COMS 4995-3: Advanced Algorithms Apr 24, 2017

Lecture 25 – Introduction, Schedule, Approximate Counting

Instructor: *Alex Andoni* Scribes: *Anton Igorevich Nefedenkov* and *Anand Sundaram*

1 Introduction

Today's lecture consisted of 2 portions. The first portion was about external memory, and completed the topic from the last lecture of showing how to efficiently perform binary search in the external memory model. The second section was about parallel algorithms and introduced several different models used to study parallel algorithms.

2 External Memory Model and Efficient Algorithms

2.1 External Memory Model recap

Recall that the main idea of the external memory model is that there are 2 different memory storage locations:

1. A cache which contains a small amount of memory storage and can be accessed rapidly
2. A main memory location (disk) which contains a much larger amount of storage space and takes much more time to access

The time complexity of algorithms in this model is measured by the cost in terms of the number of I/O operations incurred to copy memory from the disk to the cache. Data is assumed to be stored both on disk and in the cache in blocks of size B, and I/O can transfer 1 block per unit of time, so the cost is the total amount of data needed from disk divided by the block size.

2.2 Search Problem Recap

The main problem we want to solve is to search for a single data element in a database as quickly as possible. The motivation is that record lookup in a database is an extremely natural, common task.

We assume that we are looking for 1 unit of data in a database of size N on an architecture with blocks of size B.

Claim 1 (Binary Search)**.** *Lookup is* $O(\log N - \log B)$

Proof. We can assume that the database will be stored on disk in whatever data structure we design it to have; we choose a Binary Search Tree (BST) storage model. Then, we can use binary search to look up the entry of interest. There are $\frac{N}{B}$ blocks of size B needed to store the full database in memory. Binary search on this many blocks takes $\log \frac{N}{B}$ I/O interactions since each interaction pulls one block from memory to cache; this is equivalent to$\log N - \log B$. □

2.3 Faster search with B-trees

We will show that we can improve on this runtime using better data structures. In particular, we will prove that B-trees can achieve better runtime.

Definition 2 (B-Tree)**.** *A B-tree is a generalization of a binary tree to have B-ary branching.*

Each B-tree node contains up to $B-1$ elements. Also, each node stores up to B pointers to sub-B-trees ,with one pointer each between each pair of elements in a node and before the first element and after the last element in a node, unless the the tree does not grow in a particular direction from this node (for example if the node is a leaf node, in which case it does not grow in any direction).

So a node would contain elements $a_1, a_2, \ldots a_{B-1}$, and pointers $A_0, A_1, \ldots A_{B-1}$. The B-tree has the property that it is sorted; that is:

1. The elements within a node are sorted:

$$(\forall i \in [1, 2, \ldots B-2]) : [a_i < a_{i+1}]$$

2. For any subtree whose pointer is stored to the left of any specific element in a node, all elements of that subtree are less than that specific element:

$$(\forall i \in [0 \ldots B-2]) \, (\forall b_i \in A_i) : [b_i < a_i]$$

3. All elements in the subtree to the right of the largest element in a node are larger than that element:

$$(\forall b_i \in A_{B-1}) : [a_{B-1} < b_i]$$

Observation 3 (B-tree height)**.** *It should be clear from this definition that, to represent an N element database, we can construct a tree with $O(\log_B N)$ height*

Proof. There are B^h elements that can be stored in a tree of size h with B-ary branching as described above. □

Search We use a recursive search algorithm to find an element x in a B-tree:

1. Look at the root. If x is an element in the root node, we are done.
2. Otherwise, find the element a_i of the root node such that $a_i < x < a_{i+1}$. Recurse into A_i, the subtree whose pointer is stored between a_i and a_{i+1}.

Runtime analysis The classic runtime we get from analysis of the algorithm just stated is $O(\log N \cdot \frac{B}{\log B})$.

Proof. The runtime is at most the height of the tree times the time spent per node searching for the element (in case we have to search all the way down to a leaf). The height of the tree is $O(\log_B(N))$, and we need to spend at most B steps in a node finding the elements which neighbor the pointer to the sub-tree containing the missing value. This gives us $O(\log_B(N) \cdot B)$ which is $O(\log N \cdot \frac{B}{\log B}) = O(\log N)$. □

This can also be sped up to $O(\log N)$ by performing binary search in each node of the tree.
In external memory:

Theorem 4 (B-Tree Speedup)**.** *Using B-Trees, we can accomplish database lookup in* $O(\log_B(N)) = O(\frac{\log N}{\log B})$ *time in the external memory model.*

Proof. $O(\log_B N)$ I/O operations are needed (the number of transfers from external memory to cache), because two transfers are sufficient to read an entire node with $B-1$ elements and the associated B pointers. □

Note Sorting takes $O(\frac{N}{B} \cdot \log_{\frac{M}{B}} \frac{N}{B})$ time. This is because we can use merge-sort and in a machine with M space in the cache, we can perform $\frac{M}{B}$-way merge-sort, giving us the $\frac{M}{B}$ base for the logarithm in the runtime.

2.4 Efficient Algorithms for the Cache-Oblivious Model

2.4.1 The Cache-Oblivious Model and Motivations

The cache-oblivious model is a variant of the external memory model where we don't know B or M. There are several motivations for this model:

1. We often want to write software that can run on many different hardware architectures, but each hardware architecture has its own cache size and block size, so it is beneficial to have optimize the software runtime for a cache-oblivious model to make it portable across architectures

2. The cache size may change over time in practice in real usage, since cache may be shared by multiple programs running in parallel on an operating system, and load may vary over time depending on what the user chooses to run. The amount of cache space and perhaps even which cache and therefore the block size of the cache allocated to a given processor may vary significantly over time.

3. In practice there are often many levels to the memory hierarchy used by real systems. For example the CPU may access an L1 cache which accesses an L2 cache which accesses some other memory which accesses disk memory, and each level of memory may have exponentially more storage space but significantly slower access times than the higher level.

For all of these reasons, it is good to have algorithms that have good runtime for all values of B and M.

2.4.2 Cache-Oblivious Binary Search

We want to run binary search as fast as possible in a cache-oblivious model.

An obvious idea is to try to store a BST in the memory array as if it were a heap. A heap is a BST stored in an array A where the root of the BST is stored in the zeroth array cell of A, $A[0]$. Then for each node n stored at position p (in cell $A[p]$), if it has left child l and right child r, l is stored in cell $A[2p+1]$ and r is stored in cell $A[2p+2]$.

However, with this method of storing a BST, search still takes $O(\log N)$ time for $N \gg B^2$. Notice that we need to follow each parent-child link to walk down the binary tree. If $f(l)$ is a function which specifies the distance between a child and its parent as a function of the number l of levels below the root that the child is located, $f(l) = \Omega(2^l)$ in the heap layout of a BST. This means after a certain point, we need to read a new cache line for every link, because the parents and their children towards the bottom of the tree are always stored in distinct lines, because 2^l will rapidly become bigger than B. In particular, if a binary tree has N elements, it has $\log N$ height, and the number of elements stored in the top half is $2^{\frac{\log N}{2}} = \sqrt{N}$. So if $N \gg B^2$, then $\sqrt{N} > B$, so every parent-child link in the bottom half of the tree must span multiple cache lines in the heap layout. When searching for a leaf node, we have to traverse all of the $\log N$ levels of the tree, so we will to access at least $\frac{\log N}{2}$ distinct cache lines in the bottom $\frac{\log N}{2}$ levels of the array.

Claim 5. *We can achieve $O(\log_B N)$ I/O transfers to look up an element from a database even in the cache-oblivious model by using binary search.*

Proof. The data structure is just a modification of a Binary Search Tree (BST). We can use a van Emde Boas tree, which is a data structure architecture which stores a BST in an array-shaped memory in such a way as to achieve asymptotically fast cache-oblivious runtime.

Notice that the top $\log B$ levels of the original BST can already be stored in 1 cache line. We don't know what B is, but we pick some small number k (say, for example, $h = 10$), and store the first k elements of the original BST (the top $\log k$ levels) together in the memory array. If $k < B$, then this is all in one cache line (or at most two, if there is misaligned storage). If $k > B$, then this is stored in at most $\lceil \frac{k}{B} \rceil + 1$ cache lines. Let's take advantage of this observation to store the entire tree in memory in a better order.

To store the entire tree, we split the BST with N elements into $\sqrt{N}$ sub-trees of size at most $\sqrt{N}$ each (where each sub-tree is still a properly structured BST) by "cutting" the tree in the middle (level $\frac{1}{2} \log N$). We keep track of the pointers between subtrees. Let's call each such sub-tree a "mini-tree". Then we recurse into each of these mini-trees and repeat the process, splitting it into $N^{\frac{1}{4}}$ mini-trees of size at most $N^{\frac{1}{4}}$. We repeat this process for as many iterations as necessary until each mini-tree at the end of some iteration contains at most k elements, for the k chosen earlier. Note that we have now achieved our goal of storing the first k elements together in a constant number of cache lines, but also every other element in the tree is stored together with a local mini-tree containing k elements in the same constant number of cache lines.

Now suppose we traverse the entire height of the original BST from root to a leaf in the new data structure. The number of distinct mini-trees we touch is the height of the BST divided by the height of each mini-tree, which is $\frac{\log N}{\log k} = \log_k N = \frac{\log_B N}{\log_B k}$. Each mini-tree is stored locally in at most $\lceil \frac{k}{B} \rceil + 1$ cache lines. Therefore the total number of cache lines we need to access is at most $O(\frac{k}{B} \frac{\log_B N}{\log_B k})$.

To simplify the analysis let's assume we pick a really small k so that for all reasonable architectures the cache lines are larger than the mini-tree: $B > k$. Without loss of generality we can assume more specifically $\sqrt{B} < k < B$, so we perform at most 1 additional sub-division into mini-trees of size less than

B; if k is smaller than this relative to B then our upper bounds in terms of B only get better. Then each tree has a number of elements k such that $k \in [\sqrt{B}, B]$, because we divided the mini-tree up once after reaching mini-trees of size B because they weren't small enough yet. This means the height of the subtrees h is such that $h \in [\frac{\log B}{2}, \log B]$. So the number of mini-trees touched is $\frac{\log N}{\frac{\log B}{2}} = 2\log_B N$, and each mini-tree is stored in at most two cache lines, so we need to access at most $4\frac{\log N}{\log B}$ cache lines. □

2.5 Cache-Oblivious Block Transfers

How should we design the cache to decide when to transfer blocks from memory and store them? We can do this automatically using an algorithm such as FIFO (first-in, first-out) or LRU (least recently used). The difference between these is as follows. Suppose the cache has 100 lines. In FIFO, any line in the cache will disappear after 100 lines different from it have been requested from memory, even if that line was used more often and more recently than some of the different lines that were requested; re-requesting a line which is already in the cache will not change its priority to be removed from the cache. In LRU, every time we re-request a line we reduce its priority to be removed from the cache, causing it to be kept for longer.

Both of these algorithms are 2-competitive. This means that the number of I/O transfers between memory and cache needed with either of these cache-management algorithms to support any other CPU algorithm is always at most 2 times as high as the number of transfers of an optimal algorithm running on a cache of size $\frac{M}{2}$.

It's still not clear from the definition above exactly what 2-competitiveness means. Competitiveness is a concept from analysis of online algorithms, which need to optimize some objective in a real-time situation without full knowledge of what requests will be made in the future. The competitive ratio of an algorithm is then a comparison of a realistic, oblivious algorithms, such as FIFO or LRU, which does not know what will be requested in the future but wants to use as few I/O transfers as possible, to an omniscient algorithm which somehow knows everything the CPU will request in the future and can arrange the cache at every time with this future knowledge to minimize the number of I/O transfers. However, we do not directly compare the practical, oblivious algorithm to the provable best performance of a hypothetical omniscient algorithm on the same size of cache, but we compare the oblivious algorithm's performance on a cache of size M to the omniscient, optimal algorithm's performance on a cache of size $\frac{M}{2}$. This technique is called "resource augmentation", and it is a tool commonly used in analysis of online algorithms.

3 Parallel Algorithms

The idea of parallel algorithms has come in and out of fashion for some time now and a number of different models have been developed. For this part, we are going to denote the size of the input by n.

3.0.1 Circuits

We are given a set of gates that is a basis set for the space of binary functions. One example of such set are gates $\{AND, OR, NOT\}$. Another example is the basis set $\{NAND\}$. All binary functions can be

rewritten in terms of NAND gates. The intuition behind the parallelism of the model is the parallism of hardware: outputs of gates at the same depth is computed in parallel. On top of this model, we can define a number of complexity classes:

- AC: solvable assuming that gates have **unbounded** fan in.
- NC: solvable assuming that gates have **bounded** fan in.
- AC_k: solvable in $O(\lg^k n)$ depth, $n^{O(1)}$ nodes/gates, and **unbounded** fan in.
- NC_k: solvable in $O(\lg^k n)$ depth, $n^{O(1)}$ nodes/gates, and **bounded** fan in.

Example 6. *Compute AND of n bits.*
AC: depth 1, $O(1)$ gates and $O(n)$ wires.
NC: depth $O(\lg n)$.

Notice, that classes are indeed dependent of the basis chosen. Consider the following example.

Example 7. *Compute XOR of n bits.*
AC: depth $\Theta(\frac{\lg n}{\lg \lg n})$, $n^{O(1)}$ gates.
Reason: we don not have XOR gates directly available to us, so we have to simulate them in terms of AND and NOT gates.

3.0.2 PRAM Model

Parallel Random Access Machine model was conceived in late 80's, but its utility proved to be limited. The excitement about the model was fueled by the hope that one day the model will be implemented directly in the hardware, something that never happened. This is the setup:

- p independent processors
- shared memory
- synchronized model

We of course have to resolve read/write conflicts. There are four different strategies to do so: exclusive read exclusive write (EREW), concurrent read exclusive write (CREW), exclusive read concurrent write (ERCW) and concurrent read concurrent write (CRCW). "Exclusive" in this context means that only one processor has access to a certain memory cell; "concurrent" means that multiple processes have access to the same memory cell. As an aside, even when using the most strict conflict resolution strategy, XOR problem discussed above requires $\Omega(\frac{\log n}{\log \log n})$ time

3.0.3 BSP: Bulk-Synchronous Parallel Model

This model is due to Leslie Valiant. While we won't define it completely, we'll see a particular variant of it, which arose from the study of MapReduce algorithms. The model below is formally called Massively Parallel Computing (MPC). This is the setup:

- Input of size n

- Parallel computation on m distinct machines / processors
- A space upper bound of s per machine

The computation proceeds in rounds. Within each round, computation is done on local information. At the end of the round, we shuffle - the local information is exchanged between different machines. The amount of information exchanged in/out is at most s per machine per round. The cost per time in this model is measured by the number of rounds of computation performed.
mbox

We will prove really efficient algorithms for this model. Notice that in the $PRAM$ family of models described above, even on the most relaxed CRCW (concurrent read concurrent write) model, sorting requires $\Omega(\frac{\log n}{\log \log n})$ time. However, we will see that in this model we can get very low upper bounds for interesting problems. We will do this in the next lecture.