Lecture 19 Sequences (Cost Graphs)

April 3, 2018

dag: Cars don't run into each other!

Cost Graphs

Cost Graph: Parallel Series, DAG with single source & sink.

Example:

```
(1 + 2) + 3

          _
         / \
        _   3
       / \  |
      1   2 |
        V   |
        _   /
          V

          _
```

Base Case	Serial Computation	Parallel Computation
Single Node (sink = source)	o-o	See Below

```
    _
   / \
  _   _
    V

    _
```

Work is Number of Nodes **Span** is Length of the Longest Path (Count Edges)

For example, in the above `(1 + 2) + 3` instance, `W = 7` and `S = 4`.

Brent's Theorem: Time to perform a computation is $O(max(\frac{W}{p}, S))$ where p is the number of processors.

Example:

```
(1 + 2) + (3 + 4)

         A
     B       G
   C   D   H   I
     E       J
         F

(Constraint: Computations below need results from Nodes above.)

Time / Processors (#1, #2)
-----+--------------------
  1  |  A
  2  |  B  G
  3  |  C  D
  4  |  H  I
  5  |  E  J
  6  |  F
```

$$max(\frac{W}{p}, S) = max(\frac{10}{2}, 4) = 5$$

Note: The particular structure of this graph does not allow the actual optimal to be achieved.

Sequences

Question: What is the cost of finding the length of a list? How can we do better? If we define our own type?

- Sequences are like `list` as we can walk through it sequentially.
- Sequences are like `tree` as it has `log` properties.

Notation: $< x_0, ..., x_{n-1} >$

Implementation

```
signature SEQ =
sig
  type 'a seq (* abstract *)

  val empty : unit -> 'a seq (* notation: <> *)
  (* We need this so that in the same code we can have 'a seq, 'b seq, so on at the same time. *)
  (* In addition, the initialization function allows us to do stuff behind the scene. *)

  (* Side Note
   * We cannot pattern match directly on sequences.
   * We can however pattern match with 'list view' & 'tree view' of sequences.
   *)

  exception Range (* like 'index out of range' *)

  val tabulate : (int -> 'a) -> int -> 'a seq
  val length : 'a seq -> int
  val nth : 'a seq -> int -> 'a
  val map : ('a -> 'b) -> 'a seq -> 'b seq (* like 'map' for List *)
```

```
    val reduce : ('a * 'a -> 'a) -> 'a -> 'a seq -> 'a (* like 'fold' but has a more stricted type *)
    val mapreduce : ('a -> 'b) -> 'b -> ('b * 'b -> 'b) -> 'a seq -> 'b
    val filter : ('a -> bool) -> 'a seq -> 'a seq

    (* etc. *)
  end
```

Analysis of the Implementation above with Cost Graphs

- `empty () = <>` : constant work & span
- `tabulate f n =` $< f(0), ..., f(n-1) >$

Cost Graph for `tabulate` :

```
                [SOURCE]
 f(0) f(1) ... ... ... f(n-1)
                 [SINK]
```

Suppose `f` has constant work & span, then $W = O(n)$ and $S = O(1)$.

However, `tabulate` for `List` has work and span of both $O(n)$ because `List` has no random access.

- `nth <x0,...,xn-1> i = xi if 0 <= i <= n-1 | raise Range otherwise`
  - Cost Graph: `o-o`
  - $W = S = O(1)$
  - For `List` : $W = S = O(n)$
- `map f <x0,...,xn-1>` : same **cost graph** as `tabulate`

- Assuming `g` is **associative**
  - Think of `g` as an `infix` operator
- `reduce g z <x0,...,xn-1> = x0 * x1 *...* xn-1 * z` where $*$ is the infix operator defined by `g`

Side Note: Sometimes (e.g. in 210) we assume `z` is an identity for `g`, which means it does not matter where to put `z` in the sequence. This is powerful for writing efficient parallel code.

Cost Graph for `reduce` :

```
                          [SOURCE]
  _ _   _ _   _ _   _ _   ...   ...   _ _
   V     V     V     V       ...       V
   _     _     _     _       ...       _
      V           V                 V
            V             ...
                     [SINK]
```

$$W = O(n)$$

$$S = O(log(n))$$

- `mapreduce f z g <x0,...,xn-1> = f(x0) * f(x1) *...* f(xn-1) * z`
- Work & Span is same as `reduce`

- `filter p <x0,...,xn-1> = <all xi s.t. p xi = true`
- $W = O(n)$ and $S = O(log(n))$ because there is some additional work (for span)

```
structure Seq :> SEQ = struct (* etc. *) end

fun sum (s : int Seq.seq) : int = Seq.reduce (op +) 0 s

type row = int Seq.seq
type room = row Seq.seq (* 2D sequence *)

fun count (class : room) : int = sum (Seq.map sum class)

fun count' (class : room) : int = Seq.mapreduce sum 0 (op +) class (* alternative implementation *)
```

For `count'`

```
                    [SOURCE]
   [sum R1]    [sum R2] ... ... ...   [sum Rn]
           V              V  ...
                ...
                [SINK]
```

$$S = O(log(n) + log(m))$$