# COMP 322: Parallel and Concurrent Programming 

# Lecture 35: Parallel Prefix Sum 

Mack Joyner<br>mjoyner@rice.edu<br>http://comp322.rice.edu

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

Given input array A , compute output array X as follows

$$
X[i]=\sum_{0 \leq j \leq i} A[j]
$$

- The above is an inclusive prefix sum since $X[i]$ includes $A[i]$
- For an exclusive prefix sum, perform the summation for $0<=j<i$
- 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 $\mathrm{i}=1$; $\mathrm{i}<\mathrm{X}$.length ; $\mathrm{i}++$ ) $\mathrm{X}[\mathrm{i}]+=\mathrm{X}[\mathrm{i}-1]$;
- ... 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. $\mathrm{X}[\mathrm{i}]=$ computeSum(A, $0, \mathrm{i}-1)$;
4. $\}$

Observations:

- Critical path length, $\mathrm{CPL}=\mathrm{O}(\log \mathrm{n})$
- Total number of operations, WORK $=\mathrm{O}\left(\mathrm{n}^{2}\right)$
- With $\mathrm{P}=\mathrm{O}(\mathrm{n})$ processors, the best execution time that you can achieve is $T_{\mathrm{P}}=$ $\max (\mathrm{CPL}, \mathrm{WORK} / \mathrm{P})=\mathrm{O}(\mathrm{n})$, which is no better than sequential!


## How can we do better?

Assume that input array $\mathrm{A}=[3,1,2,0,4,1,1,3]$
Define $\operatorname{scan}(A)=$ 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 $\operatorname{scan}(B)=[0,4,6,11]$
- How can we use A and $\mathrm{scan}(\mathrm{B})$ to get $\operatorname{scan}(\mathrm{A})$ ?


## Remember the "Pointer Skipping" Idea?

- 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



## Another way of looking at the parallel algorithm

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

$$
\begin{aligned}
X[6] & =A[0]+A[1]+A[2]+A[3]+A[4]+A[5]+A[6] \\
& =(A[0]+A[1]+A[2]+A[3])+(A[4]+A[5])+A[6]
\end{aligned}
$$

Approach:

- Combine reduction tree idea from Parallel Array Sum with partial sum idea from Sequential Prefix Sum
- Use an "upward sweep" to perform parallel reduction, while storing partial sum terms in tree nodes
- Use a "downward sweep" to compute prefix sums while reusing partial sum terms stored in upward sweep


## 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

Exclusive prefix sums
$+\mathbf{A}[\mathrm{i}]$
Inclusive prefix sums

## Summary of Parallel Prefix Sum Algorithm

- Critical path length, $\mathrm{CPL}=\mathrm{O}(\log n)$
- Total number of add operations, $W O R K=O(n)$
- Optimal algorithm for $\mathrm{P}=\mathrm{O}(\mathrm{n} / \log \mathrm{n})$ processors
-Adding more processors does not help
- Parallel Prefix Sum has several applications that go way beyond computing 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


## Parallel Filter Operation

Given an array input, produce an array output containing only elements such that $\mathbf{f}(\mathrm{elt})$ 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 $>10$
output [17, 11, 13, 19, 24]

Parallelizable?

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


## Parallel prefix to the rescue

1. Parallel map to compute a bit-vector for true elements (can use Java streams)

$$
\begin{aligned}
& \text { input }[17,4,6,8,11,5,13,19,0,24] \\
& \text { bits }[1,0,0,0,1,0,1,1,0,1]
\end{aligned}
$$

2. Parallel-prefix sum on the bit-vector (not available in Java streams) bitsum [0, 1, 1, 1, 1, 2, 2, 3, 4, 4]
3. Parallel map to produce the output (can use Java streams) output [17, 11, 13, 19, 24]
```
output = new array of size bitsum[n-1] +1
FORALL(i=0; i < input.length; i++){
    if(bits[i]==1)
    output[bitsum[i]] = 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 $A[0 \ldots i]$ and $X[i]$.index contains the index of the first occurrence of $X[i]$.max in $A[0 \ldots i]$
- 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.)

