CS 7880: Algorithms for Big Data (Fall'22) **Northeastern University**

Lecture 1

September 9, 2022

Instructor: Soheil Behnezhad *Scribe: Erika Melder*

Disclaimer: *These notes have not been subjected to the usual scrutiny reserved for formal publications.*

1 Overview

Linear-time linear-space algorithms have long been considered the gold standard of efficiency. Indeed, it is hard to imagine more efficient algorithms since even reading and storing the input just once requires linear time and linear space. However, as the size of inputs get larger and larger, even such algorithms become inefficient. Our goal in this course is to study extremely resource-efficient algorithms that can process these massive inputs. We will particularly discuss:

- **Sublinear Time Algorithms:** These algorithms assume the input is already in memory and attempt to provide an answer with $\ll n$ queries to the input of size n.
- **Sublinear Space Algorithms:** Also known as **streaming algorithms**. In this model, we assume that the input arrives piece by piece one at a time and the algorithm does not have enough space to store all of the input.
- **Massively Parallel Computation (MPC):** A framework where many computers over a network each get some small fraction of the total input and can solve a subproblem, and then communicate with other network computers, with the goal being to minimize the number of rounds of computation and communication.

This lecture presents examples of each model in action.

2 Sublinear Time Example

The following problem is a motivating example for sublinear time algorithms.

Problem 1.

Input: An array $A[1, \ldots, n]$ which is some permutation of $[n]$.

Goal: Output an index i such that $A[i]$ is even.

Let us first see a simple deterministic algorithm for this process.

Algorithm (Deterministic): For $i \in \{1, \ldots, n\}$, query each $A[i]$ in order.
If $A[i]$ is even, return i.

Now consider an adversarial input which contains all the odd elements in $[n]$ in its first $\lceil \frac{n}{2} \rceil$ indices. The running time of this algorithm on such an input is $\Omega(n)$, since it must iterate through $\lceil \frac{n}{2} \rceil + 1$ indices to find an even element. As such, deterministic techniques are insufficient to solve this problem quickly.

Remark. This result holds for any deterministic algorithm. One may construct an adversarial input by placing the odd elements of $[n]$ in the first $\lceil \frac{n}{2} \rceil$ indices that the algorithm checks. Then the algorithm takes $\Omega(n)$ time to run on that adversarial input. We will see examples of such lower bound proofs in future lecture notes.

One solution to this problem is to permit the use of *randomized algorithms*, which are allowed to flip coins and branch on the outcome of the coin flips. Here is one such algorithm:

Algorithm (Randomized): Choose a random $i \in n$ and query $A[i]$.
If $A[i]$ is even, return i. Otherwise, repeat.

While this algorithm seems to have a similarly-bad worst case, it is resistant to the kind of adversarial inputs that plague deterministic approaches to this problem. Moreover, the expected runtime is bounded above by a constant value.

Proposition 1. *Let k be the number of iterations required for the above algorithm to produce an output. Then, on any input, the algorithm satisfies $\mathbb{E}[k] \leq 2$.*

Proof. For simplicity, assume that n is even. Note that

$$\mathbb{E}[k] = \sum_{i=1}^{\infty} \Pr(\geq i \text{ queries made})$$

Since n is even, each query succeeds with probability $\frac{1}{2}$. Thus,

$$\Pr(\geq i \text{ queries made}) = \frac{1}{2^{i-1}}$$

Substituting back into the sum yields

$$\mathbb{E}[k] = \sum_{i=1}^{\infty} \frac{1}{2^{i-1}} = 1 + \frac{1}{2} + \frac{1}{4} + \cdots \leq 2. \quad \square$$

3 Sublinear Space Example

A modification of the previous problem demonstrates some of the fundamental difficulties with devising sublinear space algorithms.

Problem 2.

Input: An array $A[1, \ldots, n-1]$ which is some permutation of $[n] \setminus \{k\}$, where $k \in [n]$ is not known. Elements of the array are sent one by one.

Goal: Determine k.

One trivial solution is to store all the elements of A and then iterate over them to determine which is missing. However, storing all the elements of A requires $\Theta(n)$ space. We can do better and achieve constant space complexity with the following algorithm:

Algorithm: Define a variable $s = 0$.
For each element $a \in A$ that is sent, set $s = s + a$.
After all elements are sent, output $\frac{n(n+1)}{2} - s$.

The proof of correctness is as follows:

Proof. Recall that $\sum_{i\in[n]} i = \frac{n(n+1)}{2}$, and note that $s = \sum_{i\in[n]\setminus\{k\}} i$.
Since $\sum_{i\in[n]\setminus\{k\}} i = \left(\sum_{i\in[n]} i\right) - k$, solving for k yields

$$k = \sum_{i\in[n]} i - \sum_{i\in[n]\setminus\{k\}} i = \frac{n(n+1)}{2} - s$$

□

Remark. There is only a single variable stored, which takes $\Theta(1)$ space. However, this variable must be able to store a value of size at most $\frac{n(n+1)}{2} - 1$, which requires $\Theta(\lg(n^2)) = \Theta(\lg n)$ *bits* of space. It is important to note whether a space requirement is given in terms of exact bits or abstracted into words.

Exercise: Prove that $\Omega(\lg n)$ bits of space are required, even for randomized algorithms.

Exercise: Consider a modified version of the problem, where $A[1, \ldots, n-2]$ is some permutation of $[n] \setminus \{k_1, k_2\}$ for unknown $k_1 \neq k_2 \in [n]$ and the goal is to find k_1 and k_2. Show that this is solvable deterministically in $O(4 \lg n)$ bits of space.

Exercise: Consider the general version of the problem, where $A[1, \ldots, n-p]$ is some permutation of $[n] \setminus \{k_1, k_2, \ldots, k_p\}$ for unknown $k_1, \ldots, k_p \in [n]$ and the goal is to find all of the k_i.

- Show that this is solvable in $O(k^2 \lg n)$ bits of space deterministically.
- Show that this is solvable in $O(k \lg \frac{n}{k})$ bits using a randomized algorithm.

4 Massively Parallel Computation

4.1 Description of the MPC Model

First, we describe the model of massively parallel computation (MPC):

- Let n be the input size of the problem. Assume each machine has $s = n^\varepsilon$ space, for some constant $\varepsilon < 1$. Assume that the input is preemptively perfectly divided among machines, so that there are $n^{1-\varepsilon}$ machines.
- Computation occurs in *rounds*. Each round consists of two component phases:
  - *Local computation*, where each machine performs some procedure on its currently-stored elements.
  - *Communication*, where any pair of machines may communicate, subject to the constraint that the total messages sent and received by each machine (including duplicates sent to/received from different machines) may not take more than s space.
- The goal is to devise some algorithm to solve the problem in the fewest rounds possible, where the number of rounds is the number of the round in which computation terminates.

Remark. Note that there is no restriction on the time complexity of the local computation - it may even be exponential in n. For optimization purposes, minimizing local computation time is secondary to minimizing the number of computation rounds in the MPC model.

Remark. Due to the round-based nature of the model, small changes to the problem specification (such as forcing machine 1 to output the answer versus allowing any machine to do so) can occasionally influence the answer. For this reason, standardizing the problem statement ahead of time is important.

4.2 Logical OR Example

Using this model, we can obtain results for a variety of problems, such as the following:

Problem 3.

Input: n bits, $b_1, \ldots, b_n$, arbitrarily partitioned among the machines.

Goal: Machine 1 must output $\bigvee_{i=1}^{n} b_i$.

For $s = \sqrt{n}$, a two-round algorithm is as such:

Algorithm: For $j = 1, \ldots, \sqrt{n}$, enumerate machine j's bits as $b_{(j-1)\sqrt{n}+1}, \ldots, b_{j\sqrt{n}}$.
Define $b'_j = \bigvee_{k=1}^{\sqrt{n}} b_{(j-1)\sqrt{n}+k}$, or the logical OR of all the bits on machine j.

R1 Local Computation: Each machine j computes b'_j.
R1 Communication: Each machine sends b'_j to machine 1.
R2 Local Computation: Machine 1 computes $\bigvee_{k=1}^{\sqrt{n}} b'_k$ and outputs it.

By associativity of $\vee$, this algorithm is correct. Note that this algorithm relies on the fact that $\varepsilon = \frac{1}{2}$, meaning that the number of machines $n^{1-\varepsilon}$ is equal to the space held in each machine n^{ε}. If $\varepsilon < \frac{1}{2}$, then the number of messages will overflow machine 1's storage capacity. This indicates that the number of rounds required by the algorithm is dependent on our choice of ε.

Proposition 2. *Consider a modified version of the algorithm which sends as many of the b'_j to machine 1 as possible, and then to machine 2, and so on, and then recurses on any machines that still hold bits until all of the messages fit onto machine 1. For $\varepsilon \leq \frac{1}{2}$, this algorithm requires $\lceil \frac{1}{\varepsilon} \rceil$ rounds.*

Proof. We first prove a prerequisite claim:

Claim 3. *At the start of round r, the bits require $n^{1-r\varepsilon}$ machines to store them.*

This claim is proven by induction.

Base Case: Trivially, when $r = 1$, the bits require $n^{1-\varepsilon}$ machines.

Inductive Step: Assume the claim holds for arbitrary $r \geq 1$. During the local computation of round r, the $n^{1-r\varepsilon}$ machines each output 1 bit. Each machine has n^{ε} storage, so these bits require a minimum of $\frac{n^{1-r\varepsilon}}{n^{\varepsilon}} = n^{1-(r+1)\varepsilon}$ machines to store them.

This concludes the proof of the claim.

We then seek r such that $n^{1-r\varepsilon} = 1$ to ensure the bits fit onto one machine. Choosing $r = \frac{1}{\varepsilon}$ satisfies this equation. □

We will later see in the course why $\Omega(1/\epsilon)$ rounds are indeed (unconditionally) necessary to solve this problem!

4.3 The 1v2-Cycle Problem

Let us now see a more realistic problem, namely the 1v2-Cycle problem. The following is one of the most fundamental open problems in the study of MPC algorithms.

Problem 4 (1v2-Cycle Problem)**.**

Input: Edges of a graph partitioned arbitrarily among the $n^{1-\varepsilon}$ machines.

Promise: The graph is either a cycle of length n, or two disjoint cycles, each of length $\frac{n}{2}$.

Goal: Determine whether the graph consists of one cycle or two disjoint cycles.

There exists an $O(\lg n)$-round algorithm for any fixed ε.

Exercise: Write such an algorithm and prove its correctness.

Improving this logarithmic-round algorithm has been a major open problem in the study of MPC algorithms. In particular, the following is a widely believed conjecture.

Conjecture 4 (Logarithmic round 1v2-Cycle conjecture)**.** *Let $\epsilon > 0$ be fixed. Suppose that we have $n^{1-\epsilon}$ machines each with $s \leq n^{\epsilon}$ space, then the 1v2-Cycle problem requires $\Omega(\lg n)$ rounds to solve.*

There is also a slightly stronger variant of the conjecture above, which even allows say n^{100} machines.

Conjecture 5 (Logarithmic round 1v2-Cycle conjecture)**.** *Let $\epsilon > 0$ be fixed. Suppose that we have $n^{O(1)}$ machines each with $s \leq n^{1-\Omega(1)}$ space, then the 1v2-Cycle problem requires $\Omega(\lg n)$ rounds to solve.*

Currently, the best known lower bound is $\Omega(\frac{1}{\varepsilon})$ rounds as in the previous problem. Moreover, proving that any problem in P admits a lower bound better than $\Omega(\frac{1}{\varepsilon})$ implies $\mathrm{NC}^1 \neq \mathrm{P}$. For more information on lower bounds see (Roughgarden, Vassilvitskii, & Wang, 2018) [1].

References

[1] T. Roughgarden, S. Vassilvitskii, and J. R. Wang. Shuffles and circuits: (on lower bounds for modern parallel computation). *Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures*, 2016. 5

[2] G. Yaroslavtsev and A. Vadapalli. Massively parallel algorithms and hardness for single-linkage clustering under ℓ_p-distances. *CoRR*, abs/1710.01431, 2017.