#### Lecture 35: Parallel Prefix Sum

Mack Joyner mjoyner@rice.edu

http://comp322.rice.edu





#### **COMP 322: Parallel and Concurrent Programming**

April 2023

Lecture 35



## **Beyond Sum/Reduce Operations** — Prefix Sum (Scan) Problem Statement

Given input array A, compute output array X as follows

- $X[i] = \sum A[j]$  $0 \le j \le i$
- The above is an <u>inclusive</u> prefix sum since X[i] includes A[i]
- For an <u>exclusive</u> prefix sum, perform the summation for  $0 \le i \le i$
- It is easy to see that inclusive prefix sums can be computed sequentially in O(n) time ...

// Copy input array A into output array X

X = new int[A.length]; System.arraycopy(A,0,X,0,A.length);

// Update array X with prefix sums

for (int i=1; i < X.length; i++) X[i] += X[i-1];

... and so can exclusive prefix sums



# An Inefficient Parallel Algorithm for Exclusive Prefix Sums

- **1.** forall(0, X.length-1, (i) -> {
- 2. // computeSum() adds A[0..i-1]
- 3. X[i] = computeSum(A, 0, i-1);4.}

#### **Observations:**

- Critical path length,  $CPL = O(\log n)$
- Total number of operations, WORK =  $O(n^2)$
- max(CPL, WORK/P) = O(n), which is no better than sequential!

With P = O(n) processors, the best execution time that you can achieve is  $T_P =$ 



Assume that input array A = [3, 1, 2, 0, 4, 1, 1, 3]Define scan(A) = exclusive prefix sums of A = [0, 3, 4, 6, 6, 10, 11, 12]

Hint:

- Compute B by adding pairwise elements in A to get B = [4, 2, 5, 4]Assume that we can recursively compute scan(B) = [0, 4, 6, 11]How can we use A and scan(B) to get scan(A)?

#### How can we do better?



- Set each node's root to its parent
- For each node, set its root to its parent's root, if it exists
- This can all be done in parallel using N tasks



### Remember the "Pointer Skipping" Idea?





- For each node's root starts as its parent
- For each node, set its root to its parent's root, if it exists
- This can all be done in parallel using N tasks



### Remember the "Pointer Skipping" Idea?





- Again:
- For each node, set its root to its parent's root, if it exists
- This can all be done in parallel using N tasks again
- Stop when no more updates can be done



### Remember the "Pointer Skipping" Idea?





Observation: each prefix sum can be decomposed into reusable terms of power-of-2-size e.g.

$$X[6] = A[0] + A[1] +$$

Approach:

- Prefix Sum
- tree nodes
- upward sweep

# Another way of looking at the parallel algorithm

A[2] + A[3] + A[4] + A[5] + A[6]A[2] + A[3]) + (A[4] + A[5]) + A[6]

Combine reduction tree idea from Parallel Array Sum with partial sum idea from Sequential

Use an "upward sweep" to perform parallel reduction, while storing partial sum terms in

Use a "downward sweep" to compute prefix sums while reusing partial sum terms stored in





COMP 322, Spring 2023 (M. Joyner)

#### Parallel Prefix Sum: Upward Sweep (while calling scan recursively)

- 1. Upward sweep is just like Parallel Reduction, except that partial sums are also stored along the way
- 2. Receive values from left and right children
- 3. Compute left+right and store in box
- 4. Send left+right value to parent





#### Parallel Prefix Sum: Downward Sweep (while returning from recursive calls to scan)

- 1. Receive value from parent (root receives 0)
- 2. Send parent's value to LEFT child (prefix sum for elements to left of left child's subtree)
- 3. Send parent's value+ left child's box value to RIGHT child (prefix sum for elements to left of right child's subtree)
- 4. Add A[i] to get inclusive prefix sum





- Critical path length,  $CPL = O(\log n)$
- Total number of add operations, WORK = O(n)
- Optimal algorithm for  $P = O(n/\log n)$  processors -Adding more processors does not help
- Parallel Prefix Sum has several applications that go way beyond computing  $\bullet$ the sum of array elements
  - Parallel Prefix Sum can be used for any operation that is associative (need) not be commutative)
    - In contrast, finish accumulators required the operator to be both associative and commutative

# Summary of Parallel Prefix Sum Algorithm





that **f(elt)** is true, i.e., output = input.parallelStream().filter(f).toArray()

# Example: input [17, 4, 6, 8, 11, 5, 13, 19, 0, 24] f: is elt > 10output [17, 11, 13, 19, 24]

Parallelizable?

- Finding elements for the output is easy
- But getting them in the right place seems hard

#### Parallel Filter Operation [Credits: David Walker and Andrew W. Appel (Princeton), Dan Grossman (U. Washington)]

# Given an array **input**, produce an array **output** containing only elements such



- 1. Parallel map to compute a bit-vector for true elements (can use Java streams) input [17, 4, 6, 8, 11, 5, 13, 19, 0, 24] bits [1, 0, 0, 0, 1, 0, 1, 1, 0, 1]
- 2. Parallel-prefix sum on the bit-vector (not available in Java streams) bitsum [1, 1, 1, 1, 2, 2, 3, 4, 4, 5]
- 3. Parallel map to produce the output (can use Java streams) output [17, 11, 13, 19, 24]

- ray of size bitsum[n-1] < input.length; i++){
- n[i]-1] = input[i];



# **Example Applications of Parallel Prefix Algorithm**

Prefix Max with Index of First Occurrence: given an input array A, output an array X of objects such that X[i].max is the maximum of elements A[0...i] and X[i].index contains the index of the first occurrence of X[i].max in A[0...i]

Filter and Packing of Strings: given an input array A identify elements that satisfy some desired property (e.g., uppercase), and pack them in a new output array. (First create a 0/1 array for elements that satisfy the property, and then compute prefix sums to identify locations of elements to be packed.)

