COMS E6998-9: Algorithms for Massive Data (Spring'19) April 9, 2019

Lecture 21: Large-scale Models (Continued) / Parallel Models

Instructor: *Alex Andoni* Scribes: *Tim Randolph*

1 Fast Binary Search in the Cache-Oblivious I/O Model

1.1 Recall: The Cache-Oblivous I/O Model

We have an input of size n, stored in main memory. When we access main memory, we receive a cache block of size B. We measure "time" in terms of the number of cache blocks we have to access.

1.2 Van Emde Boas Tree

A Van Emde Boas tree is a binary tree, stored in memory according to a specific recursive procedure. We store a binary tree T with n nodes in memory as follows:

1. If T has some small constant number of nodes (say, 8) we just store it explicitly. Otherwise:
2. Recursively store the tree T_0 consisting of the approximately $\sqrt{n}$ nodes at depth $\frac{log(n)}{2}$ or less in T.
3. Recursively store the trees T_1, T_2, ... T_m rooted at the leaves of T_0.

Remark: the Van Emde Boas tree is static as described, but can be made dynamic.

1.3 Van Emde Boas Trees Allow Cache-Oblivious Binary Search in $O(log_B(n))$ Accesses

Claim 1. *Consider a Van Emde Boas tree T with n nodes. Running a binary search on T requires $O(log_B(n))$ memory accesses.*

Proof. The Van Emde Boas procedure recursively divides T into smaller and smaller trees, breaking each tree of size m into trees of size $\sqrt{m}$. Consider the smallest such tree R such that $|R| > B$. Letting $R_0, R_1, ..., R_k$ be the subtrees that constitute R, we have $|R_0| = |R_1| = ... = |R_k| \geq \sqrt{B}$. Accordingly, we need to access memory at most twice to pass through a tree of size $|R_0|$.

Consider the path that the binary search takes through T. This path has length $\log_2(n)$, while trees of size $|R_0|$ have depth at least $\log(\sqrt{B})$. Thus the path passes through at most $\log_2(n)/\log_2(\sqrt{B})$ such trees, and the number of memory accesses we need is upper-bounded by

$$2\frac{log_2(n)}{log_2(\sqrt{B})} = 4log_B(n). \tag{1}$$

□

2 Models for Parallel Computation: Circuits

A circuit consists of n input wires $x_1, x_2, ..., x_n$ which take binary input values. Wires meet at gates such as AND ($\wedge$), OR ($\vee$), and NOT ($\neg$), which compute on their inputs and assign the result to their output wire. Circuits have a single output wire, and thus each circuit C represents a function $C : \{0,1\}^n \to \{0,1\}$.

In this model, we define

$$ptime := \text{circuit depth (length of the longest path from input wire to output wire.)} \tag{2}$$

$$work := \text{total number of gates.} \tag{3}$$

$$fan-in := \text{maximum number of wires that go into a gate.} \tag{4}$$

These metrics can vary slightly based on what *base* of gates we use: for instance, we could use $\{AND, OR, NOT\}$ or just $\{NAND\}$. However, both of these bases are *universal*, meaning they can be used to construct circuits for any computable function and can simulate other gates/bases with constant factor blowup.

Several complexity classes encompass the problems that can be computed by circuits with certain restrictions. These classes are called *non-uniform*, meaning you're allowed to construct a different circuit for each size of input.

$$AC_i := \text{Problems solvable with } ptime = O(log^i(n)), \text{unbounded fan-in, } n^{O(1)} \text{ work.} \tag{5}$$

$$AC = \bigcup_{i \in \mathbb{N}} AC_i. \tag{6}$$

$$NC_i := \text{Problems solvable with } ptime = O(log^i(n)), O(1) \text{ fan-in, } n^{O(1)} \text{ work.} \tag{7}$$

$$NC = \bigcup_{i \in \mathbb{N}} NC_i. \tag{8}$$

$$\tag{9}$$

Example 2. ***Computing*** $AND(x_1, x_2, ..., x_n)$**.** *This problem is in AC_0, because we can just AND all the inputs together with a single gate (fan-in is unbounded.) Intuitively, this problem is NOT in NC_0, because if our fan-in is bounded by c, we need at least $log_c(n)$ gates to incorporate all the information from the input in the output wire. However, it IS in NC_1 by the same intuition (just create a c-ary tree of AND gates).*

Theorem 3. *Computing $XOR(x_1, x_2, ..., x_n)$ requires $\Omega(\frac{log(n)}{log(log(n))})$ depth on AC circuits.*

Claim 4. *Any function $f : \{0,1\}^n \to \{0,1\}$ can be computed by a circuit with fan-in at most n, 2^n gates, and depth $O(1)$.*

Proof sketch. Consider a circuit in which the first layer is $O(n)$ input wires, the second layer contains $O(2^n)$ gates which each accept a single string in $\{0,1\}^n$ (encoded by input wires), and the third layer is the AND of all gates corresponding to strings $s \in \{0,1\}^n$ such that $f(s) = 1$. □

3 Models for Parallel Computation: PRAM

In this model, we have p processors and a memory with size M (which stores input, output, and our workspace). Time is synchronous, and every processor can access the memory once per time-step. In this model, we define

$$ptime := \text{number of ticks required to solve the problem.} \tag{10}$$

$$work := p \cdot ptime \text{ (We can think of this as 'energy consumed.')} \tag{11}$$

It's not immediately clear how we should resolve conflicts when two processors try to read or write from the same location at the same time. This gives rise to several models:

- Exclusive-read exclusive-write (EREW): multiple processors are not permitted to read/write in the same time step.
- Concurrent-read concurrent-write (CRCW): multiple processors ARE permitted to read/write in the same time step. Conflicts are resolved randomly or by priority order of processors.
- Exclusive-read concurrent-write (ERCW) and Concurrent-read exclusive-write (CREW): defined by analogy to the above.

Theorem 5. *Even in the CRCW (most powerful) variety of PRAM, $XOR(x_1, x_2, ..., x_n)$ requires ptime $= \Omega(\frac{log(n)}{log(log(n))})$.*

Remark: although the PRAM model was very popular in the 80s and 90s, it turned out to be easier to make faster hardware and better algorithms than massively parallel hardware, so it has declined in popularity over time.

4 Models for Parallel Computation: Massively Parallel Computation

Massively Parallel Computation (MPC) is a species of the Bulk Synchronous Parallel (BSP) model. In particular, we have an input of size N and m machines (processors). Each machine has space s. (Typically, we have $ms = O(N)$, $s = N^\epsilon$, $m = O(N^{1-\epsilon})$.) Note that we must have $ms \geq N$ or else we can't even store all the input. The input to the problem is divided up between all the machines, and we assume it's distributed between machines in a worst-case manner. If the output size is greater than s, it is also stored in a distributed fashion.

Time is divided into *rounds*. Each round consists of the following two steps:

1. Each machine performs an unlimited amount of local computation.
2. Each machine sends at most s bits of information to other machines, and receives at most s bits of information in turn.

$$ptime := \text{number of rounds required to solve the problem.} \tag{12}$$

Example 6. ***Computing*** *$XOR(x_1, x_2, ..., x_N)$. Each machine can XOR its local bits, and then s machines can transfer their output bits to the same place. Repeating this procedure until we're left with a single bit takes $log_s(N)$ rounds.*