Parallelism

15-150 Lec 2, Frank Pfenning
Lecture 19
Tuesday April 7, 2020

Outline

- Cost Graphs as an abstract model of parallel evaluation
- Brent's Theorem for evaluation on p processors
- Sequences as a high-level abstraction for parallel programming
- Example: Parallelizing n-Queens

Deterministic Parallelism

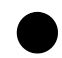
- Outcome of computation is uniquely determined
- But not exactly how the computation proceeds
- Enabled by
    - Pure functions (no sharing, “race conditions”)
    - High level abstraction: sequences
    - Complex compiler and runtime system

Cost Graphs

- Cost Graphs model the interactions between sequential and parallel computation
- Help us visualize
  - Work and span of a computation
  - Scheduling of computation (on a fixed number of processors)
- A cost graph is a directed acyclic graph with a source and a sink
  - We show the source (beginning of computation) at the top
  - We show the sink (end of computation) at the bottom

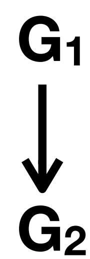
Cost Graph Constructions

Expression or Value



(single node = source = sink)

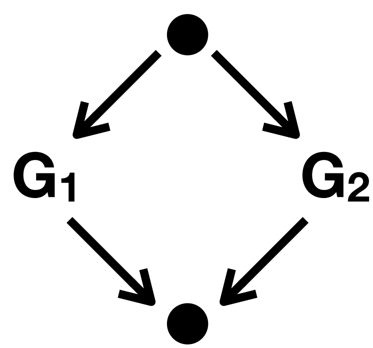
Sequential Composition



(edge from sink of G_1 to source of G_2)

(first G_1, then G_2)

Parallel Composition

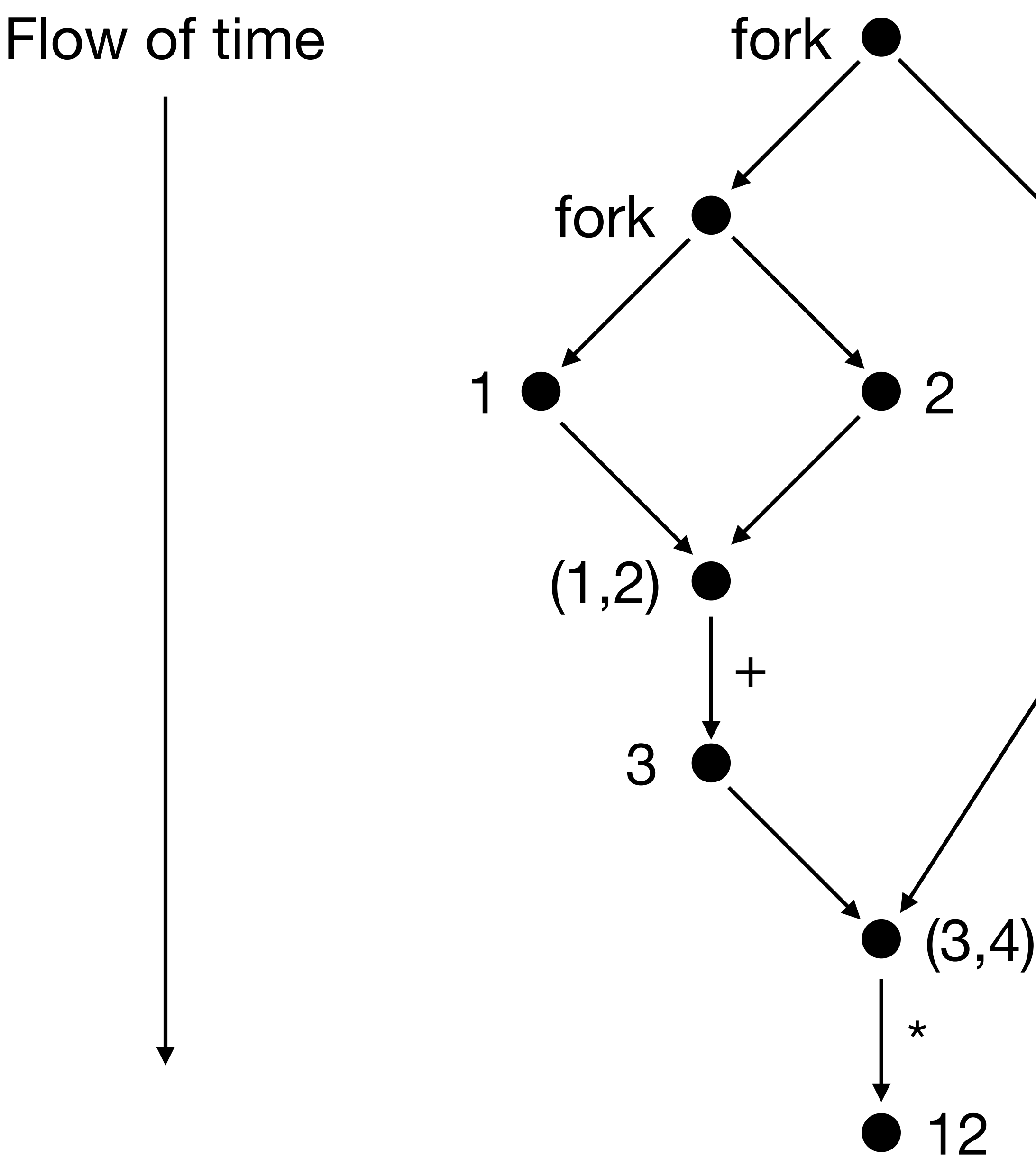


(new source and sink)

(G_1 and G_2 in parallel)

fork / join

Example: (1 + 2) * 4



Work = total number of nodes

Here: 9

Span = longest path from source to sink

Here: 7

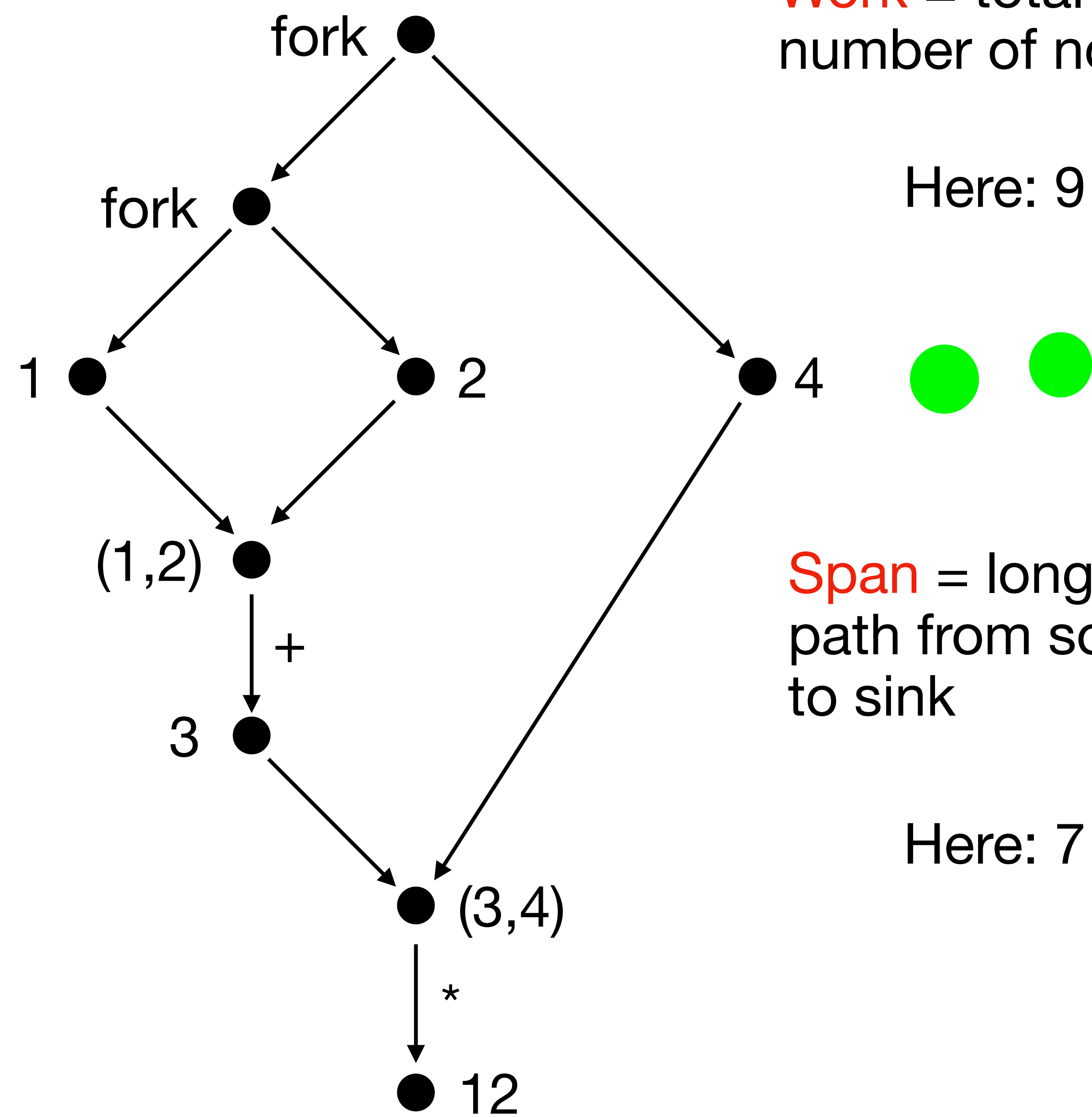
Brent’s Theorem

Theorem: An expression e with work W and span S can be evaluated on a p-processor machine in time

$$O(\max(W/p, S))$$

Scheduling with Pebbles

Flow of time



Work = total number of nodes

Here: 9

Span = longest path from source to sink

Here: 7

Sequences

- The Sequence library provides an abstract type of `‘a seq`
- Write sequences as

$$\langle x_0, \ldots, x_{n-1} \rangle$$

- There are multiple different implementations (array-like, tree-like)
- Functions on sequences are given with the cost graphs / work and span
    - Not all implementations may achieve those exactly
    - Also remember Brent’s Theorem!
- Idea: functions in the library are parallel to the extent possible!

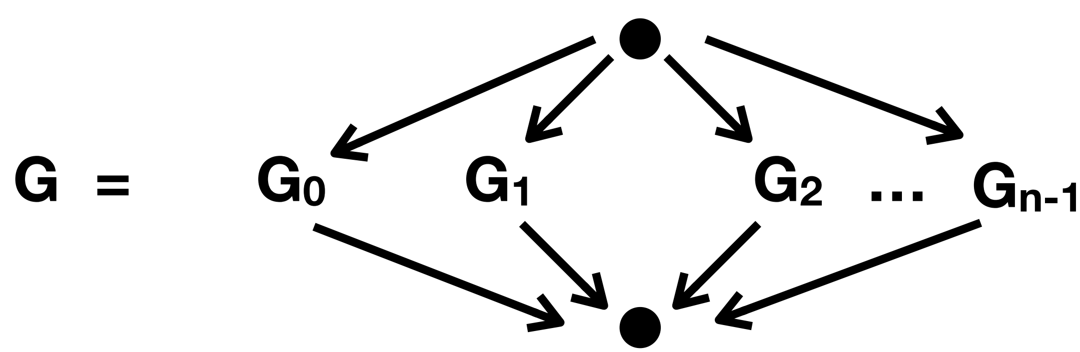
```
signature SEQUENCE =
sig
  type ‘a seq  (* abstract)
  val tabulate : (int -> ‘a) -> int -> ‘a seq
  val map : (‘a -> ‘b) -> ‘a seq -> ‘b seq
  val reduce : (‘a * ‘a -> ‘a) -> ‘a -> ‘a seq -> ‘a
  (* lots more stuff *)
end
```

`tabulate`

`tabulate f n == <f(0), ... , f(n-1)>`

- Cost graph

G_i is the cost graph for evaluating `f(i)`



Work(G) = Sum(Work(G_i)) + 2

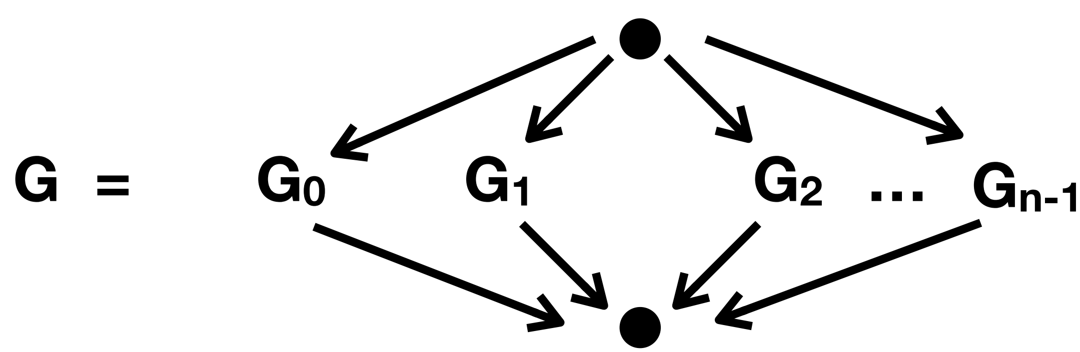
Span(G) = Max(Span(G_i)) + 2

map

$\mathtt{map\ f}\ <x_0,\ldots,x_{n-1}> \ == \ <\mathtt{f}\ x_0,\ \ldots\ ,\ \mathtt{f}\ x_{n-1}>$

- Cost graph

G_i is the cost graph for evaluating $\mathtt{f}\ x_i$



Work(G) = Sum(Work(G_i)) + 2

Span(G) = Max(Span(G_i)) + 2

reduce

$$\texttt{reduce g z } \langle x_0,\ldots,x_{n-1}\rangle \texttt{ == } x_0 \odot \ldots \odot x_{n-1}$$

$$\texttt{reduce g z } \langle\ \rangle \texttt{ == z}$$

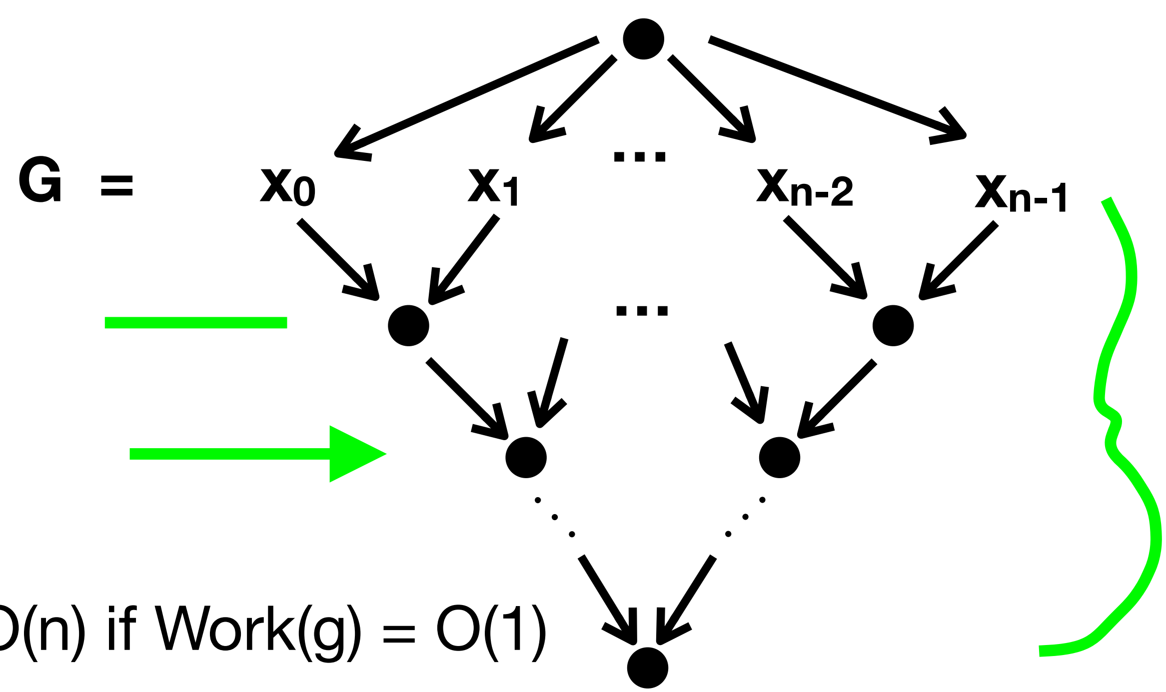
- Here, g must be an associative operator with unit z (“requires”)
  - g(g(w,x),y) == g(w,g(x,y))
  - g(x,z) == x
- Write $x \odot y = g(x,y)$

reduce

$$\texttt{reduce g z <}x_0, \ldots, x_{n-1}\texttt{>} == x_0 \odot \ldots \odot x_{n-1}$$

$$\texttt{reduce g z < > == z}$$

- Cost graph



Work(G) = O(n) if Work(g) = O(1)

Span(G) = O(log(n)) if Span(g) = O(1)

Summary

- Cost Graphs as an abstract model of parallel evaluation
- Brent's Theorem for evaluation on p processors
- Sequences as a high-level abstraction for parallel programming
- Example: Parallelizing n-Queens