CS5234: Week 10

Nov. 1, 2018

Parallel Algorithms

Background: Moore's Law

→ every X years, # transisters/chip doubles

→ more transisters ⇒ more speed!

Moral: Clock speeds are not increasing
instructions / cycle growing rapidly

If we want to process big data fast, do it in parallel!!

In CS5234: Sampling → lots of parallelism
Sketches → lots of parallelism
Cache-efficient? many open questions

Types of Parallelism:
1) multicore
2) multisocket
3) Clusters / data centers
4) distributed networks

Different settings ⇒ different costs
⇒ different solutions

Today: multicore / multisocket
Next week: Clusters

How to model / program?

1) PRAM: p processors (where $p \to \infty$)

shared memory

program each proc. separately

Ex. Given $A[1, n]$, return true if all $A[j] = 0$

false otherwise

running on proc. $j \in [1, p]$

$\text{AllZero}(A, 1, n, p)_j$

for $i = \left(\frac{n}{p}\right)(j-1)+1$ to $\left(\frac{n}{p}\right) j$

someone initialized answer=true??

if $A[i] \neq 0$ then answer = false

done = done + 1

race condition? Use a lock?

Wait until done = p

return answer

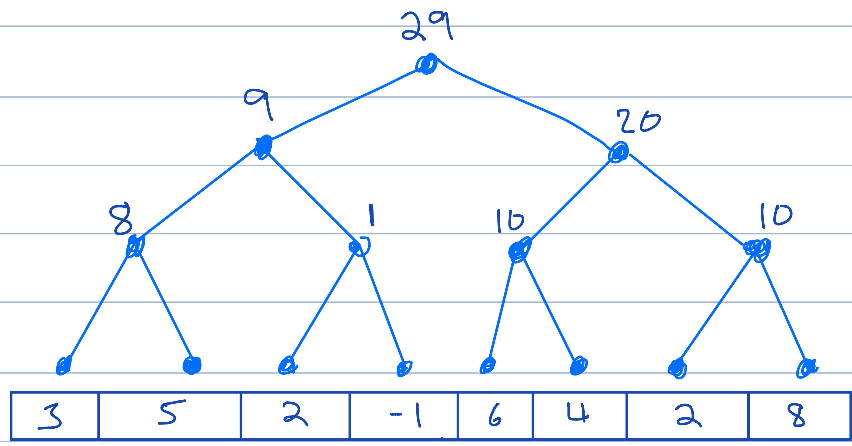
→ must carefully manage process interactions

→ manually divide problem among procs, or coordinate

→ low-level way to design parallel algorithms

Another example: sum an array (without locks)

Idea: a tree



RandomSum: Repeat until root is not empty
1. Choose a random node in tree
2. If both children are not empty then fill in node.

Claim

Finishes in $\Theta\left(\frac{n \log n}{p} + \log n\right)$ (Non-trivial example of coupon collector.)

Or: Statically assign procs to nodes in tree

Each proc computes all assigned nodes,

In order from deepest to highest.

$\Rightarrow O\left(\frac{n}{p} + \log n\right)$

why $\log(n)$?

→ messy to express algorithm

Alternate approach: (For any # procs.)

```
SUM(A, B, E)
    if (B = E) return A[B]
    mid = ⌊(B+E)/2⌋
    in parallel:
        ① L = SUM(A, B, mid)
        ② R = SUM(A, mid+1, E)
    return L+R
```

Same tree calculation!

→ base case = leaves

→ each (L+R) calculates one node in tree

How fast? How many processors?

Which processor executes what?

If all executed by 1 processor:

$$T_1(n) = 2T_1(n/2) + O(1)$$
$$= O(n)$$

one proc per node in tree

If $p = \infty$ and each recursive call is executed by a different processor:

$$T_\infty(n) = T_\infty\left(\frac{n}{2}\right) + O(1)$$
$$= O(\log n)$$

both recursive calls in parallel take same time

On p processors? Depends on scheduler

Work = cost on 1 proc = total compute steps

Span = cost on ∞ proc = critical path
= longest sequential part of program

Fork-Join Model of parallelism

→ program can spawn parallel threads
  ↳ "fork"
  ↳ "in parallel"

→ program can wait until (all) parallel threads resolve ("join")

→ procs assigned to threads by scheduler

→ alg does not specify # procs

Implemented in most parallel programming platforms
(in Java, Intel Parallel Studio, Cilk, etc.)

Clean way to think about <u>algorithms</u>
(even if you implement it differently)

Schedulers

Claim: Greedy scheduling finishes a fork-join program with work W and span S on p procs in time:

$$O\left(\frac{W}{p} + S\right)$$

Greedy: schedule each proc to a thread, if any available. Otherwise, do nothing.

Why? $\leq W/p$ steps when all procs busy
$\leq S$ steps when not all procs busy
make progress on critical path

just as good as $p = \infty$!

Also: Provably-efficient work-stealing schedulers are fast and also guarantee $O(W/p + S)$ time.

Time for SUM: $O(n/p + \log n)$

How many procs do you need?

$T_p \geq S, \quad T_p \geq W/p$

if $p = W/S$, then $T_p = O\left(\frac{W}{W/S} + S\right) = O(S)$ ← optimal

$$\boxed{\text{Parallelism} = W/S}$$

Data Structure: Set

→ insert, delete: $W=O(\log n)$, $S=O(\log n)$

→ divide: $W=O(\log n)$, $S=O(\log n)$

↳ divide set into two pieces, approximately the same size: $|S_1| \geq |S|/c$, $|S_2| \geq |S|/c$

→ union: $W=O(n+m)$, $S=O(\log n)$

↳ combines 2 sets, removing duplicates

set sizes: n, m, where $n \geq m$

→ set difference: $W=O(n+m)$, $S=O(\log(n))$

→ set minus: $W=O(n+m)$, $S=O(\log n)$

→ intersection: $W=O(n+m)$, $S=O(\log n)$

Union/intersection are expensive because must look at all elements → duplicates!

Implement a set as a balanced binary tree

↳ e.g., a red-black tree or a treap

Support:

1) $Split(T, K) \rightarrow T_1$ = items in $T < K$

T_2 = items in $T > K$

$X = K$ if $K \in T$, $\perp$ otherwise

2) $Join(T_1, T_2) \rightarrow T$ Note: Union

if $\max(T_1) < \min(T_2)$ ← does not require

[Hint: split does tree walk searching for key, dividing tree into many subtrees that are joined to form T_1, T_2. join pastes trees and balances]

For both: $W = O(\log n + \log m)$ n, m = size of

$S = O(\log n + \log m)$ trees T_1, T_2

insert/delete: normal, sequential

divide: find median: K (or choose random median

$Split(T, K)$ or use root to split if weight balanced)

Union(T_1, T_2)

if T_1 = null return T_2

if T_2 = null return T_1

Key = root T_1

(L, G, X) = split(T_2, Key) ← cost: $O(h_2)$

R = root T_1

in parallel:

1) T_L = Union(R.left, L)

2) T_R = Union(R.right, R)

T = join(T_L, T_R) ← cost: $O(h_1+h_2)$

if $X \neq \perp$ then insert(T, X) ← cost: $O(h_1+h_2)$

return T

Analysis: h_1 = height T_1, h_2 = height T_2

$$S(h_1, h_2) \leq S(h_1-1, h_2) + O(h_1+h_2)$$
$$= O(h_1(h_1+h_2))$$
$$= O(\log^2 n + \log n \log m)$$
$$= O(\log^2 n) \text{ if } n > m$$

More optimization: $O(\log n)$

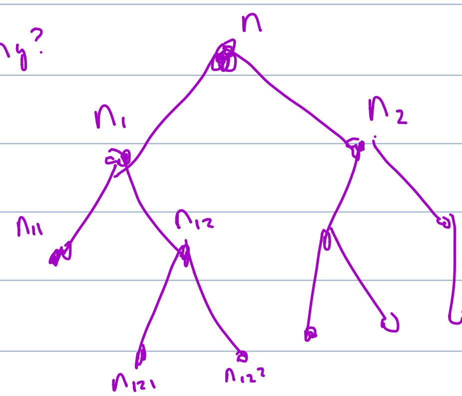
$$W(n,m) = W(n_1, m_1) + W(n_2, m_2) + O(\log n + \log m)$$

$$\leq W(n_1, m) + W(n_2, m) + O(\log n + \log m)$$

$n_1 + n_2 = n$

$$= O(n \log m)$$

Why?



Optimized: $O(m \log \frac{n}{m})$

↑ smaller tree

Recurrence tree $= T_1$

Assigns cost $\log m + \log(n_j)$

to node n_j

$\Rightarrow$ total cost: $O(n \log m)$

Intersection: if root of T_1 is in T_2: include root T_1

split T_2 by T_1's root

Recursively find difference $T_1.L \oplus L$, $T_1.R \oplus R$

Join 2 results

if include root T_1: insert root T_1

Difference: if root of T_1 not in T_2: include root T_1

split T_2 by T_1's root

Recursively find difference $T_1.L \oplus L$, $T_1.R \oplus R$

Join 2 results

if include root T_1: insert root T_1

SetMinus:

BFS

$F \leftarrow \{s\}$

repeat until $F = \emptyset$:

  $F' = \emptyset$

  for each $u \in F$:

    visited[u] = true

    for each nbr v of u:

      if visited[v] = false

        then F'.add(v)

  $F = F'$

Parallel BFS

→ Process frontier in parallel

→ assume u.nbrs is a set

  [Ex: build sets in parallel from adjacency list]

→ Use set operations to manipulate frontier F, visited

ParBFS(G, s)

  $F = \{s\}$

  $D = \emptyset$

  repeat until $F = \emptyset$:

    D = Union(D, F) ← mark frontier done

    F = ProcessFrontier(F)

    F = SetMinus(F, D)

ProcessFrontier(F)

if $|F|=1$ then: u = node in F

return u.nbrs ← set

$\langle F_1, F_2 \rangle$ = Divide(F)

in parallel:

1) F_1 = ProcessFrontier(F_1)

2) F_2 = ProcessFrontier(F_2)

return Union(F_1, F_2)

Analysis:

For ProcessFrontier on n nodes with m nbrs:

$$W(n,m) \le 2W\left(\tfrac{n}{2}, m\right) + O(m \log m) + O(\log n)$$
$$= O(m \log n \cdot \log m)$$

$$S(n,m) = S(n/2, m) + O(\log n) + O(\log^2 m)$$
$$= O(\log^2 m \log n)$$

For BFS: $\text{Work} \leq \sum_j W(F_j, m) + O(m \log m)$

$$\leq O(m \log^2 n) \qquad (m \geq n), \; (m < n^2)$$

$$\text{Span} \leq \sum_j S(F_j, m) + O(\log^2 m)$$

$$\leq O(D \log^2 m \log n)$$

$$= O(D \log^3 n)$$

D = diameter

Conclusion: ParBFS on p procs runs in time:

$$O\left(\frac{m \log^2 n}{p} + D \log^3 n\right)$$

Note: $\Omega(D)$ is almost inevitable

Only good if $p > \log^2 n$ [improvements exist]

DFS is, as best we know, $\Omega(n)$

→ use BFS, not DFS