Scans and List Ranking

Introduction

1. Considering ways to extract parallelism from algorithms that, on the surface, don't lend themselves to parallel execution
2. List ranking: Given a singly-linked list and a pointer to the head, compute the position of every node in the list
    - Trivially sequentially, but more difficult in parallel
    - Traditional way of storing a linked list is wrong

Prefix Sums Definitions

1. Prefix sum at i = sum(A[1:i])
2. Fancy term for cumulative sum

Prefix Sums Quiz

1. Compute the prefix sums of the following array:
    - [3, 4, -2, -1, 7, -5, 6, 4]
2. Prefix sums:
    - [3, 7, 5, 4, 11, 6, 12, 16]

Scans

1. A scan generalizes prefix sum to other operations
    - +-scan ("add-scan" = "prefix sum")
    - max-scan (max to that point)
    - − -scan ("product-scan" = "prefix products")
    - and-scan (cumulative logical and)

Parallel Scan

1. Consider the serial algorithm for computing a scan

```
let A[1:n] = array of values
for i <- 2 to n do
    A[i] <- A[i-1] op A[i] // op is whatever scan we're doing
```

2. Can this algorithm be parallelized by replacing the for with a par-for?
    - No, this will result in a race condition because we are reading and writing from A simultaneously (iterations are not independent)

A Naive Parallel Scan

1. Consider the naive parallel algorithm for computing a scan

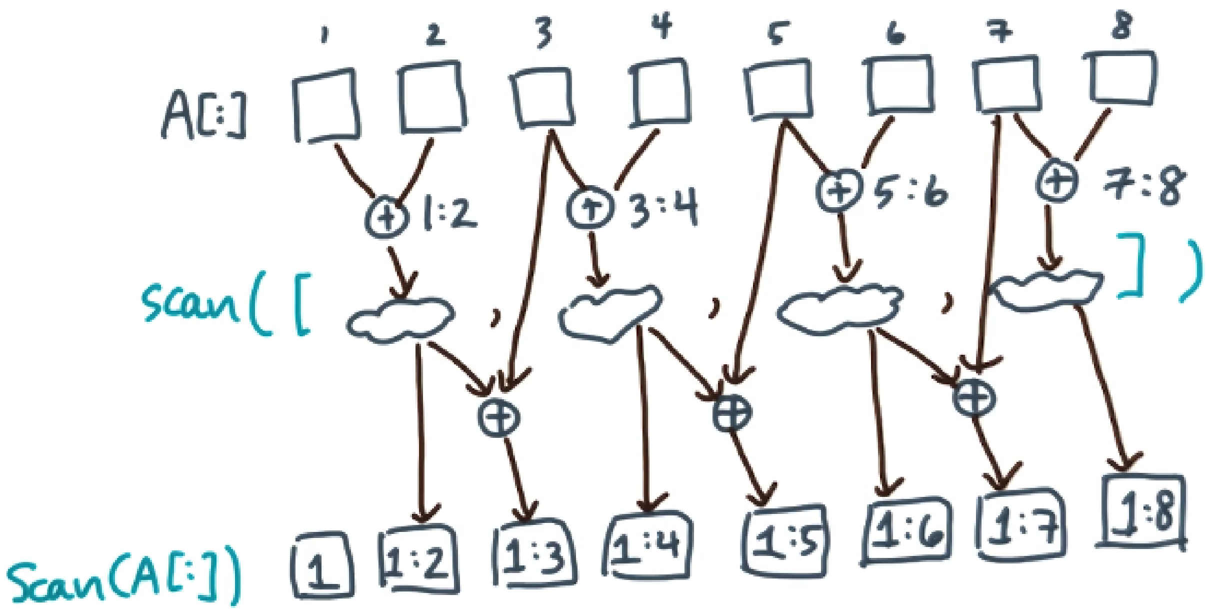
```
let A[1:n] = array of values
let B[1:n] = array of values
parfor i <- 1 to n do
    B[i] <- reduce(A[1:i])
```

2. What are the work and span of the naive parallel algorithm?
    - W(n) = O(n^2)
        - Every iteration operates on O(i) values and there are n total iterations
        - Much worse than the linear case
    - D(n) = O(log(n))

Parallel Scans

1. We can split a scan into independent blocks if the operation is associative
    - Associativity: (a + b) + c = a + (b + c)
2. The following psuedo-code provides the addScan algorithm
    - Assume n = 2 ^ k

```
addScan(A[1:n])
{
    if n == 1 then return A[1]
    let Io[1:n/2] = odd indices
    let Ie[1:n/2] = even indices
    A[Ie] <- A[Ie] + A[Io]
    A[Ie] <- addScan(A[Ie])
    A[Io] <- A[Ie[2:]] + A[Io[2:]]
}
```



Parallel Scans

Parallel Scans Quiz

1. What are the work and span of the parallel addScan algorithm?
    - W(n) = O(n)
    - D(n) = O(log(n)^2)

Parallel Scan Analysis

1. Recurrence for the total work:
    - W(n) = n - 1 + W(n/2) if n >= 2
    - W(n) = 0 if n <= 1
    - W(n) = O(n)
    - The parallel algorithm has a constant factor of ~2, while the serial algorithm has a factor of ~1
        - Do we need to pay some price for parallelism?
2. Recurrence for the total span:

- D(n) = O(log(n)) + D(n/2) if n >= 2
- D(n) = O(1) if n <= 1
- D(n) = O(log(n)^2)

Parallel Quicksort

1. Consider the following parallel quicksort algorithm:

```
QS(A[1:n])
{
    if n == 1 then return A[1]
    pivot <- any value from A (random)
    L <- {A[i] : A[i] <= pivot}
    R <- {A[i] : A[i] > pivot}
    Al <- spawn QS(L)
    Ar <- spawn QS(R)
    sync
    return Al ++ Ar // concatenation
}
```

Parallel Partitioning Quiz

1. Consider the following algorithm for gathering the elements less than or equal to a value (pivot)

```
getSmallerEqual(A[1:n], pivot)
{
    let L[1:n] = output array
    k <- 1 // L[k] = next free element
    for i <- 1 to n do {
        if A[i] <= pivot then {
            L[k] <- A[i]
            k <- k + 1
        }
    }
}
```

2. Is this algorithm safe and efficient?
    - No, need to synchronize accesses to k
        - This reduces parallelism; need a different approach

Conditional Gathers Using Scans

1. Can use a parallel scan to find all elements less than or equal to some value
    - 1 if less than or equal, 0 otherwise
2. If we do a prefix sum on this array, we get the output array indices
    - The last element is the size of the output array
    - Wherever the sum increases, we know the value is less than or equal
    - The current sum is the index in the output array
        - Can write to the output array with no conflicts
3. The below algorithm formalizes the conditional gather
    - W(n) = O(n)
    - D(n) = O(log(n))
4. The primitive of doing an index scan followed by a write is useful

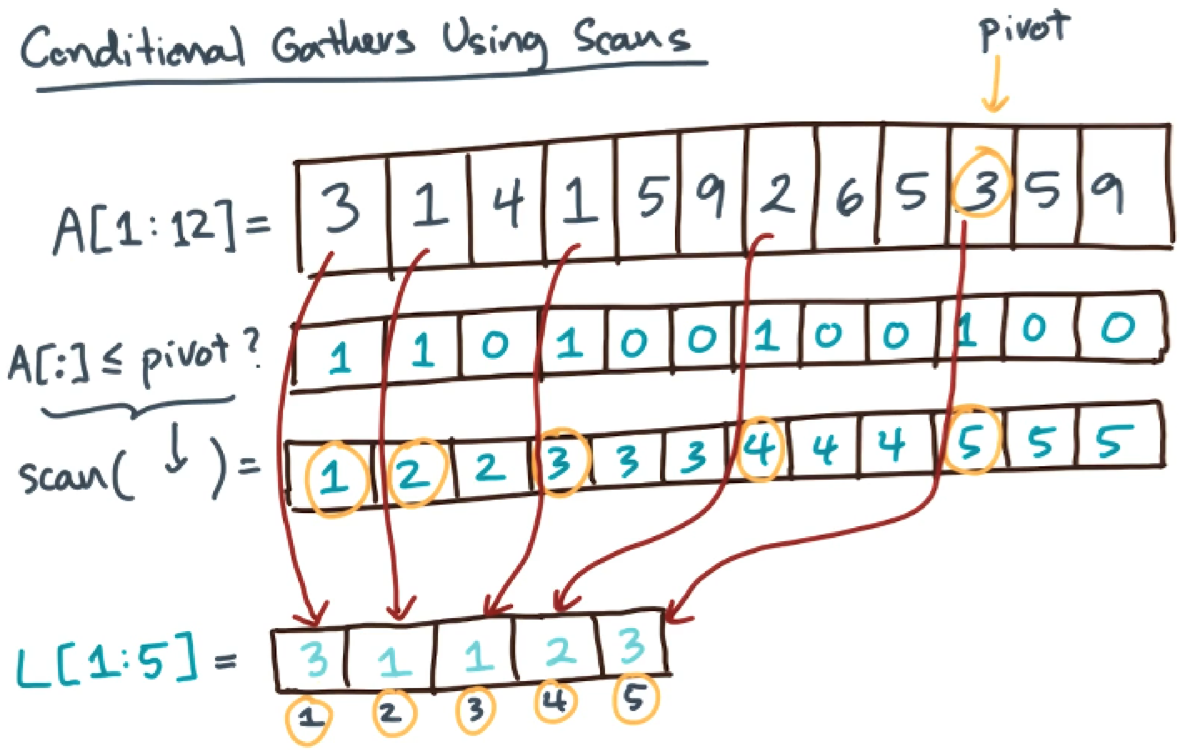
```
getSmallerEqual(A[1:n], pivot)
{
```

```
    let F[1:n] = array of {0, 1} flags
    F[:] <- (A[:] <= pivot)
    return gatherIf(A[:], F[:]) // return A[F[:]]
}

gatherIf(A[1:n], F[1:n])
{
    let K[1:n] = array of indices
    K[:] <- addScan(F[:])
    let L[1:K[n]] = output array
    par-for i <- 1 to n do
        if F[i] = 1 then L[K[i]] <- A[i]
    return L[:]
}
```

5. Updating the quicksort algorithm to use the conditional gathers:

```
QS(A[1:n])
{
    if n == 1 then return A[1]
    pivot <- any value from A (random)
    L <- A[A[:] <= pivot]
    R <- A[A[:] > pivot]
    Al <- spawn QS(L)
    Ar <- spawn QS(R)
    sync
    return Al ++ Ar // concatenation
}
```



Conditional Gathers

Segmented Scans

1. Suppose you want to perform scans over segments of the list
   - Provided an array F of true/false indices
2. The following algorithm provides a serial implementation:

```
segAddScan(A[1:n], F[1:n])
{
    for i <- 1 to n do
        if !F[i] then
            A[i] <- A[i-1] + A[i]
}
```

3. We can distill the logic inside the loop to its own operation:

```
let xi = (ai, fi) // flag, value tuple
op(xi, xj)
{
    if !fj then
        return (ai + aj, fi | fj)
    return xj
}
```

4. Then, we can write the segmented add scan as follows:

```
segAddScan(A[1:n], F[1:n])
{
    let X[0:n] = array of n+1 pairs
```

```
    X[0] <- (0, false) // identity operators for add/or
    for i <- 1 to n do
        X[i] <- (A[i], F[i])
    for i <- 1 to n do
        X[i] <- op(X[i-1], X[i])
    for i <- 1 to n do
        A[i] <- left(X[i])
```

Scan Ingredients

1. What must be true about our operator op?
    - op cost = O(1)
        - False; only affects work and span, not correctness
    - op is in-place
        - False; op is used as a function, so being in place doesn't matter
    - op is associative
        - True; op(op(a,b), c) == op(a, op(b,c))
    - op(a, b) = op(b, a)
        - False; parallel scan only combines consecutive values, so we only need associativity

List Ranking Definitions

1. The following algorithm describes the serial version of list-ranking
    - Rank(node) = distance from head
    - List ranking is a notoriously hard problem to speed up

```
rankList(head)
{
    r <- 0
    cur <- head
    while cur != NIL do
        cur.rank <- r
        cur <- cur.next
        r <- r + 1
}
```

List Ranking Quiz

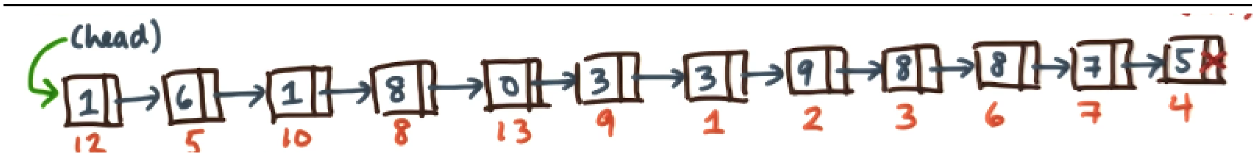
1. What value should the list contain such that an add-scan returns the list ranking?
    - 0 -> 1 -> 2 -> 3 -> 4 -> 5 -> 6 -> 7 -> 8 -> 9 -> 10 -> 11
    - Answer: 0, 1, 1, 1, 1, 1, 1, 1, 1, 1

Linked Lists as Array Pools

1. The linked list representation doesn't allow for random access
    - This is necessary for scan to work well (prevents having to traverse the entire data structure)
2. Array pool:
    - Place values in array
    - Replace "next" pointers with indices
    - Array pool representation is critical because it gives us a way to have multiple entry points into the list

Linked Lists as Array Pools Quiz

1. Represent the following linked list as an array pool



Conditional Gathers

2. V[1:14] = [3, 9, 8, 5, 6, 8, 7, 8, 3, 1, ?, 1, 0, ?] // values
3. N[1:14] = [2, 3, 6, 0, 10, 7, 4, 13, 1, 8, 0, 5, 9, 0] // next

What Does This Do?

1. Consider the following program:

```
foo(V[1:m], N[1:m])
{
    let P[1:m] = index array
    P[:] <- NIL
    par-for i <- 1 to m do
        if N[i] != NIL then
            P[N[i]] <- i
    return P[:]
}
```

2. Explain what foo() does
   - This sets each next pointer to itself, effectively reversing the list or creating a doubly-linked list

A Parallel List Ranker

1. Jump: Move the next pointer so that it moves to the neighbor's neighbor
   - Doing a jump at each node splits the list into two sublists, which is needed for divide and conquer

```
jumpList(Nin[1:m], Nout[1:m])
{
    par-for i <- 1 to m do
        if Nin[i] != NIL then
            Nout[i] <- Nin[Nin[i]]
}
```

2. To treat list ranking as an add scan, we need to set the initial values correctly
   - Put 0 at the head and 1s everywhere else
   - Maintain an invariant: rank(i) = sum(V[k=head:i])
   - Before we do a jump from a node, take its value and push it to its successor

```
updateRanks(Rin[1:m], Rout[1:m], N[1:m])
{
    par-for i <- 1 to n do
        if N[i] != NIL then
            Rout[N[i]] <- Rin[i] + Rin[N[i]]
}
```

How Many Jumps?

1. For an array pool of size n, what is the maximum number of jump steps?
   - O(1)
   - O(log(n) (yes)

- O(log(n)^2)
- O(n)
- O(nlog(n))
- O(n^2)

A Parallel List Ranker

1. Need two copies of the rank and next arrays to maintain independence (write to one array, read from the other)
2. The following algorithm describes the Wyllie list ranking algorithm

```
rankList(V[1:m], N[1:m], head)
{
    let R1[1:m], R2[1:m] = arrays of ranks
    let N1[1:m], N2[1:m] = index ("pointer") arrays
    R1[:] <- 1; R1[head] <- 0; N1[:] <- N[:]
    R2[:] <- 1; R2[head] <- 0; N2[:] <- N[:]
    for i <- 1 to ceil(log(m)) do
        updateRanks(R1[:], R2[:], N1[:])
        jumpList(N1[:], N2[:])
        swap(R1, R2);
        swap(N1, N2);
```

A Parallel List Ranker Quiz

1. What is the work and span of this parallel list ranking algorithm?
   - W(n) = O(mlog(m))
   - D(n) = O(log(m)^2)
2. This algorithm is not work optimal
   - Naive sequential algorithm only has linear cost; constants are very low
   - Need really long list and lots of processors to see any speedup

Conclusion

1. Scan/parallel prefix is a powerful primitive for exposing data parallelism
   - Vectorizes well
   - Can convert seemingly irregular serial computations into those that are both regular and parallel
2. Scans move more data than their sequential counterparts; this is hidden in the asymptotic analysis