Parallel And Sequential Data Structures and Algorithms

Divide-and-Conquer and Reduce

Learning Objectives

- Understand **folds** and **reductions** as generalizations of sums
- See how to implement **reduce** using **divide-and-conquer**
- See how to implement **divide-and-conquer** using **reduce**
- Learn the divide-and-conquer algorithm for **merge**
- Understand the uses of merge as an **associative function**

Reduce

Recall: Parallel Sum

- In **Lecture One**, we saw this parallel divide-and-conquer algorithm for sum
- It runs in $O(n)$ work and $O(\log n)$ span

```
fun sum(S : sequence<int>) -> int:
  match length(S) with:
    case 0: return 0       // Empty sequence
    case 1: return S[0]    // Singleton sequence
    case _:
      L, R = split_mid(S) // Helper function
      Lsum, Rsum = parallel (sum(L), sum(R))
      return Lsum + Rsum
```

Question: How can we generalize this?

Generalizing Sums: Folds

- There are several ways we can generalize the notion of a sum

$$s_0 + s_1 + s_2 + \cdots + s_{n-1} + s_n$$

Definition (Fold): A (left) **fold** over some sequence s with a binary operation f and an initial value I computes

$$f(f(f(f(f(I, s_0), s_1), s_2), \ldots, s_{n-1}), s_n)$$

- Left/right folds are defined left-to-right or right-to-left, so they are inherently sequential! We can not hope to parallelize this.

Generalizing Sums: Reductions

- Folds are not parallelizable because they typically prescribe a particular evaluation order (e.g., left-to-right or right-to-left)
- A ***reduction*** is a fold that can be **arbitrarily** parenthesized

$$\left(s_0 + (s_1 + s_2)\right) + \cdots + (s_{n-1} + s_n)$$

Definition (Reduce): A **reduce** over some sequence s with a binary **associative** operation f and an **identity** value I computes a fold of f over s where the order of applications of f may be arbitrary

$$f\left(f\left(s_0, f(s_1, s_2)\right), \ldots, f(s_{n-1}, s_n)\right)$$

Reduce

- By relaxing the order, reduce becomes highly parallelizable
- The implementation matches our parallel sum, which took advantage of the associativity of + for integers

```
fun reduce(f : (T, T) -> T, I : T, S : sequence<T>) -> T:
  match length(S) with:
    case 0: return I
    case 1: return S[0]
    case _:
      L, R = split_mid(S)
      Lres, Rres = parallel (reduce(L), reduce(R))
      return f(Lres, Rres)
```

Understanding Reduce

```
reduce : (f : (T, T) -> T, I : T, S : sequence<T>) -> T
```

- `f` must be an **associative function.** Formally
  - `f(f(x,y),z) = f(x,f(y,z))` for all values `x,y,z` of type `T`
- `I` must be an **identity**, Formally
  - `f(I,x) = x` and `f(x, I) = x` for all values `x` of type `T`

Theorem (Cost of Reduce): Assuming that f can be evaluated in constant time, reduce costs $O(|S|)$ work and $O(\log |S|)$ span.

- Work recurrence is leaf dominated with $O(|S|)$ leaves

Look Familiar?

Recall from MCSSLab

```
fun smcss(S : sequence<int>) -> (int,int,int,int):
  match length(S) with:
    case 0: return (0,0,0,0)
    case 1:
      m = max(0, A[0])
      return (m, m, m, A[0])
    case _:
      L, R = split_mid(S)
      (m1,p1,s1,t1), (m2,p2,s2,t2) = parallel (smcss(L), smcss(R))
      return (max(s1 + p2, m1, m2),
              max(p1, t1 + p2),
              max(s2, t2 + s1),
              t1+t2)
```

Hang on, That's Just Reduce!

```
type sums = (int,int,int,int)

fun combine_smcss((m1,p1,s1,t1) : sums, (m2,p2,s2,t2) : sums) -> sums:
  return (max(s1 + p2, m1, m2),
          max(p1, t1 + p2),
          max(s2, t2 + s1),
          t1+t2)
```

```
fun smcss(S : sequence<int>) -> (int,int,int,int):
  fun base(x : int): return (max(0,x),max(0,x),max(0,x),x)
  return reduce(combine_smcss, (0,0,0,0), map(base, S))
```

Recall from Paren Match (Reci)

```
fun excessParens(p : sequence<Paren>) -> (int,int):
  match length(p) with:
    case 0: return (0,0)
    case 1:
      if p[0] == L: return (0,1)
      else:         return (1,0)
    case _:
      L, R = split_mid(S)
      (i,j), (k,l) = parallel (excessParens(L), excessParens(R))
      if j <= k: return (i + k - j, l)
      else: return (i, l + j - k)
```

It's Reduce Again!

```
fun combine_paren((i : int, j : int), (k : int, l : int)) -> (int, int):
  return (i + k - j, l) if j <= k else (i, l + j - k)
```

```
fun excessParens(p : sequence<Paren>) -> (int,int):
  fun base(x : Paren): return (0,1) if x == L else (1,0)
  return reduce(combine_paren, (0,0), map(base, p))
```

Reduce as "Generic D&C"

```
fun algo(S : sequence<T>):
  match length(S) with:
    case 0: return empty
    case 1: return base(S[0])
    case _:
      L, R = split_mid(S)
      Lres, Rres = parallel (algo(L), algo(R))
      return combine(Lres, Rres)
```

⇕

```
fun algo(s : sequence<T>):
  return reduce(combine, empty, map(base, s))
```

Reduce as "Generic D&C"

- Works when the "divide" step is trivial: just split input in half and recurse--don't do anything with the halves before recursion
- When the "conquer" step can be expressed as an associative function `combine` which combines the left and right result

```
fun algo(S : sequence<T>):
  match length(S) with:
    case 0: return empty
    case 1: return base(S[0])
    case _:
      L, R = split_mid(S)
      Lres, Rres = parallel (algo(L), algo(R))
      return combine(Lres, Rres)
```

Merge

Merge

- Recall the merge operation from 15-122 and 15-150:

Definition (Merge): Given two sorted sequences, return a sorted sequence containing the elements of both

```
fun merge(A : sequence<T>, B : sequence<T>) -> sequence<T>
```

- A simple sequential implementation runs in $O(n)$ time
- How can we make this parallel?

Answer: Let's try divide-and-conquer!

Divide-and-conquer merge

Question: Right off the bat, what makes merge more complicated than most divide-and-conquer algorithms?

- Input is *two sequences*, not one! May not be the same length!
- Which one(s) do we divide? What if we divide both in half?

`1,2,3,4,5,6,7,10`   `8,9,11,12,13,14,15,16`

Not sorted!

Finding the Split Point

- Let's still split the first sequence (`A`) in half

Question: Where should `B` be split?

Answer: Left side should have numbers less than `A[mid]` (i.e., 8)

`1,2,3,4,5,6,7` `8,9,10,11,12,13,14,15,16`

Making Merge Efficient

- Divide-and-conquer is efficient when splitting the input in halves
- We always split A in half, but B might be split badly

Question: How can we fix this?

Answer: Always split the *larger* of the two sequences

Question: How do we find the split point?

Answer: Use binary search!

Implementing Merge

```
fun merge(A: sequence<T>, B : sequence<T>) -> sequence<T>:
  if |B| > |A|: swap(A, B) // WLOG assume A is larger than B

  if |B| == 0: return A
  if |A| == |B| == 1: return [min(A[0], B[0]), max(A[0], B[0])]

  LA, RA = split_mid(A)
  k = binary_search(B, RA[0])  // k = smallest index such that B[k] >= RA[0]
  LB, RB = subseq(B,0,k), subseq(B,k,|B|-k)
  ML, MB = parallel (merge(LA, LB), merge(RA, RB))
  return append(ML, MB)
```

Oops: This implementation has an efficiency problem :(

Cost Analysis of Merge

Claim (Cost of Merge): This `merge` function costs $\Theta(n \log n)$ work

- *The primary issue is the* `append(ML, MB)` *step, which costs* $O(n)$
- *This makes the work recurrence* ***balanced****, so it solves to* $O(n \log n)$
- *i.e., the function does* $O(n)$ *work per level for* $O(\log n)$ *levels of recursion*

Fix: An efficient merge needs to write into a pre-allocated output array to avoid the expensive append (i.e., **impure**)

Efficient (Impure) Merge

```
fun merge(A: sequence<T>, B : sequence<T>, Out : mutable sequence<T>):
  if |B| > |A|: swap(A, B) // WLOG assume A is larger than B

  if |B| == 0: Out[0...|A|] ← A; return
  if |A| == |B| == 1: Out[0...1] ← [min(A[0], B[0]), max(A[0], B[0])]; return

  LA, RA = split_mid(A)
  k = binary_search(B, RA[0])  // k = smallest index such that B[k] >= RA[0]
  LB, RB = subseq(B,0,k), subseq(B,k,|B|-k)
  Lout, Rout = subseq(Out,0,|LA|+|LB|), subseq(Out,|LA|+|LB|,|RA|+|RB|)
  _, _ = parallel (merge(LA, LB, Lout), merge(RA, RB, Rout))
```

Note: You **can** write an $O(n)$ work pure merge, but you must store the input as balanced BSTs (they can be appended in $O(\log n)$ time!)

Cost Analysis of Efficient Merge

Claim (Cost of Merge): The efficient `merge` (impure) function costs $O(n)$ work and $O(\log^2 n)$ span, where $n = |A| + |B|$.

- $|A| \geq |B|$ and $|A|$ gets halved, so n in the recursive calls is $\in [0.25n,\ 0.75n]$
- Let $W(n)$ be the work of merge on an input of size n.

$$W(n) = W(\alpha n) + W\big((1-\alpha)n\big) + \Theta(\log n), \qquad \alpha \in [0.25, 0.75]$$

- Can verify with the **substitution method** that $W(n) \in O(n)$
- For the span, $S(n) \leq S(0.75n) + \Theta(\log n)$
- Unrolling this recurrence, it has $\log_{4/3} n$ levels, so $S(n) \in O(\log^2 n)$

Merge as an Associative Function

Claim (Merge is Associative): The Merge operation on two sorted sequences is an associative operation.

Note: `singleton(x)` returns `[x]`

- Therefore, we can reduce it!

```
reduce(merge, [], map(singleton, s))
```

```
reduce(merge, [], [[8],[5],[7],[4],[1],[3],[9],[2]])
```

Question: What algorithm is this?

Answer: Its MergeSort! In just one line of code :D

Final Thoughts

- Instead of reduce, what if we did a left fold?

```
fold_left(merge, [], map(singleton, s))
```

Question: What algorithm is this?

Answer: Its Insertion Sort!

Final Note: You *can* improve the span of merge to $O(\log n)$ but the algorithm is a little more complicated.

Summary

- **Reduce** is a **fold** over an **associative operation**
  - Applicable to fewer functions because of the associativity requirement
  - But as a result, highly parallel instead of completely sequential!
- Reduce can be implemented with **divide-and-conquer**
  - $O(|S|)$ work and $O(\log |S|)$ span assuming `f` is constant time
- **Divide-and-conquer** algorithms with trivial divide steps can be converted into reductions with a specific combine function!
- **Merge** can be implemented as a divide-and-conquer algorithm
- Efficiency sometimes requires impure code!