Cost Graph

series - parallel, acyclic

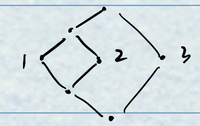
Base case) • (single node, source = sink) modeling no computation

Sequential Composition) $G_1 \downarrow G_2$ (edge from G_1's sink to G_2's source) modeling sequential computation

⁝ no computation

Parallel) source G_1 G_2 sink (fork & Join)

(1 + 2) * 3



Work = 7 Span = 5

Work : # of nodes in G

Span : # of nodes on the longest path from G's source to G's sink

Brent's Theorem

An expression e with W, S can be evaluated on a p-processor machine in time $\Omega(\max(W/P, S))$

like Big-O but lower-bound (≅)

Scheduling

pebbling p pebbles (p = # of processors)

Sequence

linear structure like lists, but support the parallelism of trees.

SEQUENCE signature

< x_0, ..., x_{n-1} > a sequence (length n)

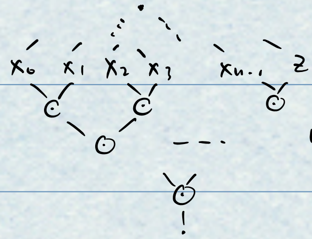
externally equiv { equal length; ≅ value at corresponding pos

```
signature SEQUENCE =
sig
    type 'a seq (* abstract *)
    exception Range of string
    val empty : unit -> 'a seq                    ; O(1)
    val tabulate : (int -> 'a) -> int -> 'a seq   f(i) O(1)  G0 G1 ... Gn-1 => W = O(n), S = O(1)
    val length : 'a seq -> int                    ; W & S = O(1)
    val nth : 'a seq -> int -> 'a                 0 ≤ i ≤ n, else raise Range   ; W & S = O(1)
    val map : ('a -> 'b) -> 'a seq -> 'b seq      same as tabulate
    val reduce : ('a * 'a -> 'a) -> 'a -> 'a seq -> 'a     W = O(n)  S = O(log n)
    val mapreduce : ('a * 'b) -> 'b -> ('b * 'b -> 'b) -> 'a seq -> 'b    same as reduce
    val filter : ('a -> bool) -> 'a seq -> 'a seq    If p has O(1) W & S => W = O(n)  S = O(log n)
    ...
end
```

type: t seq   client write: eg. t Seq.seq

tabulate f n ≅ < f(0), ..., f(n-1) >

reduce g z < x_0, ..., x_{n-1} > ≅ $x_0 \oplus x_1 \cdots x_{n-1} \oplus z$ g is associative infix representing g $\quad g(g(x,y),z) \cong g(x, g(y,z))$



O(log n) levels

Work : O(n)

Span : O(log n)

<u>mapreduce</u> f z g < x0, ..., xn-1 > ≅ (f x0) ⊙ ... ⊙ (f xn-1) ⊙ z

<u>filter</u> implementation vary @ using mapreduce => Work = O(n log n)

Span = O(log n)

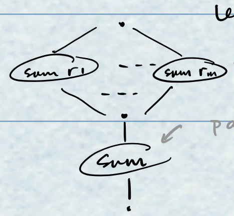
```
fun sum (s : int Seq.seq) : int = Seq.reduce (op +) 0 s
type row = int Seq.seq
type room = row Seq.seq  -> int Seq.seq Seq.seq
fun count (class : room) : int = sum(Seq.map sum class)
```

m rows, n students each

Work : O(mn)

Span : O(log n + log m)

let c = < row1, ..., rowm >



each contains n integers

passed-in argument contains m integ

alt : val count : room -> int =