6.854 Advanced Algorithms

Lecture 22: 12/1/2003 Lecturer: Erik Demaine

Scribes: Grant Wang

22.1 Streaming algorithms

22.1.1 Introduction

We will see more of streaming algorithms next week, but we will see a short introduction today. The idea behind streaming algorithms is very similar to that of online algorithms. A streaming algorithm receives data as a sequence, which we call a stream. We get one piece of data at a time – we process it, and then get another piece of data, ad nauseum.

The key difference between online and streaming algorithms is that online algorithms is a matter of time: online algorithms is all about the future, whereas streaming algorithms are all about the past. That is, in online algorithms, we are concerned with how our decisions will affect us in the future. For instance, the ski rental problem, we would like to know whether or not to buy despite the fact that we do not know what will occur in the future. But in streaming algorithms, we only care about the *past*: we want to compute some function of the data that has been given to us.

This sounds trivial: we can just store all the data, and compute the function in the end. The key catch is that we are not allowed to store all the data. If the stream contains n items, we are only allowed to use $\log n$ space or even constant space. Clearly, we must give up on computing the function exactly, so we look for approximate solutions.

There are numerous applications of streaming algorithms. Consider the previous topic of external memory algorithms. If we need to read off of disk, we would like to minimize the number of times we do this. Streaming algorithms deal with the case when we can only use a single scan of the disk. Another application area is that of Internet routers. Imagine a router sitting on the Internet, as it watches packets stream by at a rate of a million per second. The number of packets is obviously too high to store, but we still wish to compute interesting properties: where are the packets going to, is there a denial of service going on, etc? Lastly, we have the application area of databases. Consider sitting at the database, watching updates and queries – we would like to know statistics such as: which entries are being queried the most? Streaming algorithms solve such problems.

22.1.2 Puzzle

To get a flavor of streaming algorithms, consider the following puzzle:

> Paul permutes a pantry of pastries and pinches a particular pastry.
>
> Carole considers the continuum of confectionaries but can't commemorate the configuration completely.

Can Carole confidently compute the captured confectionary?

That is, consider the numbers $\{1 \ldots n\}$. Paul has removed a number i from $\{1 \ldots n\}$. Carole sees the stream of numbers $\{1 \ldots n\} \setminus \{i\}$, but she cannot store the bit vector of $\{1 \ldots n\}$ (she can only use $\log n$ space). She wants to determine the missing number.

The solution is quite simple: Carol maintains the sum s and the count c. When the count reaches $n-1$, she reports the missing number as:

$$\frac{n(n+1)}{2} - s$$

Let's consider the slightly harder version: *Paul pinches a pair.* That is, he removes two numbers $\{i, j\} \subseteq \{1 \ldots n\}$. Can Carol recover i, j using $\log n$ space given the stream $\{1 \ldots n\} \setminus \{i, j\}$. Yes, using a very similar solution: she stores s, the sum of the numbers in the stream, and s', the sum of the *squares* of the numbers in the stream. After seeing $n-2$ elements, we solve for i and j using the following two equations:

$$\begin{aligned} i + j &= \frac{n(n+1)}{2} - s \\ i^2 + j^2 &= \frac{n(n+1)(2n+1)}{6} - s' \end{aligned}$$

This generalizes to k missing elements using symmetric polynomials; we won't discuss that here, but will point to a reference:

22.2 Parallel algorithms and Circuits

Next, we will consider two models of computation that are very closely related: parallel algorithms (PRAM) and the circuit model:

- **PRAMs**: Here, we have n processors that can all access global shared memory. In a time step, each processor either accesses a memory location or does one standard operation. We assume the processors are clocked synchronously. The measure of complexity here is the number of processors, as well as the number of time steps to compute a desired function.
- **Circuits**: A circuit is a directed, acyclic graph where each node of the graph is an AND, OR, or NOT gate. The edges are directed; those leaving a node are the outputs, and those entering a node are the inputs. The entire graph has certain input nodes and certain output nodes. The measure of complexity here is the number of gates, which measures the size, and the depth of the circuit (the longest path from output to input). If we believe that each "level" of the circuit must execute before the next level, the depth of the circuit measures the number of time steps the circuit needs to compute the function.

We discuss each of these models in more detail, as well as algorithms in each model next.

22.2.1 Circuits

A key issue we glossed over above is the number of input or output wires any gate is allowed. The *fan-in* is the number of input wires to a gate, and the *fan-out* is the number of output wires to a gate.

Another issue we glossed over are the asymptotics of circuits. We wish to study the size of a circuit that solves a certain problem. However, a circuit of a fixed size can only solve problems of a given size. Instead, for a certain algorithmic problem, we will look at a *family* of circuits that solve a certain problem; that is, a circuit for each input size that solves all instances of that input size. Now we are ready to define complexity classes:

Definition 1 *$AC(k)$ is the class of problems for which there exists a family of circuits such that instances of size n are solved with circuits of size bounded above by a fixed polynomial with depth $O(\log^k n)$. The circuits are allowed to have unbounded fan-in and fan-out.*

Definition 2 *$NC(k)$ is the same class as above, but the circuits have bounded fan-in and fan-out.*

Here are two easy relationships between these classes:

$$\begin{aligned} NC(k) &\subseteq AC(k) \\ AC(k) &\subseteq NC(k+1) \end{aligned}$$

The first inclusion is by definition. The second inclusion follows by replacing any gate with large fan-in with a binary tree of equivalent gates, and any gate with large fan-out with an "upside-down" binary tree. We make one last final definition:

Definition 3

$$NC = \cup_{k=1}^{\infty} NC(k)$$

We think of NC as those problems that can be efficiently parallelized.

Addition

A simple problem to consider for the circuit model of computation is addition. We are given two numbers in binary: $a = a_{n-1} \ldots a_0$ and $b = b_{n-1} \ldots b_0$, and we want the binary representation of $s = a + b$.

The most simple circuit, known as the ripple/carry circuit, follows the grade-school algorithm for addition. We consider the task of computing $a_i + b_i + c_{i-1}$, where c_{i-1} is the last carry bit. The result of computing this addition is s_i, the output bit in the ith location and c_i, the carry bit.

It is clear that this circuit takes $O(n)$ gates: to compute the requisite sum in each of the n places, we need a constant number of gates. However, the depth is $O(n)$ – we must always wait for the last carry bit. Although the size of the circuit is optimal, we would like to decrease the depth to something more like $O(\log n)$. We will see how to do this with a carry/lookahead circuit.

Note: I'll scribe the carry/lookahead circuit soon in the next few days, but there's a very good treatment of the material in CLR, edition 1.

22.2.2 PRAM

We should note that the PRAM model isn't exactly realistic; the assumption of a global shared memory is erroneous. Furthermore, the number of machines is usually fixed at a constant rather than growing with the problem size. In these respects, the PRAM model is not a good model.

However, the PRAM model does accentuate how well a certain problem can be parallelized. Because it is stronger than any realistic model of parallel computation, it allows a greater deal of parallelization. This itself is an interesting issue to study with respect to computational problems, and that is its place.

We also swept other problems under the rug when we introduced this model of computation. In a given time step, each processor can either read or write to memory. However, there are different models in the PRAM model that allow for certain amounts of concurrency or exclusion, which we define next:

Definition 4 $CRCW(k)$ *is the class of problems that can be solved with a polynomial number of processors that are allowed concurrent read and concurrent write to a global shared memory, in* $O(\log^k n)$ *time.*

Definition 5 $EREW(k)$ *is the same class above, except that the machines are allowed only exclusive write and read of the global shared memory.*

Here are simple consequences of the definitions:

$$\begin{aligned} EREW(k) &\subseteq CRCW(k) \\ CRCW(k) &\subseteq EWER(k+1) \end{aligned}$$

The first inclusion is immediate; the second conclusion is not too hard to see. Consider a time step where many processors wish to read one memory location. We will create a binary tree, in which a single processor reads in one time step, and then writes the data to another memory slot. The number of processors that can read in any one step doubles, so in $\log n$ more steps, every processor has read the memory location. The same argument holds for writes, as well.

One might think that there exists a connection between circuits and PRAMs. Indeed such a connection exists:

$$NC(k) \subseteq EREW(k)$$

The idea behind this inclusion is that we can put a processor at each gate, and simulate a circuit layer in one time step. We also have the following inclusion:

$$EREW(k) \subseteq NC(k+\delta)$$

The inclusion here is a bit more difficult to see and we just sketch the argument here. For each machine, we can make a collection of gates that simulates its behavior on inputs of a certain size.

With the above relationships, we get the following nice connection:

$$NC = \cup_{k=1}^{\infty} EREW(k) = \cup_{k=0}^{\infty} CRCW(k)$$

The reason that we have two different models is that sometimes it is easier to think about circuits, whereas other times it is easier to think about processors. Next, we consider a simple problem in the PRAM model: list ranking.

List ranking

Consider a linked list l of length n, with one processor on each item. Our algorithmic problem is as follows: we wish to compute the rank of each item in the list, i.e. the distance of each item to the end of the list.

The method we use is called pointer jumping. Each node i in the linked list has a next pointer, $\mathsf{next(i)}$. At each timestep, we execute the following instruction for each processor:

$$\mathsf{next(i)} = \mathsf{next(next(i))}$$

We claim that in $\log n$ steps, every processor has $\mathsf{next(i)}$ equal to the last node. This occurs because the distance spanned by each pointer doubles in each step. How do we use this to compute the rank of the nodes?

For every node, we keep a variable $d(i)$, which is initially set to 1. In each time step, we make the following update:

$$d(i) += d(\mathsf{next(i)})$$

It is easy to prove by induction on the number of steps that $d(i)$ is equal to the number of nodes the pointer spans.

The number of processors here is n, and the total amount of time spent is $O(\log n)$. The total amount of work, if we were to simulate using a single machine, would therefore be $O(n \log n)$. However, a single processor machine could compute these distances in $O(n)$ time. In the next lecture we will see how to reduce the total amount of work to $O(n)$.