Parallel Algorithms

Umut Acar
Carnegie Mellon University

July 10, 2018

1 Resources

15210 CMU
Sequences
Contraction
Divide and Conquer

2 Recap

Importance of having a clean semantics for parallelism.
Clean cost model.
Using lambda calculus for the cost model:

- Work and Span
- $T_p \sim \frac{W}{p}$
- Use scheduling algorithms to achieve: $T_p = \frac{W}{p} + S \leq 2 * OPT$
- $\frac{W}{p}$ is $O\left(\frac{n}{p}\right)$
- S is $O\left(log^3(n)\right)$

Goal: $\frac{W}{p} + S \sim \frac{W}{p}$
If $S \sim W$ this is a problem.

3 Designing Algorithms with Low Span

- Fundamental Data Structure: Sequences

3.1 Sequences

`Sequence` $a = (a_0, a_1, a_2, a_3, \ldots, a_n)$
$a[0] = a_0$ $a[i] = a_i$ `length`$(a) = |a|$ `subsequence`$(a, i, j) = a[i, \ldots, j]$
Indexing: $O(1)$ work and span
Length: $O(1)$ work and span
Subsequence: $O(1)$ work and span
`splitMid(a) = subseq(`$0, \ldots, \left\lfloor \frac{n}{2} \right\rfloor - 1$`),subseq(`$\left\lfloor \frac{n}{2} \right\rfloor, \ldots, n-1$`)`

3.1.1 tabulate

`tabulate:` `(int`$\to \alpha$`)`$\to$`int`$\to \alpha$ `seq`
`tabulate` $(\lambda i \to i)$ `n = (0,1,...,n-1)`
Work: $O(n)$, $\sum_i W(f(i))$
Span: $O(1)$, $Max_i\ S(f(i))$
Tabulate Usage:
`empty = tabulate` $(\lambda i.i)$ 0
`simplify = tabulate` $(\lambda i.e)$ 1
`map f a = tabulate` $(\lambda i.f(a[i]))$ $|a|$
`append a b = tabulate` $(\lambda i.$`if` $i < |a|$ `then` $a[i]$ `else` $b[i - |a|])$ $|a| + |b|$
For append work is $O(|a| + |b|)$ and span is $O(1) \Leftarrow$ Perfectly parallel.

3.1.2 iterate

`iterate:` $\beta \to ((\beta \times \alpha) \to \beta) \to \alpha$ `seq` $\to \beta$
Example:
$a = (a_0, a_1, \ldots, a_{n-1})$
`iterate b (`λ`(x,y).x+y) a`
`iterate 0 (`λ`(x,a`$_1$`).x+a`$_1$`) a`
$(\ldots((0+1)+2)+\ldots+(n-1))$
Now want a function that turns (1,0,4,5,2,0,0,3,4) into (0,1,1,4,5,2,2,2,3):
```
fun skipzero (x,y) = if x > 0 then x else y
```
Example (Insertion Sort):
```
insertionSort a = iterate () insert a
fun insert(x,r) = iterate ...
```
Work $= \sum_i W(f(x_i, a[i]))$
Span $= \sum_i S(\ldots) \leftarrow$ No parallelism.
Summing up a sequence a:
`iterate 0 (`λ`(x,y).x+y) a`
But, by taking advantage of associativity the operations can be reordered.
Correction for skipZero:
`iteratePrefixes:` $\beta \to (\beta \times \alpha \to \beta) \to \alpha$ `seq` $\to \beta$ `seq`

3.1.3 reduce

`reduce id f a`
`id` is identity for `f`

Should produce: `(f(f(f(id,a[0]),a[1]),a[2])`

```
fun reduce id f a = if |a| = 0 then id else if |a| = 1 then a[0] else
         let (b,c) = a[0,...,|a|/2-1], a[|a|/2,...,|a|-1]
            (rb,rc) = reduce id b || reduce id c
         in f (rb, rc) end
```

$W(n) = 2W\frac{n}{2} + O(1) = O(n)$
$S(n) = max(S(\frac{n}{2} + 0, 1)) = log(n)$

```
mergesort = reduce () merge (map singleton a)
fun reduce id f a =
   if |a| = 0 then 0
   else if |a| = 1 then a
   else
      b = tabulate (λi.f(a[2i],a[2i+1])) |a|/2
      reduce id f b
```

$W(n) = W(\frac{n}{2}) + O(n) = O(n)$
$S(n) = S(\frac{n}{2}) + 1 = O(log(n))$
So iterate is sequential and reduce is parallel.

3.1.4 scan

`scan id f a`
Gives a sequence:
(`reduce id f ()`
`reduce id f (a[0])`
`reduce id f (a[0],a[1])`
⋮
`reduce id f a`)
Example:
(1,0,2,7,0,5)
`reduce 0 skipzero ()`
`reduce 0 skipzero (1)`
⋮
`reduce 0 skipzero (1,0,2,7,0,5)`
and returns `((0,1,1,2,7,7),5)`
With sums this would return: ((0,1,1,3,10,10),15)