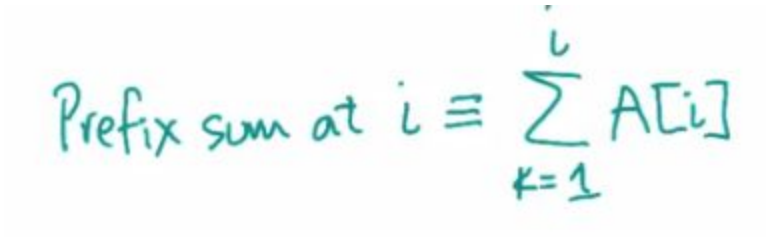
Lesson 1- 4

Prefix Sum Definitions

Prefix sum - given an array...The prefix sum is the sum of all the elements in the array from the beginning to the position, including the value at the position.



The sequential algorithm is fairly straightforward - start at the front of the array and go to the back. It would maintain a current sum and add values to it. It would not need to allocate additional space to perform the sum.

Scans

Prefix includes more than sums, for example a prefix-max computation is also possible.
The generalization of a prefix operation is called a scan.
For example: sum-scan, product-scan, max-scan, etc.

Parallel Scans

Parallel scans require iterations of the loop to be independent of one another.

A Naive Parallel Scans

To parallelize a scan - do 'n' independent parallel reductions.
The span for a parallelized scan is … O(log n) because both par-for and reduce have logarithmic span.

The work is $O(n^2)$ … each reduce costs O(i) work, so the total work is the sum of all i from 1 to 'n'.

This is worse than the sequential operation (which has O(n) operations).

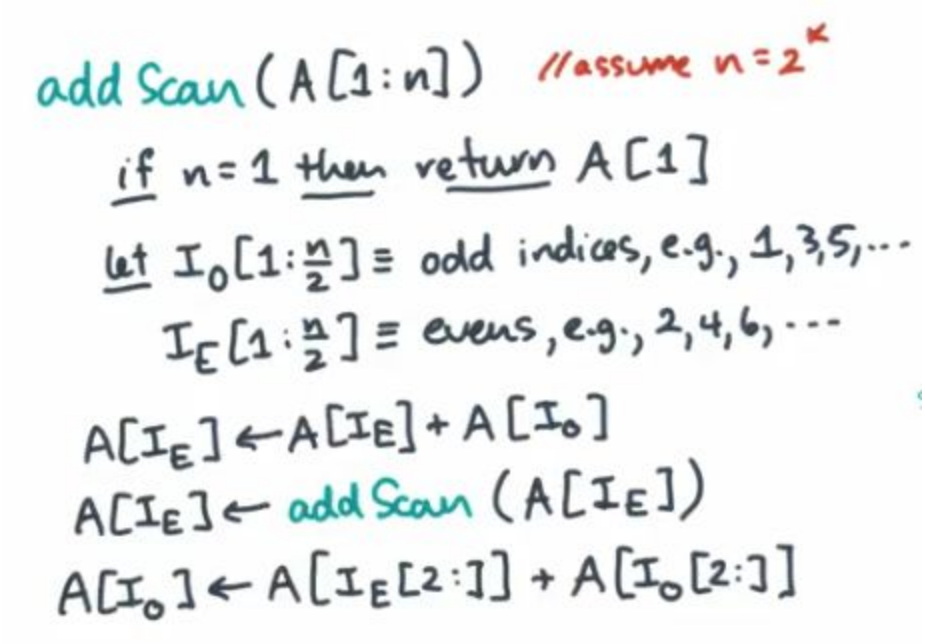
Therefore - doing independent parallel reductions is not a good idea (it's lame).

Parallel Scans

Parallelize the prefix scan problem, the same concept can be used for parallelizing other scans.

Prefix Sum Parallelization steps:

1. If prefix sum can be reduced to single value and associativity can be assumed, the the partial sums can be rearranged.
2. Combine an element with its neighbor. This will be a new set of partial consecutive sums. The first element of the partial sums is the first element of the array (not the sum).
3. Do a scan on the partial consecutive sums. This will give you all of the even results.
4. To get the odd results - take the partial sum and add the odd value to it. For example: to get 1:3 → Add 1:2 plus 3



Work for this algorithym → O(n)
Span for this algorithm → $O(\log^2 n)$

Parallel Scan Analysis

Work
Note that the number of additions the algorithm does at different stages. Addscan operates on a problem that is ½ the size.

W(n) ~ 2n → this is hidden in the work and only counts addition operations.
The sequential version of the algorithm W(n) = n. The parallel version does twice the amount of work. Theoretically this is not important, but it means there is a cost to be paid for parallelism.

Span

addScan operates on a sub-problem of half the size. The other operations can be implemented using par-for. addScan has O(log n) span and the other operations are O(log n). When combined the total span is $O(\log^2 n)$.

The master theorem can be used to solve this quickly and easily.

Master Theorem

Parallel QuickSort

Sequential Quicksort:

1. Choose a pivot element - select one uniformly at random.
2. Partition the input around the pivot value. Elements that are less than or equal to go on one side, greater than go on the other side.
3. Now the two partitions can be independently sorted.

Parallel Quicksort:

1. Spawn the two partitions as independent tasks.
2. Repeat: Choose a pivot, partition, and spawn.

The Parallel Algorithm for Quicksort

```
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}
  A_L ← spawn QS(L)
  A_R ← QS(R)
  sync
  return A_L ++ A_R
```

Parallel Partition:

You cannot just substitute a par-for for the ‘for’ loop. You need to have a lock on K, without this the par-for loop will give incorrect values.

Conditional Gathers Using Scan

To do the quicksort in parallel →

1. Select a pivot.
2. In parallel → compare each element to the pivot. If the element is less than or equal to the pivot put a ‘1’ in the auxiliary array. If the element is greater than put a ‘0’ in the auxiliary array.

This will result in a array consisting of 1s and 0s. This array can be scanned.
The following information can be gotten from the scan:

1. The last element in the array is the total number of elements less than or equal to the pivot. This means we can allocate an output array of the appropriate size.
2. Every time there is an increase in the scan, this corresponds to an element in the array that is less than or equal to the pivot. This makes the desired elements easy to find in a scan.
3. Within the scan - the incremental values are both unique and consecutive. These values can be used as indices to the output array and can be written in parallel.

This is the pseudo code for the conditional gathers using scans.

W(n) = O(n)
D(n) = O(log n)

A flag scan followed by a write can be combined into an algorithmic primitive called gatherIf.

gatherIf(A[:], F[:]) → A[F[:]]

Segmented Scans

To do independent scans on segments of the array - the segments do not have to be the same size.

1. Begin with an array of flags, consisting of True and False. A true value is placed wherever a segment begins.
2. Then an algorithm looks at the flag array and if it sees a true flag it does nothing. If it sees a false flag it does a scan.

Use the ‘op’ primitive to replace the conditional statement. Then we can use it to parallelize a segmented scan.

The ‘op’ primitive needs to be associative.
op(op(a,b),c) = op(a,(b,c))

The cost of an operation affects its work and span, but not its correctness.

List Ranking Definitions

List ranking is hard to speed up.

List ranking asks:
Given a linked list - what is the distance of each node from the head?

Sequentially the problem is easy.

If given a linked list, add a one for each node except the head node. Then you can scan the list.

Linked Lists as Array Pools

A linked list has only one entry point, to make it more accessible, use two arrays. One array will hold the value of the node, the second array will hold the index (the next pointer).

The array representation must be at least as large as the linked list.

A Parallel List Ranker

To make this parallel, use the jump primitive.
The jump primitive → takes a nodes pointer (that was pointing to the neighbor) and points it to the neighbors neighbor.

If the jumps occur at the same time, the list is split into two sublists. Each sublist has every other node in the list.

The simultaneous jumps can be done repeatedly, creating many sublists of smaller and smaller size.

The parallel list ranker:

1. store the list as an array pool
2. List ranking is basically an addscan
3. Use the jump primitive to divide and conquer.

Candidate invariant: if 1s are used to denote rank, the jumps will disturb this method. So the ranking should be changed to be an addscan - add the ranks and store this in the array.

Update Ranks is a primitive that pushes (adds the two ranks) the rank value onto the receiving node. This preserves the rank of each node. These updates can be done in parallel.

If the array pool is of size ‘n’, the maximum number of jump steps is O(log n)

A Parallel List Ranker (formal)

index of head

```
rankList (V[1:m], N[1:m], h)
  let R1[1:m], R2[1:m] ≡ arrays of ranks
  let N1[1:m], N2[1:m] ≡ index ("pointer") arrays
  R1[:] ← 1 ; R1[h] ← 0 ; N1[:] ← N[:]
  R2[:] ← 1 ; R2[h] ← 0 ; N2[:] ← N[:]
  for i ← 1 to ⌈log m⌉ do
     updateRanks (R1[:], R2[:], N1[:])
     jumpList (N1[:], N2[:])
     swap (R1, R2) ; swap (N1, N2)
```

$W(n) = O(m \log m)$

$D(n) = O(\log^2 m)$