_{f(x)}$

The natural generalization is to say that accessing memory location x costs $f(x)$. We make the assumption that

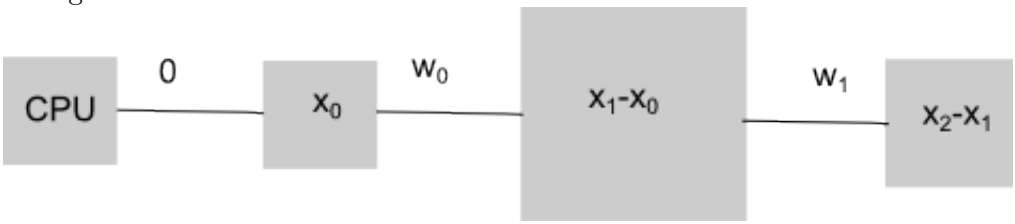
$$f(2x) \leq cf(x)$$

for some constant $c > 0$, so thus f is "polynomially bounded" - for intuition, note that polynomials are polynomially bounded but exponentials aren't.

We write

$$f(x) = \sum_i w_i \cdot [x \geq x_i?]\,,$$

a weighted sum of threshold functions.



Then, each memory chunk is of size $x_i - x_{i-1}$.

5.3 Uniform Optimality

Consider $f_M(x) = [x \geq M?]$ (so there's exactly one term, corresponding to a cache of size M.

Definition 4. *An algorithm is **uniformly optimal** if it's optimal on $HMM_{f(x)}$ for all M) (alternately, it's **oblivious** to M.*

Proposition 3. *If an algorithm is uniformly optimal for $f(x) = f_M(x)$ (optimal for all M) then the algorithm is optimal for all $f(x)$.*

Results in $HMM_{f_M(x)}$ model:

Problem	**Time**	**Speedup**
Semiring matrix multiplication	$\Theta(\frac{N^3}{\sqrt{M}})$	$\Theta(\sqrt{M})$
Fast Fourier Transform	$\Theta(N \log_M N)$	$\Theta(\log M)$
Sorting	$\Theta(N \log_M N)$	$\Theta(\log M)$
Scanning input	$\Theta(N - M)$	$\Theta(1 + \frac{1}{M})$
Binary search	$\Theta(\log N - \log M)$	$\Theta(1 + \frac{1}{\log M})$

5.4 Implicit HMM Memory Management

Instead of a more advanced eviction strategy, one could consider using some conservative replacement strategy, like FIFO (First in, First out) or LRU (Least Recently Used). It's shown that, given a polynomially increasing growth function,

$$T_{LRU}(N, 2M) \leq 2T_{OPT}(N, M)$$

,

or that an LRU strategy on twice the size of the cache, takes at most twice as long as the optimal eviction strategy.

This can be achieved by splitting the memory into chunks where caches are included in the chunk until $f(x)$ doubles. When LRU evicts, it should do so into the next chunk.

5.5 HMM with Block Transfer

Copying memory intervals in blocks is easier than each location individually. With Block Transfers, the bounds for common operations will depend on f.
In this model, we have the following results

Problem	$f(x) = \log x$	$f(x) = x^\alpha, 0 < \alpha < 1$	$f(x) = x$	$f(x) = x^\alpha, \alpha > 1$
Dot product, merging lists	$\theta(N \log N)$	$\theta(N \log \log N)$	$\theta(N \log N)$	$\theta(N^\alpha$
Matrix mult.	$\theta(N^3)$	$\theta(N^3)$	$\theta(N^3)$	$\theta(N^\alpha)$ if $\alpha > 1.5$
Fast Fourier Transform	$O(N \log N)$	$O(N \log N)$	$O(N \log^2 N)$	$O(N^\alpha)$
Sorting	$O(N \log N)$	$O(N \log N)$	$O(N \log^2 N)$	$O(N^\alpha)$
Binary search	$O(\frac{\log^2 N}{\log \log N})$	$O(N^\alpha)$	$O(N)$	$O(N^\alpha)$

5.6 Memory Hierarchy Model

This is a multi-level version of external memory model. We have n caches $M_0, M_1, .., M_n$ of increasing depth and size. Between two consecutive caches M_i and M_{i+1}, allow transferring blocks of size B_i in t_i time. We also allow parallel transfer a.k.a all levels can be transferring blocks at the same time.
We restrict our attention to the **Uniform Memory Hierarchy** (UMH) model[3], i.e. a memory hierarchy model where we fix **aspect ratio** $\alpha = \frac{M/B}{B}$ and **block growth** $\beta = \frac{B_{i+1}}{B_i}$
Then we can express the cache size, block size, and block transferring time of each cache M_i in the parameters α, β as follow:

- $B_i = \beta^i$
- $M_i/B_i = \alpha\beta^i$
- $t_i = \beta^i f(i)$

In the UMH model, we have the following results:

Problem	Upper bound	Lower bound
Matrix Transpose ($f(i) = 1$)	$O((1 + \frac{1}{\beta^2})N^2)$	$\Omega((1 + \frac{1}{\alpha\beta^4})N^2)$
Matrix Mult. ($f(i) = O(\beta^i)$)	$O((1 + \frac{1}{\beta^3})N^3)$	$\Omega((1 + \frac{1}{\beta^3})N^3)$
FFT ($f(i) \leq i$)	$O(1)$	$\Omega(1)$

5.7 Cache-Oblivious Model

Similar to the External Memory Model, but we don't know B and M. Block transfers are automatic, as we don't know the size or blocksize of the cache. This model is clean, and generalizes to the Multilevel Memory Hierarchy since we can imagine those caches to the left of some boundary as the cache, while the ones to the right are disk. Because our model works well for all sized caches, it will perform over that boundary and all others.

This is largely covered earlier in this course.

5.7.1 Scanning

We can assume $M \geq cB$ for a constant $c > 0$, and run $O(1)$ parallel scans. Since we don't know M or B, we can't do more parallel scans. But this means we can merge two lists.

5.7.2 Searching

We can use the van-Emde-Boas layout to store cache obliviously.

It loses to External Memory, as we know that Cache Oblivious search takes $(\lg e + o(1)) \log_B N$. This can be made dynamic with B-Trees in $O(log_B N)$.

5.7.3 Sorting

The tall cache assumption assumes the cache is at least as tall as it is wide, or $M \geq \Omega(B^{1+\epsilon})$.

If we make the tall cache assumption, we can achieve $O(\frac{N}{B} \log_{M/B}(\frac{N}{B})$ with a mergesort analog, or distribution sort analog.

If we don't make the tall cache assumption, we can't achieve the sorting bound. We also can't achieve the permutation bound, as we can't measure whether the sorting strategy is better than moving each individual item itself in the Cache Oblivious model.

References

[1] H. Jia-Wei and H. T. Kung, "I/O complexity: The red-blue pebble game," in *Proceedings of the Thirteenth Annual ACM Symposium on Theory of Computing*, ser. STOC '81. New York, NY, USA: ACM, 1981, pp. 326–333. [Online]. Available: http://doi.acm.org/10.1145/800076.802486

[2] A. Aggarwal and S. Vitter, Jeffrey, "The input/output complexity of sorting and related problems," *Commun. ACM*, vol. 31, no. 9, pp. 1116–1127, Sep. 1988. [Online]. Available: http://doi.acm.org/10.1145/48529.48535

[3] B. Alpern, L. Carter, E. Feig, and T. Selker, "The uniform memory hierarchy model of computation," *Algorithmica*, vol. 12, no. 2, p. 72, Sep 1994. [Online]. Available: https://doi.org/10.1007/BF01185206