Algorithm Design for Big Data Systems

Divide and conquer

Graph algorithms

Streaming algorithms

Embarrassingly Parallel Problems

- Problems that can be easily decomposed into many independent problems.
- Examples.
  - Word count
  - k-means
  - PageRank

Divide and Conquer

code at:

https://www.cse.ust.hk/msbd5003/nb/DC.ipynb

Classical Divide-and-Conquer

- Classical D&C
  - Divide problem into 2 parts
  - Recursively solve each part
  - Combine the results together
- D&C under big data systems
  - Divide problem into p partitions, where (ideally) p is the number of executors in the system
  - Solve the problem on each partition
  - Combine the results together
- Example: sum(), reduce()

Prefix Sums

- Input: Sequence *x* of *n* elements, binary associative operator +
- Output: Sequence *y* of *n* elements, with $y_k = x_1 + \ldots + x_k$
- Example:

  x = [1, 4, 3, 5, 6, 7, 0, 1]

  y = [1, 5, 8, 13, 19, 26, 26, 27]
- Algorithm:
  - Compute sum for each partition
  - Compute the prefix sums of the p sums
  - Compute prefix sums in each partition

Variants of Prefix Sums

- Assign consecutive id’s for each element
  - zipWithIndex()
- Given a list of words, find the first appearance of “spark”
- Given two long strings, compare them lexicographically
- Given a sequence of integers, check whether these numbers are monotonically decreasing.

Sorting (Sample Sort)

- Step 1: Sampling
  - Master node collects a sample of sp elements (will determine s later)
- Step 2: Choose splitters
  - Pick every $(i \cdot s)$-th element in the sample as splitters, $i = 1, \dots, p-1$
  - Broadcast them to all machines
- Step 3: Shuffling
  - Each machine partitions its data using the splitters
  - Send data to the target machine
- Step 4: Sort each partition
  - Each machine sorts all data received

Probability Tools

- Chernoff inequality
  Theorem: Let $X_1, X_2, \dots, X_n$ be independent 0-1 random variables (not necessarily identical), and let $X = \sum_i X_i$ with $\mu = E[X]$. Then for any $0 \le \delta \le 1$,

$$\Pr[X \le (1-\delta)\mu] \le \exp\left(-\frac{\mu\delta^2}{2}\right)$$

$$\Pr[X \ge (1+\delta)\mu] \le \exp\left(-\frac{\mu\delta^2}{3}\right)$$

- Union bound. Given events $E_1, \dots, E_n$, independent or not,

$$\Pr\left[\bigcup_{i=1}^{n} E_i\right] \le \sum_{i=1}^{n} \Pr[E_i]$$

Determining Sample Size

- Goal: No machine receives more than $(1+\epsilon)\frac{N}{p}$ elements w.h.p.
  – How large should s be?
- Let the elements be $a_1, \dots, a_N$ in sorted order
- A sub-sequence $a_i, \dots, a_{i+(1+\epsilon)\frac{N}{p}-1}$ is bad if it contains $< s$ sampled elements
  – Goal achieved if no sub-sequence is bad
- Consider a particular sub-sequence
  – $X = \#$ sampled elements in it; $E[X] = \frac{sp}{N} \cdot (1+\epsilon)\frac{N}{p} = (1+\epsilon)s$
  – By Chernoff inequality:
  $$\Pr[X < s] \le \Pr\left[X < \left(1 - \frac{\epsilon}{2}\right)E[X]\right] \le e^{-\epsilon^2 s/8}$$
- By union bound, $\Pr[\exists \text{ a bad subsequence}] \le N \cdot e^{-\epsilon^2 s/8}$
  – If want failure probability 1%, suffices to set $s = \frac{8\ln(100N)}{\epsilon^2}$
  – A tighter analysis improves the $\log N$ term to $\log\frac{p}{\epsilon}$.

Distributed Sampling

- Q: How to sample one element uniformly from n elements stored on p servers?
- A:
  - First randomly sample a server
  - Then ask that server to return an element randomly chosen from its n/p elements.
  - The probability of each element being sampled is $\frac{1}{p} \cdot \frac{p}{n} = \frac{1}{n}$
- Q: How to sample many elements at once?
- A: Do each of the two steps above in batch mode
  - First sample sp servers with replacement (this can be done at the master node).
  - If a server is sampled k times, we ask that server to return k samples (with replacement) from its local data.

The Maximum Subarray Problem

Input: Profit history of a company of the years.

Year	1	2	3	4	5	6	7	8	9
Profit (M$)	-3	2	1	-4	5	2	-1	3	-1

Problem: Find the span of years in which the company earned the most

Answer: Year 5-8 , 9 M$

Formal definition:

Input: An array of numbers $A[1 \dots n]$, both positive and negative

Output: Find the maximum $V(i,j)$, where $V(i,j) = \sum_{k=i}^{j} A[k]$

A divide-and-conquer algorithm

Year	1	2	3	4	5	6	7	8	9
Profit (M$)	-3	2	1	-4	5	2	-1	3	-1

Idea:

Cut the array into two halves

All subarrays can be classified into three cases:

- Case 1: entirely in the first half
- Case 2: entirely in the second half
- Case 3: crosses the cut

Largest of three cases is final solution

The optimal solutions for case 1 and 2 can be found recursively.

Only need to consider case 3.

Solving case 3

Year	1	2	3	4	5	6	7	8	9
Profit (M$)	-3	2	1	-4	5	2	-1	3	-1

Idea:

Let $q = \lfloor (p+r)/2 \rfloor$

Any case 3 subarray must have starting position $\leq q$, and ending position $\geq q+1$

Such a subarray can be divided into two parts $A[i..q]$ and $A[q+1..j]$, for some i and j

Just need to maximize each of them separately

Maximize $A[i..q]$ and $A[q+1, j]$:

Let i’, j’, be the indices that maximize the values.

i’, j’ can be found using linear scans to left and right of q

A[i’,j’] has largest value of all subarrays that cross q

The (binary) divide-and-conquer algorithm

```
MaxSubarray(A,p,r):
if p = r then return A[p]
q ← ⌊(p + r)/2⌋
M_1 ← MaxSubarray(A,p,q)
M_2 ← MaxSubarray(A,q + 1,r)
L_m ← −∞, R_m ← −∞
V ← 0
for i ← q downto p
      V ← V + A[i]
      if V > L_m then L_m ← V
V ← 0
for i ← q + 1 to r
      V ← V + A[i]
      if V > R_m then R_m ← V
return max{M_1, M_2, L_m + R_m}

First call: MaxSubarray(A,1,n)
```

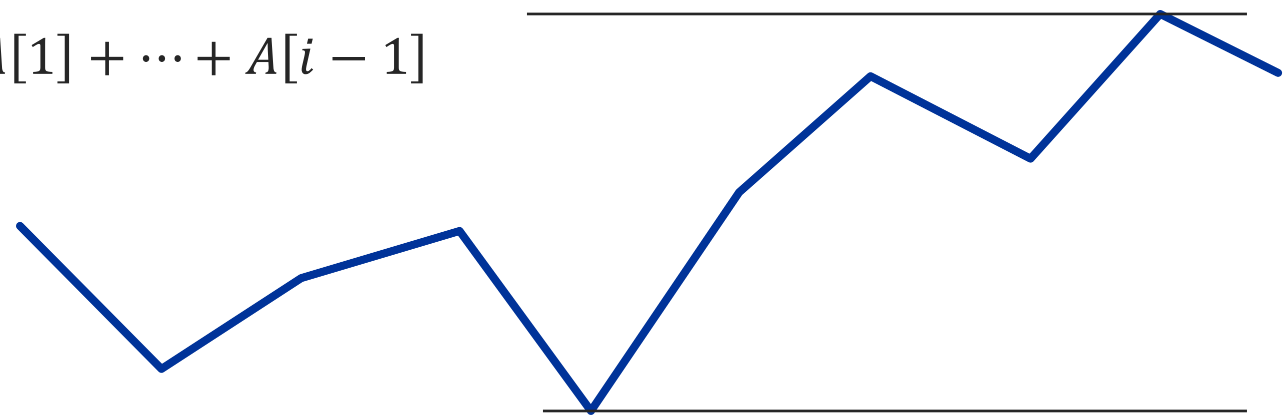
- Analysis:
  - Recurrence: $T(n) = 2T(n/2) + n$
  - So, $T(n) = \Theta(n \log n)$

If we use the same algorithm on Spark:

- Level 1:
  - Naively: 2 executors are working, all others idle
  - time = $O(n/2)$
  - Smarter: L_m and R_m can be found by the prefix-sum algorithm
  - Can use all executors, time = $O(n/p)$
- Level 2:
  - We have 4 subarrays, and solve two prefix-sums for each subarray
  - Each subarray has size $n/4$, and we make sure that each has the same number of partitions
  - Time = $O(n/p)$
- Level 3: Time = $O(n/p)$
- Stop recursion when each subarray is one partition.
- Total time: $O\left(\frac{n}{p} \cdot \log p\right)$

A linear-time algorithm?

- Define: $X[i] = A[1] + \cdots + A[i-1]$



Year	1	2	3	4	5	6	7	8	9
Profit (M$)	-3	2	1	-4	5	2	-1	3	-1
$X[i]$		-3	-1	0	-4	1	3	2	5

Observations:

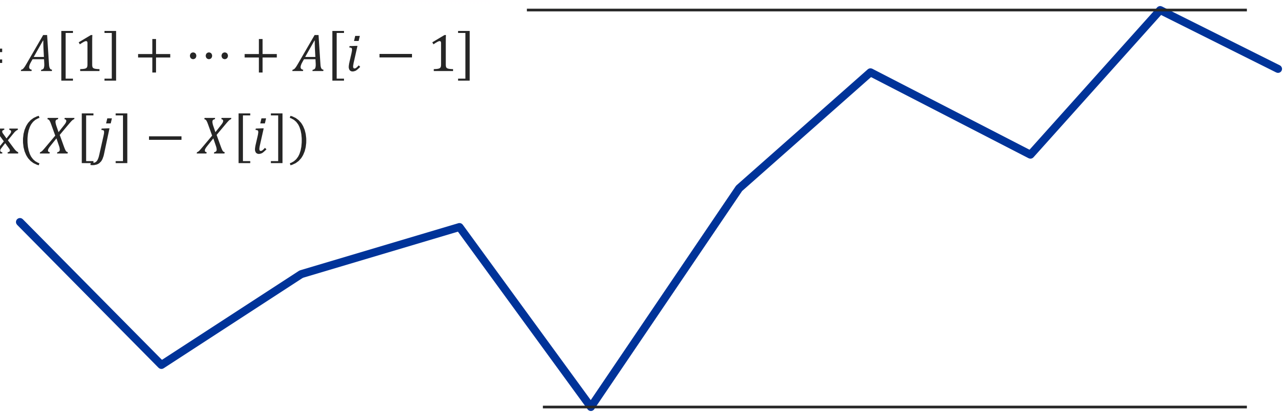
$V(i, j-1) = \sum_{k=i}^{j-1} A[k] = X[j] - X[i]$

For fixed j, finding largest $V(i, j-1)$ is same as finding the index i, $i < j$ for which $X[i]$ is smallest

Idea: doing this for each j, then find overall largest $V(i, j)$

A linear-time algorithm?

- Define: $X[i] = A[1] + \cdots + A[i-1]$
- Goal: Find $\max_{i<j}(X[j] - X[i])$



Year	1	2	3	4	5	6	7	8	9
Profit (M$)	-3	2	1	-4	5	2	-1	3	-1
$X[i]$		-3	-1	0	-4	1	3	2	5

Algorithm:

For each j, needs to know i < j that minimizes $X[i]$ (i.e., maximizes $X[j] - X[i]$)

- (Then maximize over all j)

Algorithm increases j by +1 each step

Keeps track of smallest $X[i]$ so far

- Could be old smallest one or it could be current $X[j]$

The linear-time algorithm

```
V_max ← −∞, X_min = 0
X ← 0, V ← 0
for i ← 1 to n do
    V ← V + A[i]
    if V > V_max then V_max ← V
    X ← X + A[i]
    if X < X_min then
        X_min ← X
        V ← 0
return V_max
```

X_{min} keeps track of smallest $X[i]$ so far.

V contains difference between current $X[i]$ and smallest $X[i]$ so far

Even “simpler”:

```
V_max ← −∞, V ← 0
for i ← 1 to n do
    V ← V + A[i]
    if V > V_max then V_max ← V
    if V < 0 then V ← 0
return V_max
```

- Observation:
  - $X < X_{min}$ iff $V < 0$
    - Because $V = X - X_{min}$
  - No need to actually store X!

A more efficient algorithm

- For each partition, solve the problem directly using one executor and the linear-time algorithm
- Now it remains to solve the “cross the boundary” case
- Find the L_m and R_m for each partition, as well as its sum
- For each contiguous subsets of partitions $(i, j), i < j$, the optimal solution with left boundary in partition i and right boundary in partition j is
$$L_m[i] + Sum[i+1, \dots, j-1] + R_m[j]$$
- Total time: $O(n/p)$