CSE 6220/CX 4220
Introduction to HPC

Lecture 7: Prefix Sums and Their Applications

Helen Xu
hxu615@gatech.edu

Georgia Tech College of Computing
School of Computational Science and Engineering

Reminder: Exam 1 in class on Sept 23

- Efficiency / speedup / scalability analysis
- Analysis of multithreaded algorithms (work / span)
- Cache complexity analysis / cache-oblivious analysis
- Prefix sums

Prefix Sum (Scan) Definition

Input: an array $x = [x_0, x_1, \ldots, x_{n-1}]$ of n elements

Output: an array $y = [y_0, y_1, \ldots, y_{n-1}]$ of running sums, where

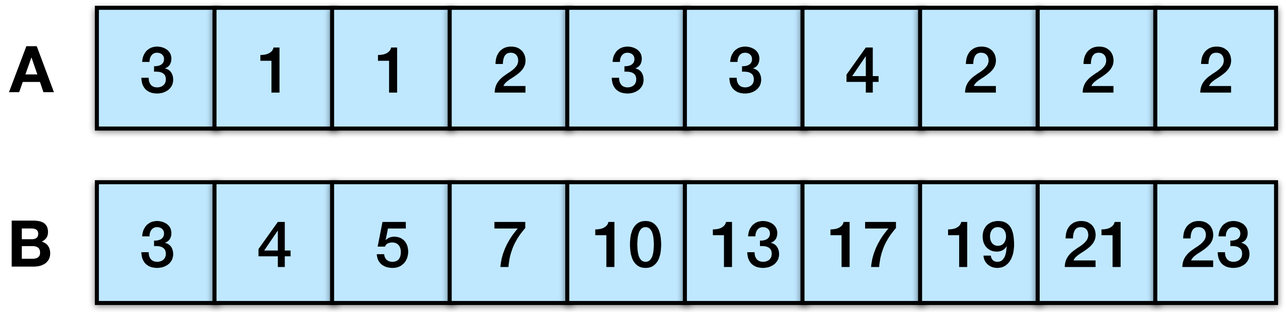
$$y_k = \begin{cases} x_0 & \text{if } k = 0 \\ x_k \oplus y_{k-1} & \text{if } k \geq 1. \end{cases}$$

Any binary associative operator (e.g., addition, multiplication, etc.)

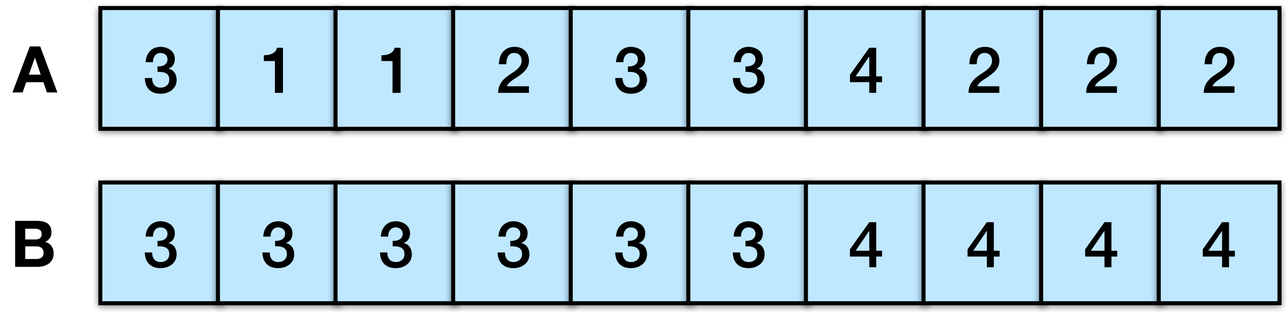
Scan Examples

Fill array with **partial reductions** from any binary associative operation

A = array
B = array
B = scan(A, +)



A = array
B = array
B = scan(A, max)



Inclusive and Exclusive Scans

Inclusive scan: includes x_k when computing y_k - as in our previous examples.

Another variant: **exclusive scan** does not include x_k when computing y_k.

A = array
B = array
B = inclusive_scan(A, +)

C = array
C = exclusive_scan(A, +)

A	3	1	1	2	3	3	4	2	2	2
B	3	4	5	7	10	13	17	19	21	23
C	0	3	4	5	7	10	13	17	19	21

Can easily get the inclusive version from the exclusive by adding the input element-wise.

For the other way, you need an inverse for the operator.

Suffix Scans

If prefix is “left to right”, suffix is just the reverse “right to left”

Input: an array $x = [x_0, x_1, \ldots, x_{n-1}]$ of n elements

Output: an array $y = [y_0, y_1, \ldots, y_{n-1}]$ of running sums, where

$$y_k = \begin{cases} x_{n-1} & \text{if } k = n-1 \\ x_k \oplus y_{k+1} & \text{if } k < n-1. \end{cases}$$

Any binary associative operator (e.g., addition, multiplication, etc.)

Shared-Memory Parallel Prefix

Can we parallelize a scan?

Serial scan takes n-1 operations.

The i-th iteration iteration of the loop **depends completely** on the (i-1)-th iteration.

```
y[0] = 0;
for (size_t i = 1; i < n; i++) {
  y[i] = y[i-1] + x[i];
}
```

From UC Berkeley CS267

First try: Parallel But Inefficient

Apply tree-like reduction at every element: put 1 processor at element 1, 2 at element 2, etc.

Span: $\lg(n)$

Work: $O(n^2)$

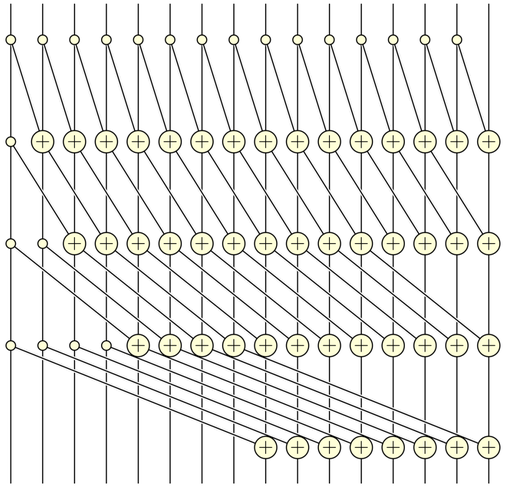
Work-efficient parallel algorithms perform **no more than a constant factor of work** over the best serial algorithm for the problem

A	1	2	3	4	5	6	7	8
B	1	3	6	10	15	21	28	36

Hillis-Steele Prefix Sum

for i = 0 up to $\log(n)$:
  for j = 0 up to n-1:
    if j < 2^i:
      $x_j^{i+1} \leftarrow x_j^i$
    else:
      $x_j^{i+1} \leftarrow x_j^i + x_{j-2^i}^i$

What is the work and span?



https://en.wikipedia.org/wiki/Prefix_sum

"Data-Parallel Algorithms," Hillis and Steele, CACM 1986.

Hillis-Steele Prefix Sum

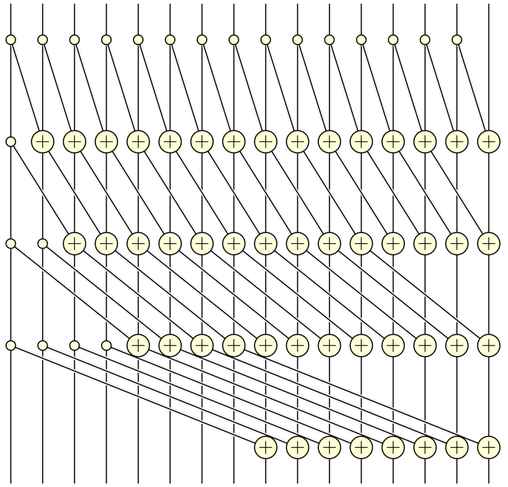
for i = 0 up to $\log(n)$:
  for j = 0 up to n-1:
    if j < 2^i:
      $x_j^{i+1} \leftarrow x_j^i$
    else:
      $x_j^{i+1} \leftarrow x_j^i + x_{j-2^i}^i$

What is the work and span?

Work = $O(n \lg n)$

Span = $O(\lg n)$

Better, but still not work efficient



https://en.wikipedia.org/wiki/Prefix_sum

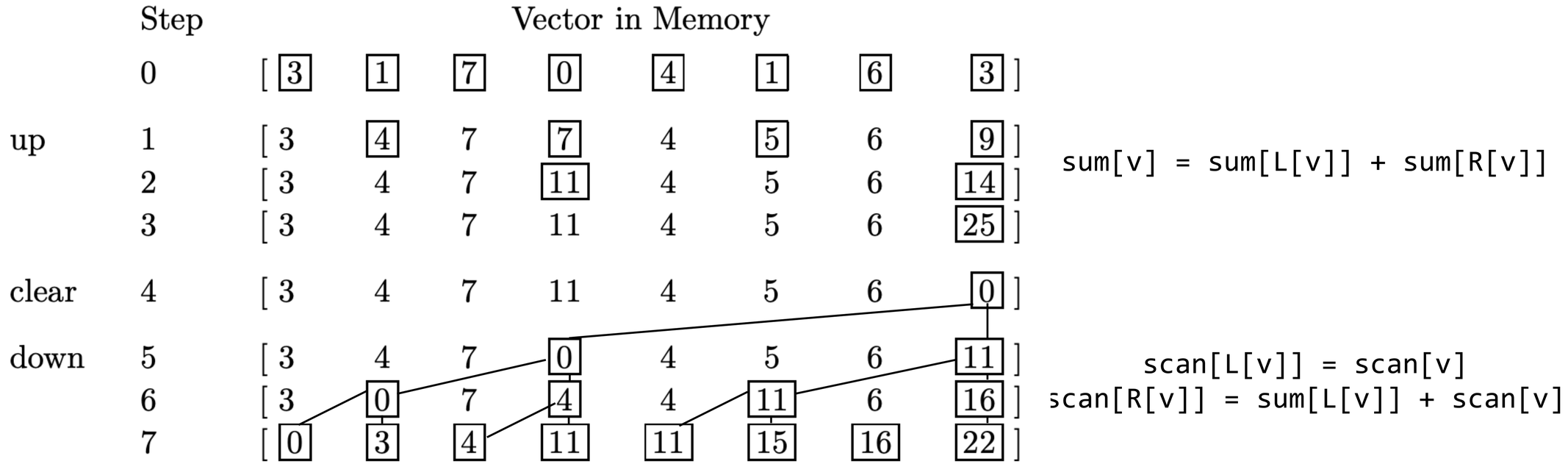
"Data-Parallel Algorithms," Hillis and Steele, CACM 1986.

Work-Efficient Parallel Prefix

Idea: Save the partial sums computed via parallel reduction (**upsweep**) and use those values in a **downsweep** pass to compute the total prefix.

The downsweep works by performing sums **down the prefix-sum tree**: at each step, each vertex at a given level passes its own value to its left child, and its right child gets the sum of the left child and the parent.

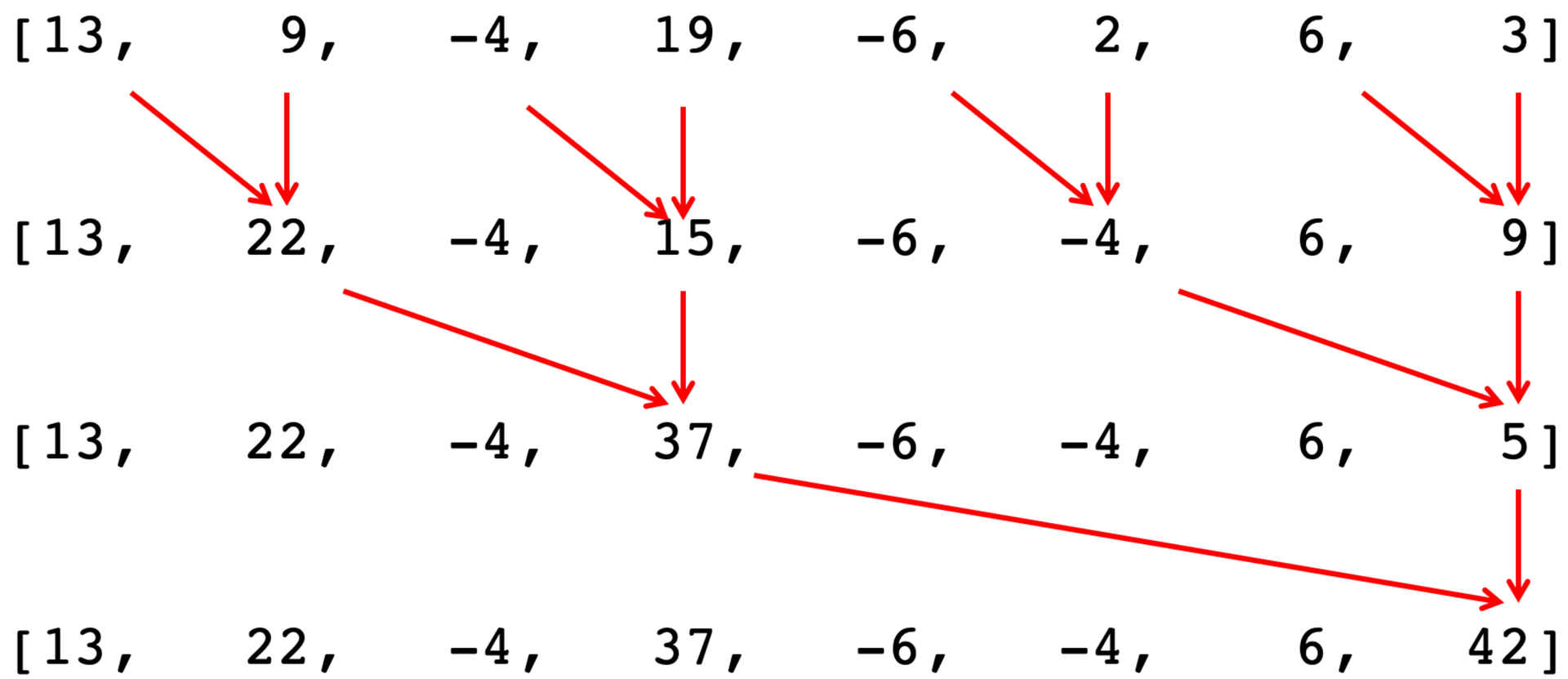
Exclusive scan:



https://en.wikipedia.org/wiki/Prefix_sum

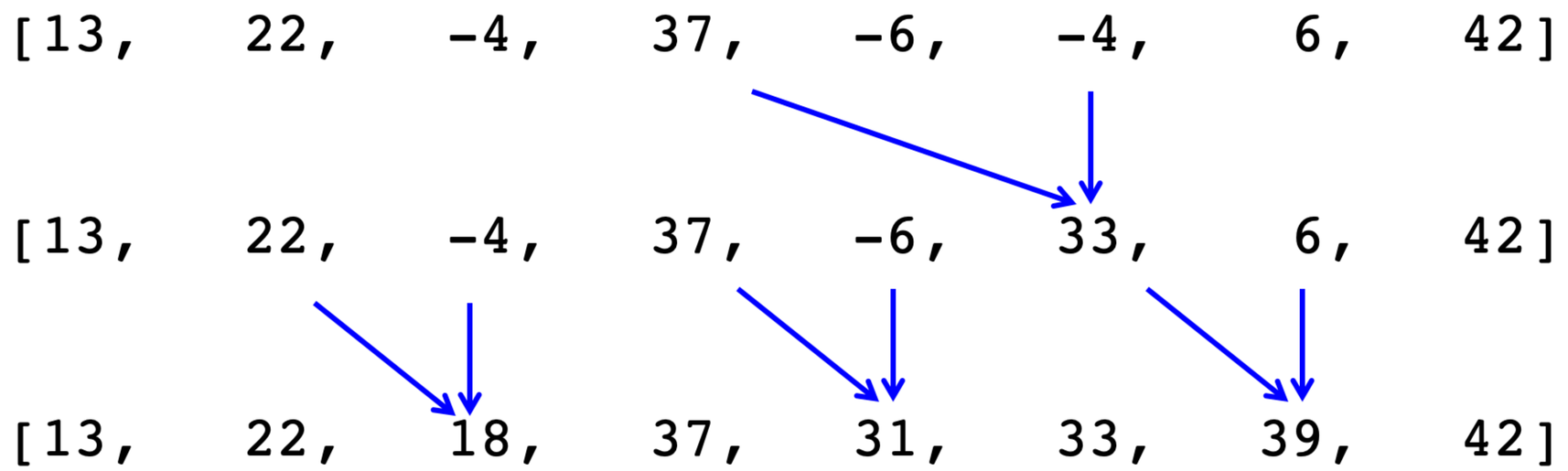
"Prefix Sums and Their Applications," Blelloch, 1990.

Upsweep Example



Downsweep Example

For an inclusive scan, only the downsweep needs to change:



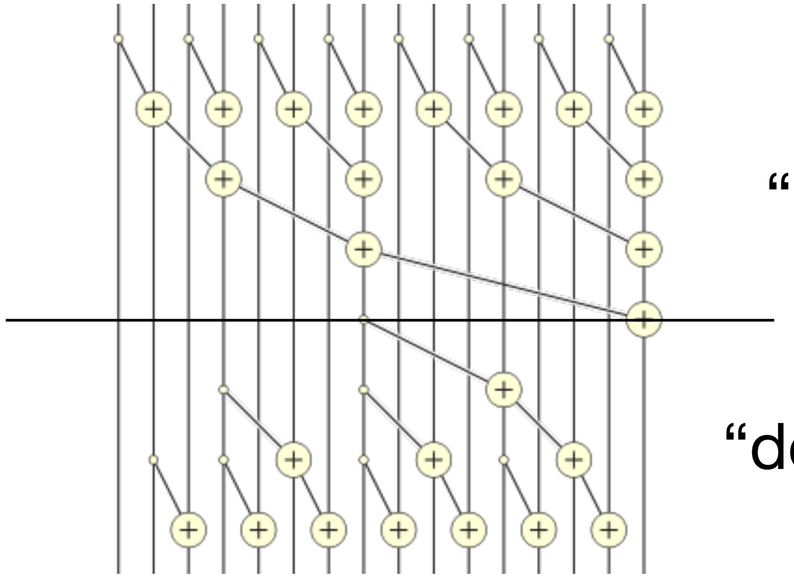
- Recall, we started with:

[13, 9, -4, 19, -6, 2, 6, 3]

Work-Efficient Parallel Prefix Pseudocode

Upsweep:
for d from 0 to $\lg(n) - 1$:
  parallel_for i from 0 to $n - 1$, i += 2^{d+1}:
    $A[i + 2^{d+1} - 1] \leftarrow A[i + 2^d - 1] + A[i + 2^{d+1} - 1]$

Downsweep:
for d from $\lg(n) - 1$ to 0:
  parallel_for i from $2^d - 1$ to $n - 1 - 2^d$, i += 2^{d+1}:
    if $i - 2^d \geq 0$:
      $A[i] = A[i] + A[i - 2^d]$



“upsweep”

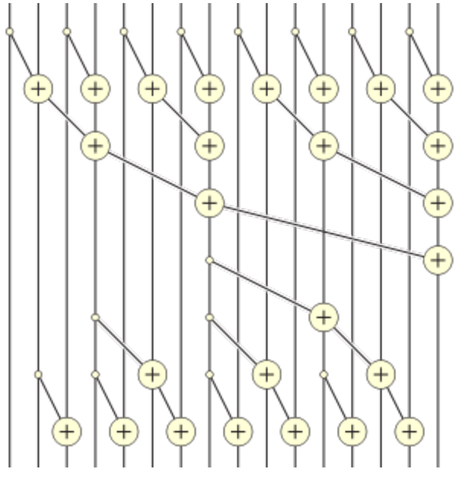
“downsweep”

https://en.wikipedia.org/wiki/Prefix_sum

“Prefix Sums and Their Applications,” Blelloch, 1990.

Analysis of Parallel Prefix

What is the work and span?



Analysis of Parallel Prefix

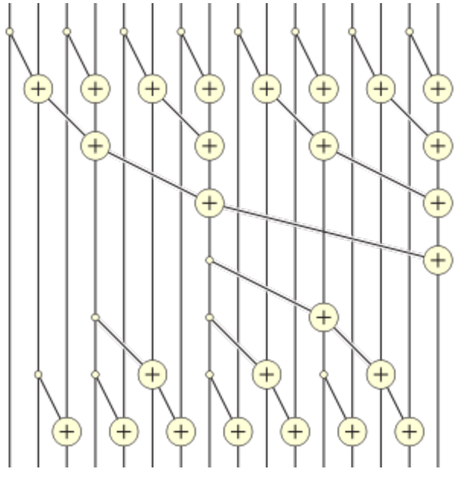
What is the work and span?

Work efficient

Work: $W(n/2) + O(n) = O(n)$

Span: $S(n) = S(n/2) + O(1) = O(\lg n)$

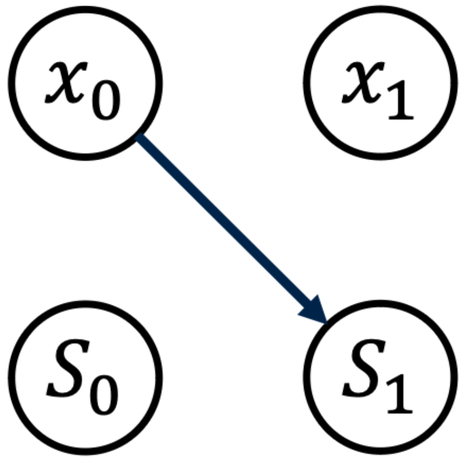
Gave up a constant factor in span compared to Hillis-Steele algorithm



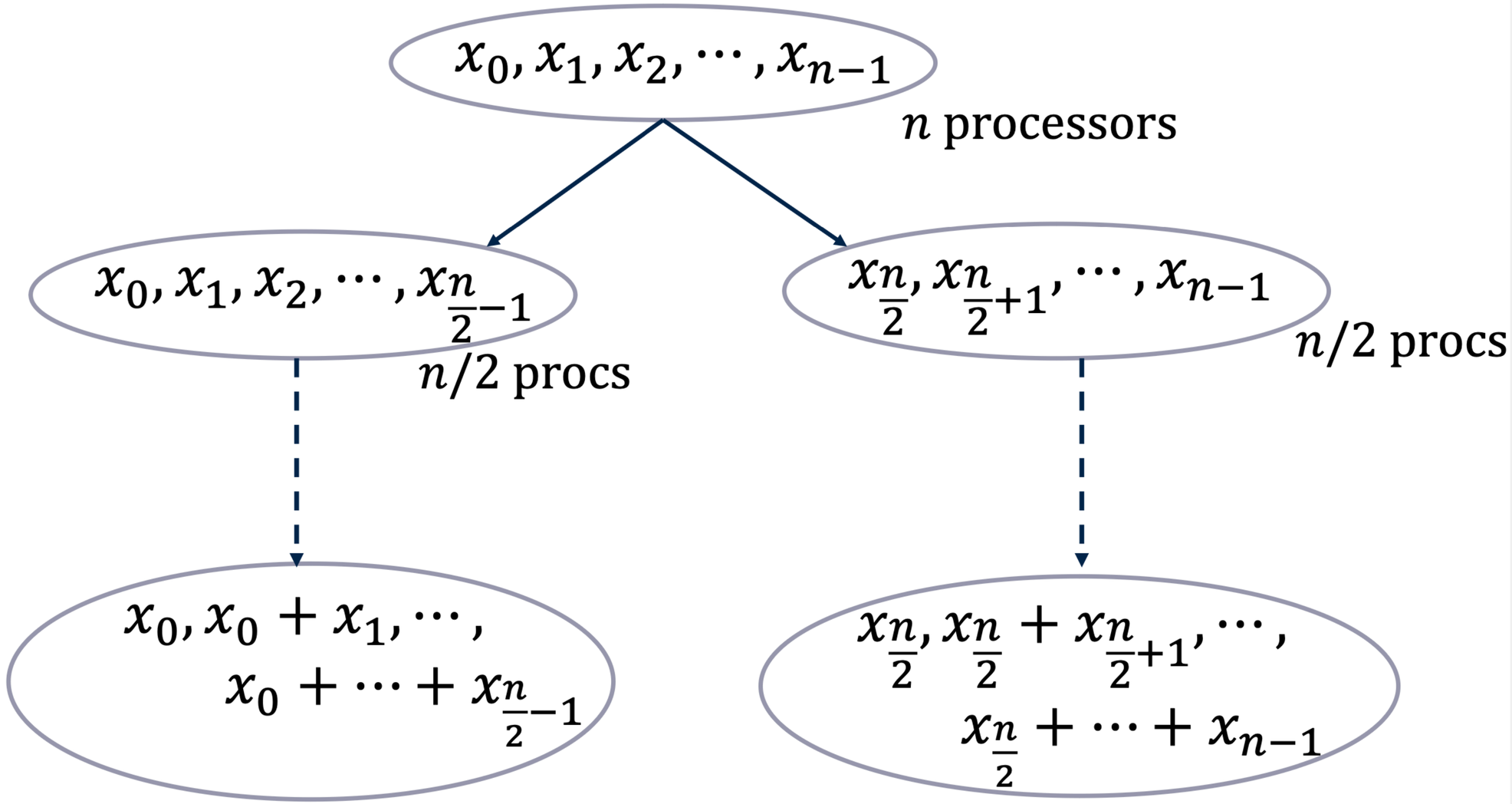
Distributed-Memory Parallel Prefix

Divide-and-conquer parallel prefix

Base case of two elements:



Divide-and-conquer parallel prefix

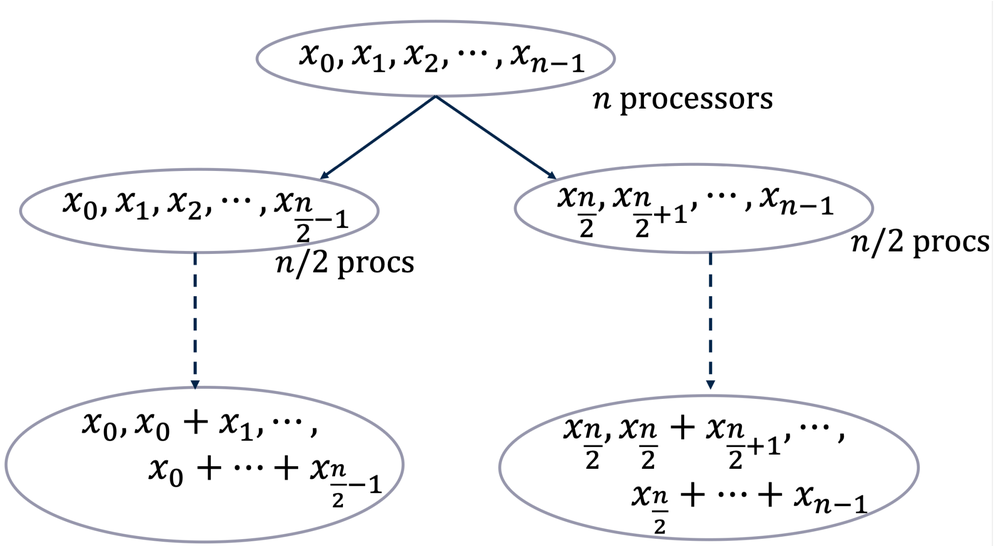


From Srinivas Aluru

Analysis of Initial Divide-and-Conquer Attempt

- To merge the two recursive calls into the final correct answer, $S_{n/2-1}$ needs to be communicated to all processors on the right.
- $T(n,n) = T(n/2,n/2) + \Theta(\lg n) = \Theta(\lg^2 n)$

To broadcast rightmost sum in the left half to all elements in the right half



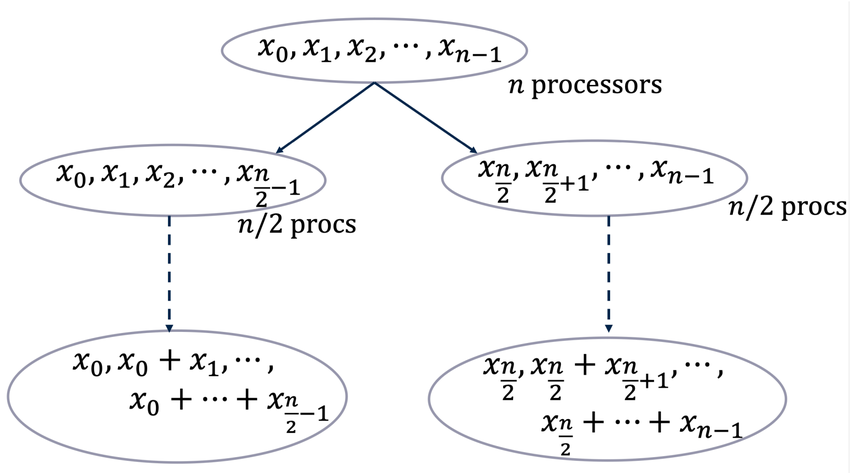
Can we do better? i.e., $T(n,n) = \Theta(\lg n)$?

A Cute Trick: Computing More for Better Efficiency

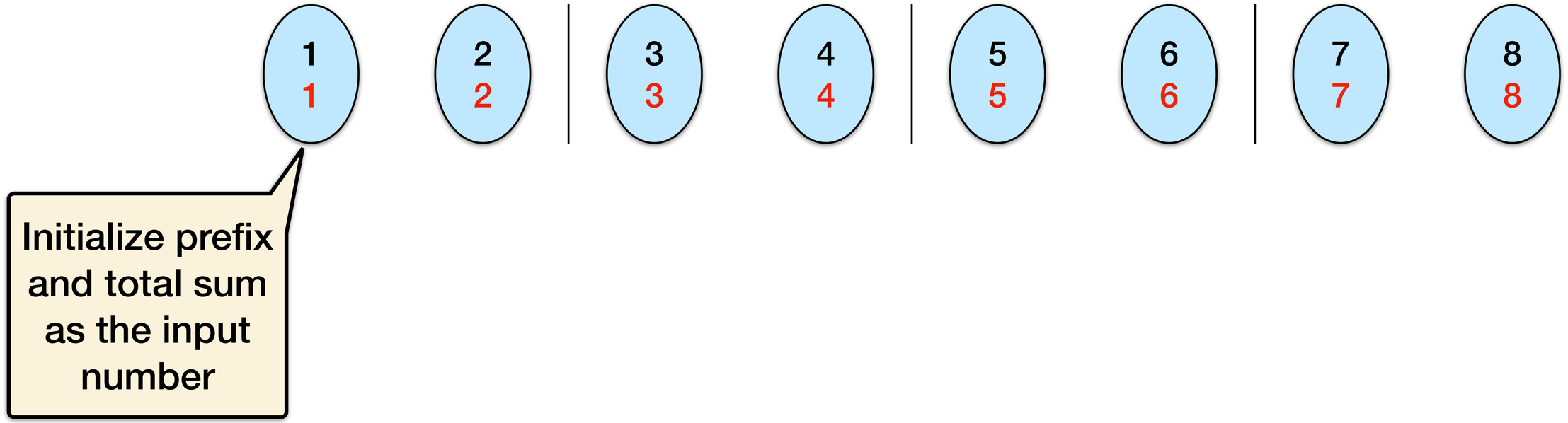
Problem: only one processor on the left side knows the total sum of that half. What if **every processor on the left half knew the total sum**?

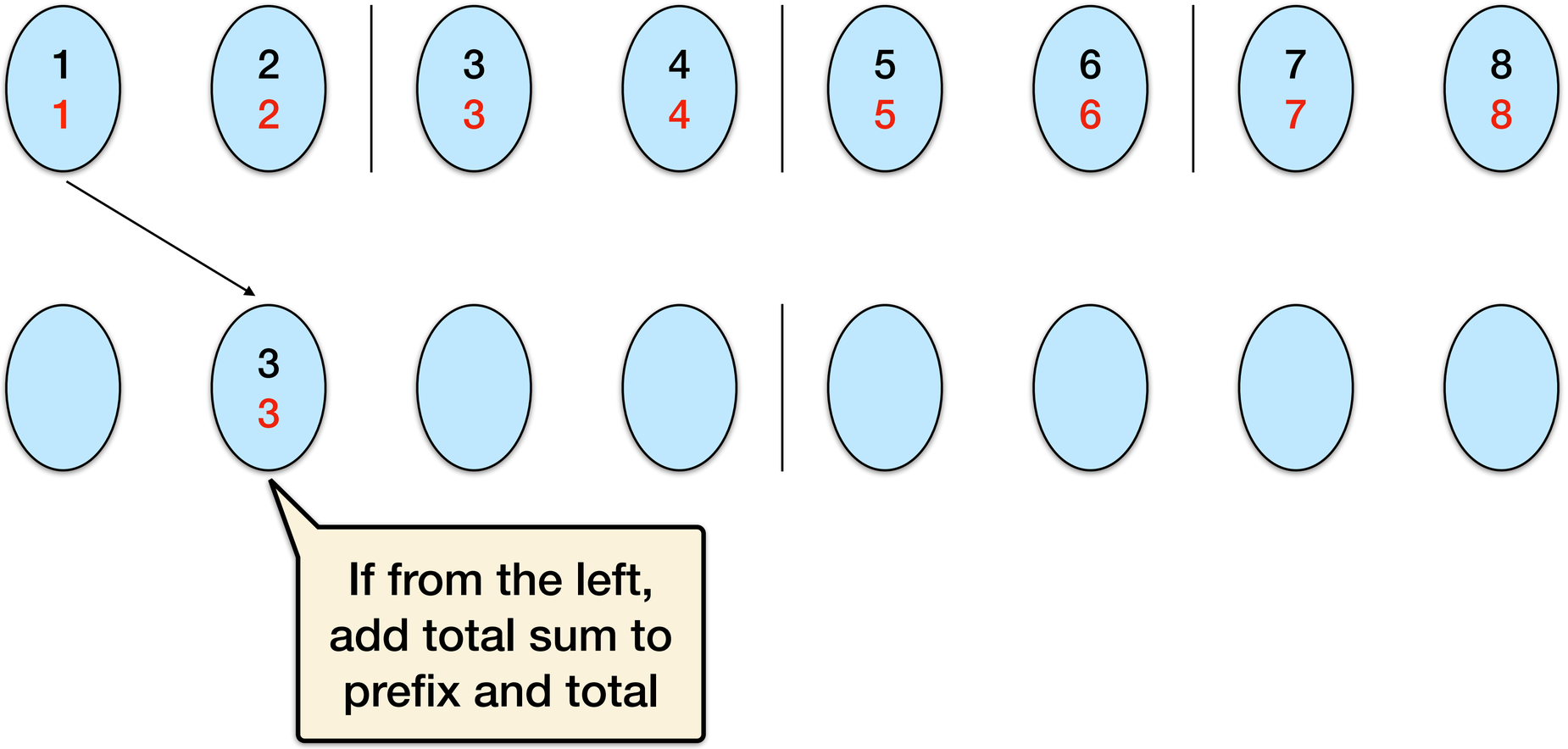
- Can then pair left and right according to a hypercubic permutation (flip the top bit)

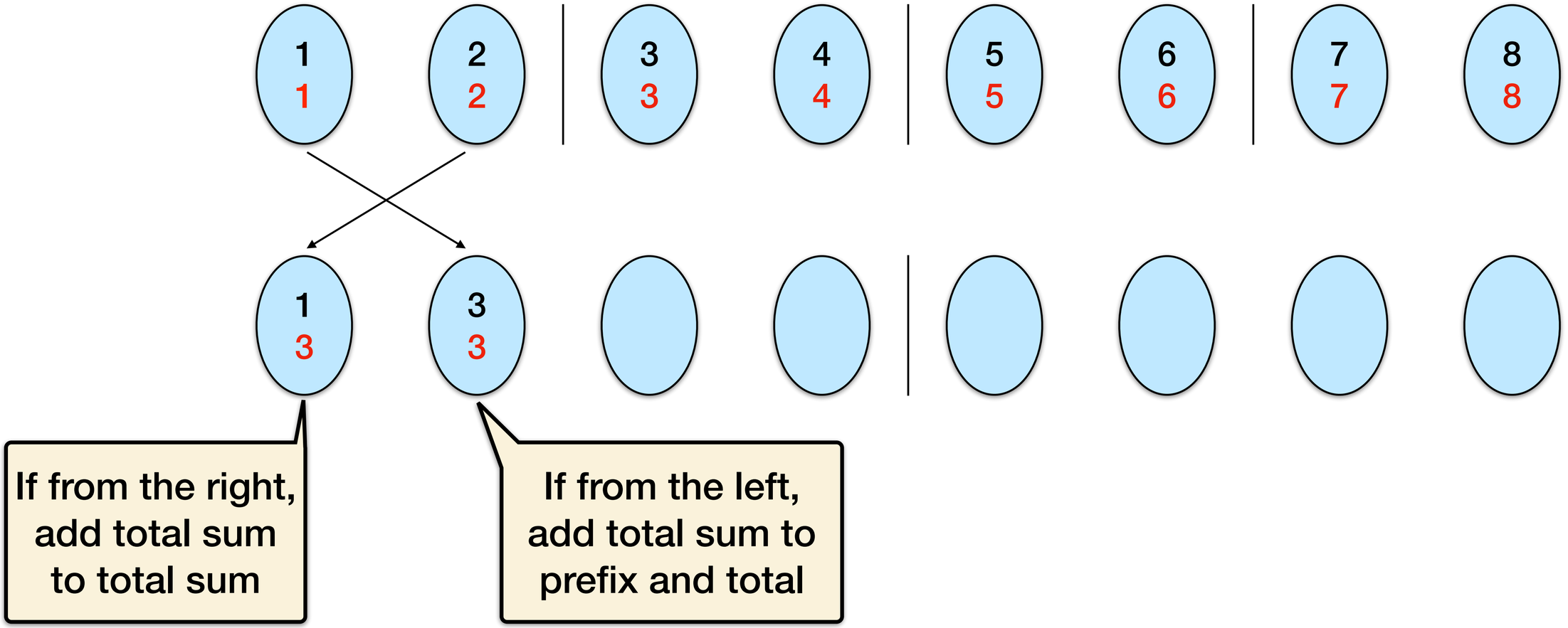
Insight: Compute **the total sum** on every processor on the left side in addition to the prefix sum.

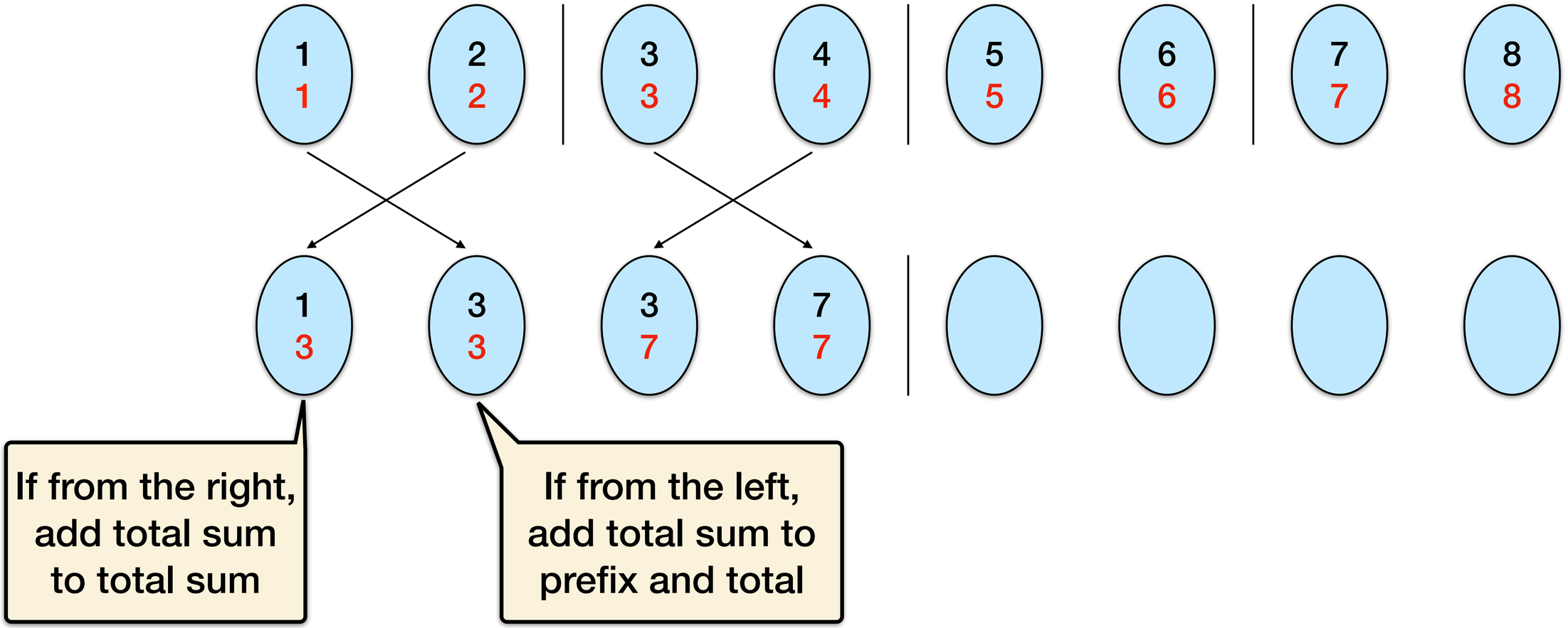


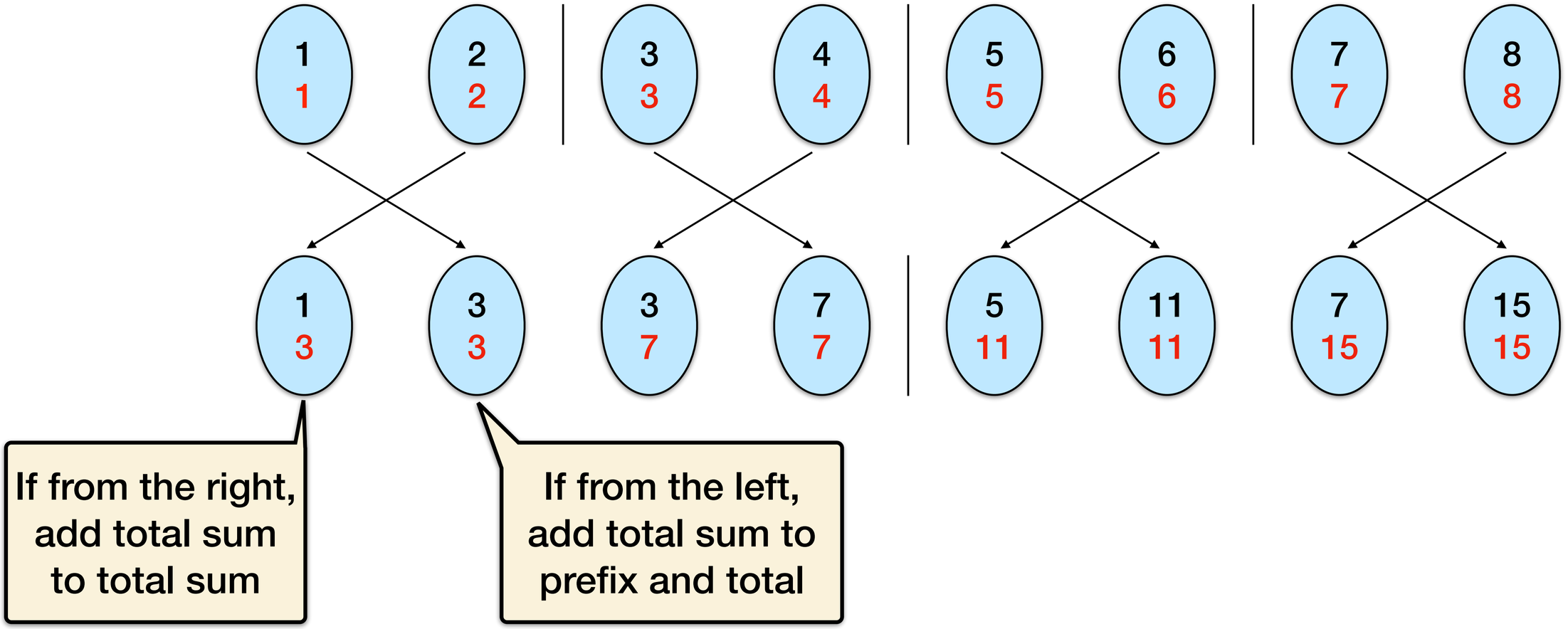
From Srinivas Aluru

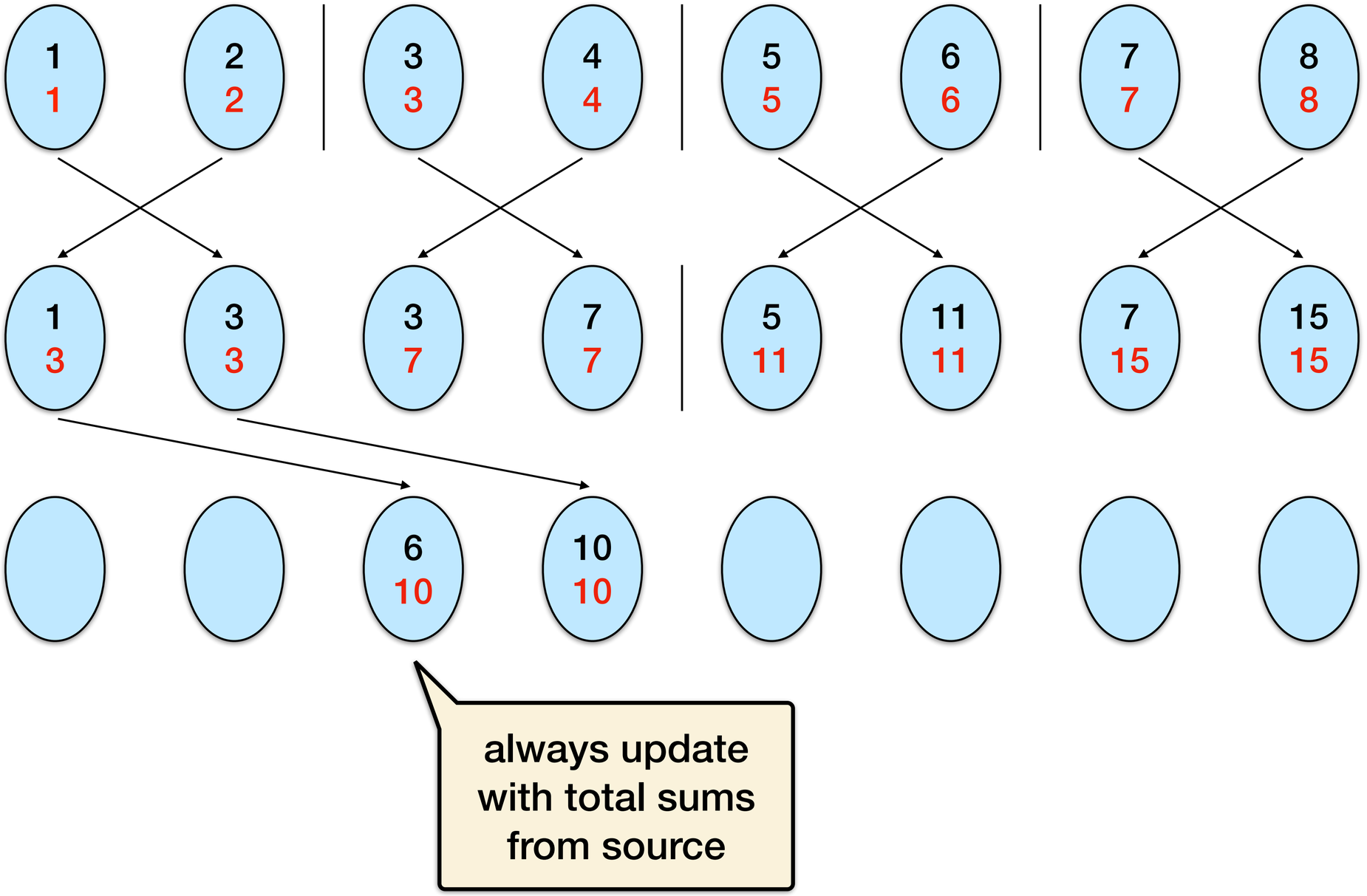

1
1
2
2
3
3
4
4
5
5
6
6
7
7
8
8
Initialize prefix and total sum as the input number

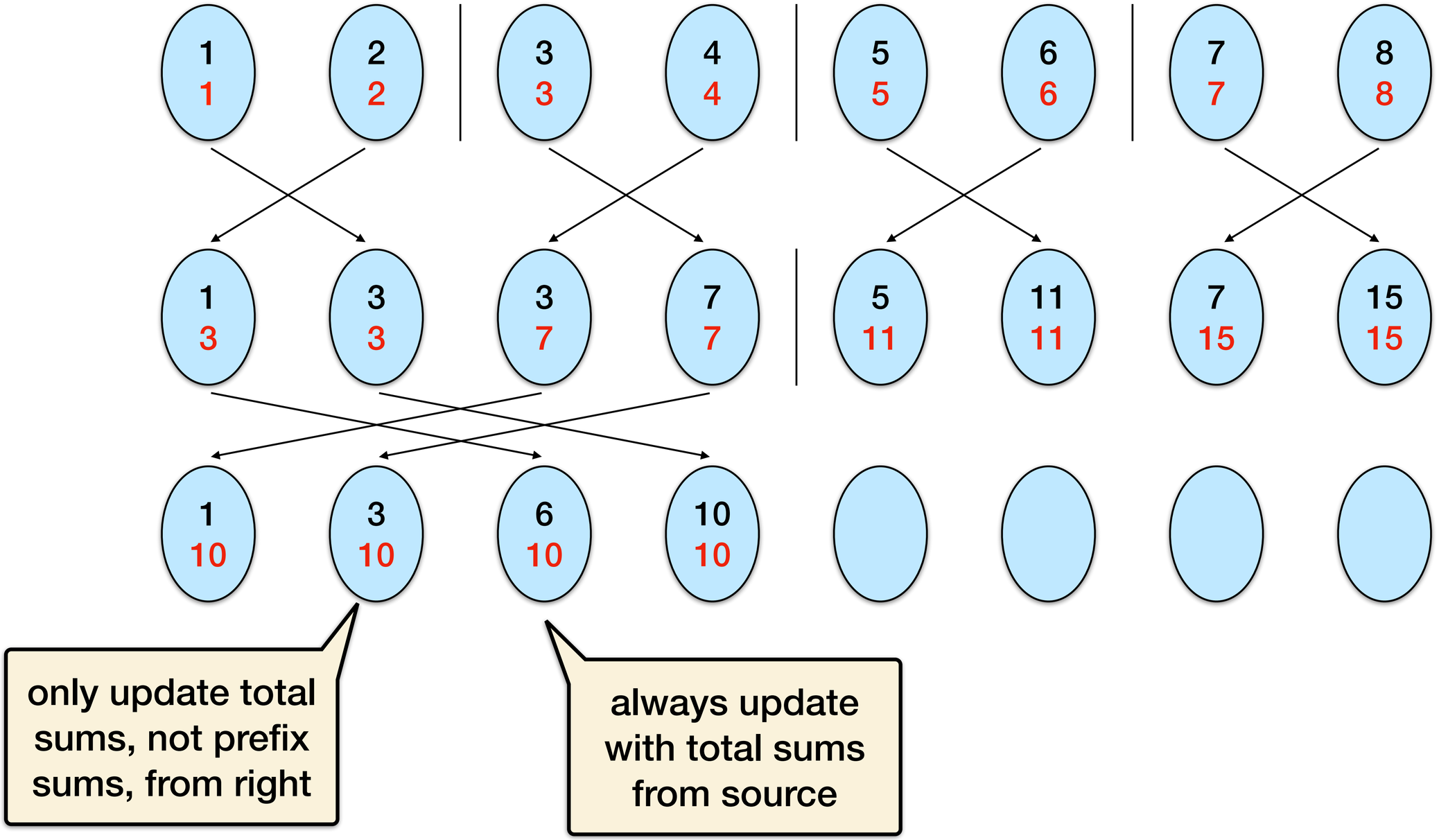

1
1
2
2
3
3
4
4
5
5
6
6
7
7
8
8
3
3
If from the left, add total sum to prefix and total

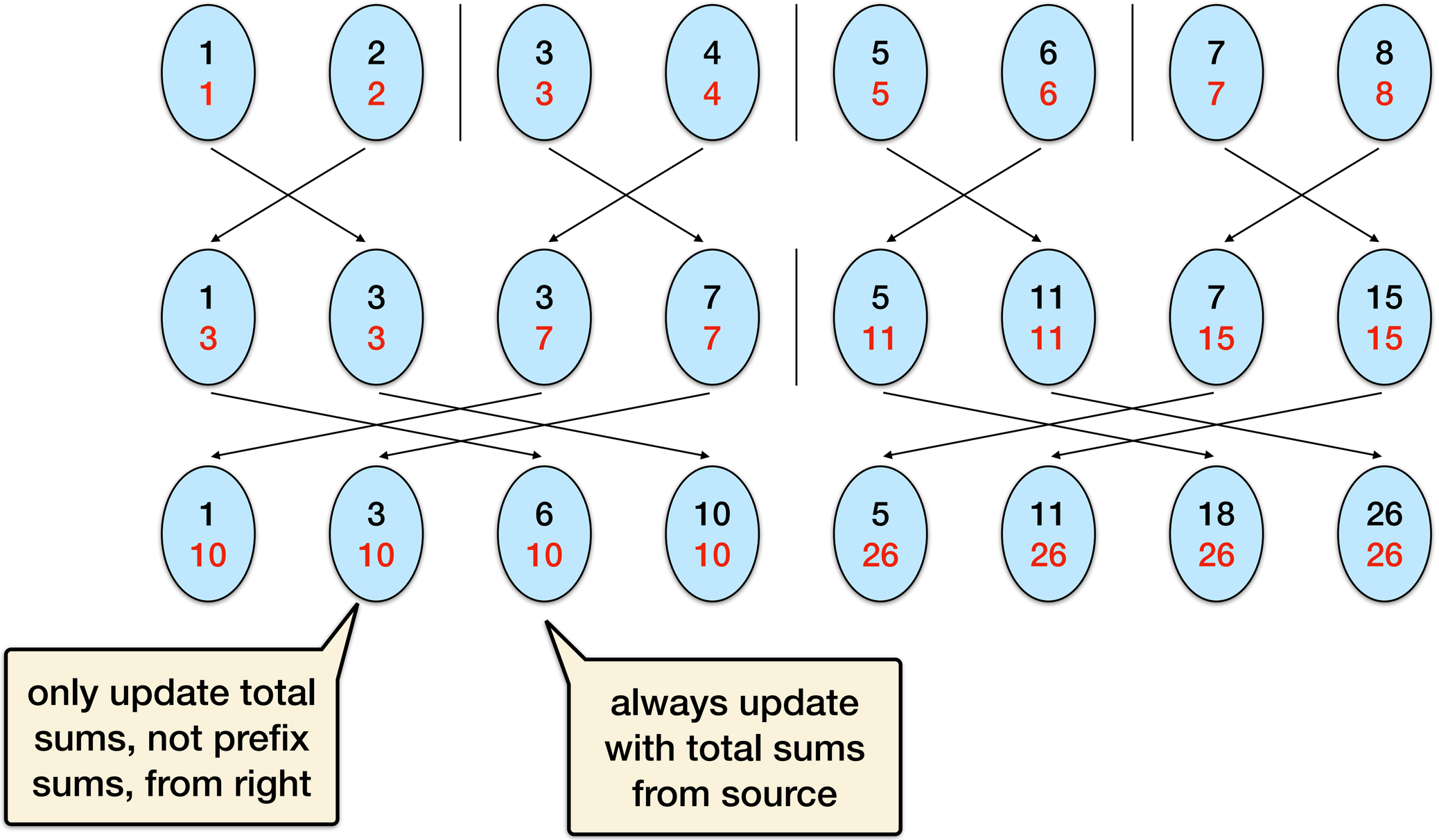

1
1
2
2
3
3
4
4
5
5
6
6
7
7
8
8
1
3
3
3
If from the right, add total sum to total sum
If from the left, add total sum to prefix and total

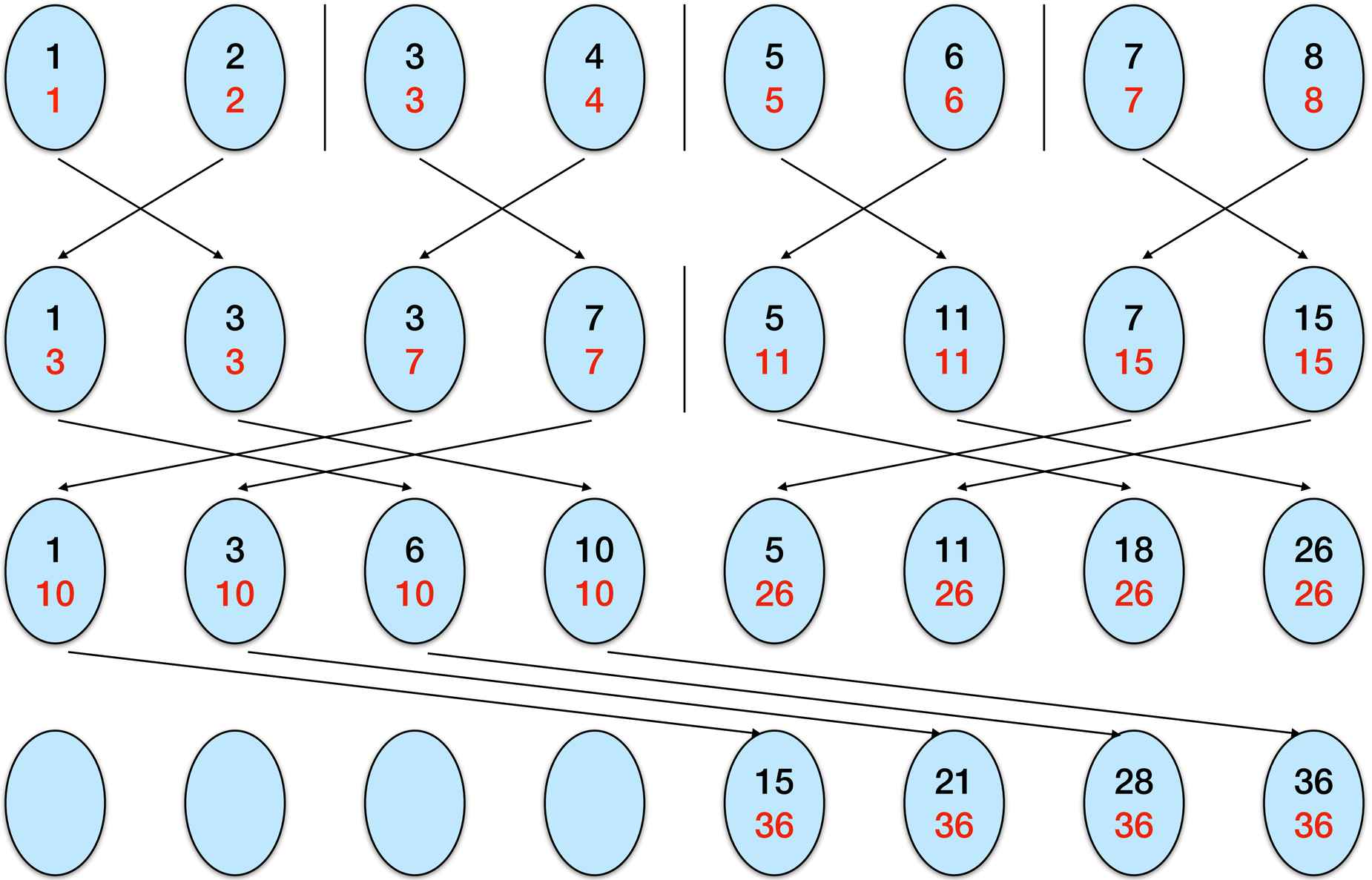

1
1
2
2
3
3
4
4
5
5
6
6
7
7
8
8
1
3
3
3
3
7
7
7
If from the right, add total sum to total sum
If from the left, add total sum to prefix and total

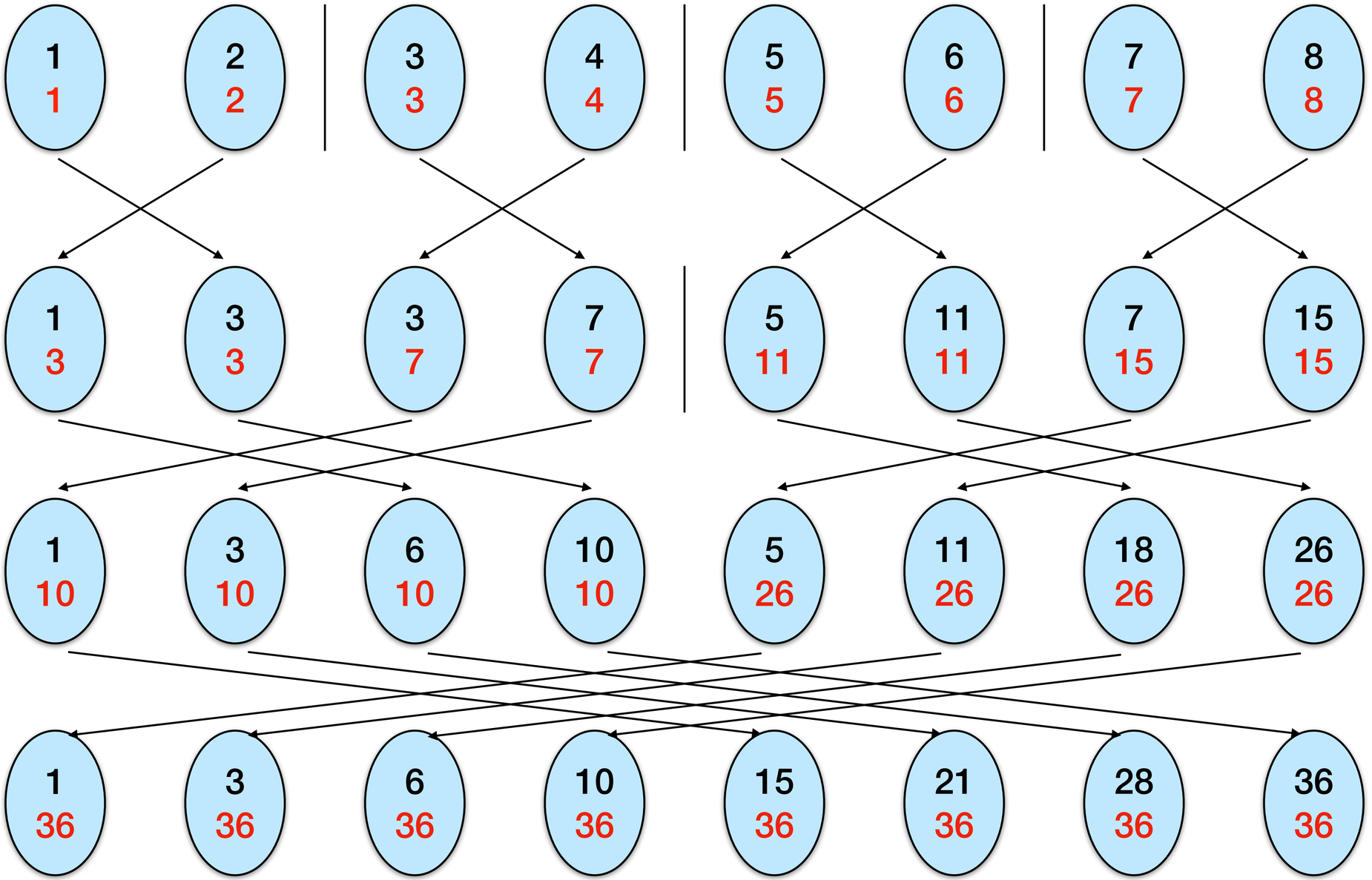

1
1
2
2
3
3
4
4
5
5
6
6
7
7
8
8
1
3
3
3
3
7
7
7
5
11
11
11
7
15
15
15
If from the right, add total sum to total sum
If from the left, add total sum to prefix and total


1
1
2
2
3
3
4
4
5
5
6
6
7
7
8
8
1
3
3
3
3
7
7
7
5
11
11
11
7
15
15
15
6
10
10
10
always update with total sums from source


1
1
2
2
3
3
4
4
5
5
6
6
7
7
8
8
1
3
3
3
7
7
5
11
11
11
7
15
15
15
1
10
3
10
6
10
10
10
only update total sums, not prefix sums, from right
always update with total sums from source


1 1
2 2
3 3
4 4
5 5
6 6
7 7
8 8
1 3
3 3
3 7
7 7
5 11
11 11
7 15
15 15
1 10
3 10
6 10
10 10
5 26
11 26
18 26
26 26
only update total sums, not prefix sums, from right
always update with total sums from source


1 1
2 2
3 3
4 4
5 5
6 6
7 7
8 8
1 3
3 3
3 7
7 7
5 11
11 11
7 15
15 15
1 10
3 10
6 10
10 10
5 26
11 26
18 26
26 26
15 36
21 36
28 36
36 36


1 1
2 2
3 3
4 4
5 5
6 6
7 7
8 8
1 3
3 3
3 7
7 7
5 11
11 11
7 15
15 15
1 10
3 10
6 10
10 10
5 26
11 26
18 26
26 26
1 36
3 36
6 36
10 36
15 36
21 36
28 36
36 36

Pseudocode for Parallel Prefix

```
Algorithm (for P_i)
total_sum ← prefix_sum ← local_number
for j=0 to d-1 do
    rank' ← rank XOR 2^j
    send total_sum to rank'
    receive received_sum from rank'
    total_sum ← total_sum + received_sum
    if (rank > rank')
        prefix_sum ← prefix_sum + received_sum
endfor
```

Pseudocode for Parallel Prefix

```
Algorithm (for P_i)
total_sum ← prefix_sum ← local_number
for j=0 to d-1 do
    rank' ← rank XOR 2^j
    send total_sum to rank'
    receive received_sum from rank'
    total_sum ← total_sum + received_sum
    if (rank > rank')
        prefix_sum ← prefix_sum + received_sum
endfor
```

Initialize prefix and total sum as the input number

Pseudocode for Parallel Prefix

```
Algorithm (for Pi)
total_sum ← prefix_sum ← local_number
for j=0 to d-1 do
    rank' ← rank XOR 2^j
    send total_sum to rank'
    receive received_sum from rank'
    total_sum ← total_sum + received_sum
    if (rank > rank')
        prefix_sum ← prefix_sum + received_sum
endfor
```

Initialize prefix and total sum as the input number

Find neighbor in this round

Pseudocode for Parallel Prefix

```
Algorithm (for Pi)
total_sum ← prefix_sum ← local_number
for j=0 to d-1 do
    rank' ← rank XOR 2^j
    send total_sum to rank'
    receive received_sum from rank'
    total_sum ← total_sum + received_sum
    if (rank > rank')
        prefix_sum ← prefix_sum + received_sum
endfor
```

Initialize prefix and total sum as the input number

Find neighbor in this round

Send/receive total sums

Pseudocode for Parallel Prefix

```
Algorithm (for Pi)
total_sum ← prefix_sum ← local_number
for j=0 to d-1 do
    rank' ← rank XOR 2^j
    send total_sum to rank'
    receive received_sum from rank'
    total_sum ← total_sum + received_sum
    if (rank > rank')
        prefix_sum ← prefix_sum + received_sum
endfor
```

Initialize prefix and total sum as the input number

Find neighbor in this round

Send/receive total sums

Always update total sum

Pseudocode for Parallel Prefix

```
Algorithm (for P_i)
total_sum ← prefix_sum ← local_number
for j=0 to d-1 do
    rank' ← rank XOR 2^j
    send total_sum to rank'
    receive received_sum from rank'
    total_sum ← total_sum + received_sum
    if (rank > rank')
        prefix_sum ← prefix_sum + received_sum
endfor
```

Initialize prefix and total sum as the input number

Find neighbor in this round

Send/receive total sums

Always update total sum

Update prefix sum if sum is coming from the left

Pseudocode for Parallel Prefix

```
Algorithm (for Pi)
total_sum ← prefix_sum ← local_number
for j=0 to d-1 do
    rank' ← rank XOR 2^j
    send total_sum to rank'
    receive received_sum from rank'
    total_sum ← total_sum + received_sum
    if (rank > rank')
        prefix_sum ← prefix_sum + received_sum
endfor
```

Initialize prefix and total sum as the input number

Find neighbor in this round

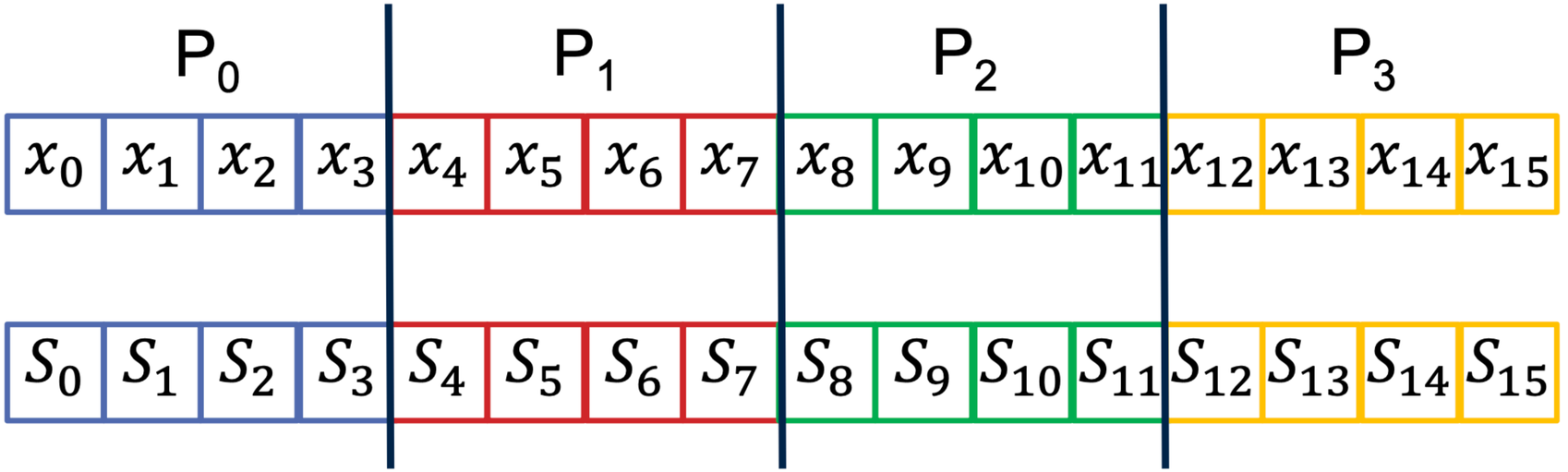
Send/receive total sums

Always update total sum

Update prefix sum if sum is coming from the left

Next: what if n > p?

Parallel Prefix with Fewer Processors



- Use Brent's Lemma?
- $T(n,p) = \Theta\left(\frac{n}{p}\log n\right)$

Algorithm Summary

1. Compute prefix sum locally on each processor

2. Perform parallel prefix sum using the last local prefix sum on each processor

3. Add the result of parallel prefix sum on the processor to each of its local prefix sums

Computation time = $\Theta(\frac{n}{p} + \lg p)$

Communication time = $\Theta((\tau + \mu)\lg p)$

Bandwidth in s/byte for convenience

From Srinivas Aluru

Algorithm Summary

1. Compute prefix sum locally on each processor
2. Perform parallel prefix sum using the last local prefix sum on each processor
3. Add the result of parallel prefix sum on the processor to each of its local prefix sums

Is this correct?

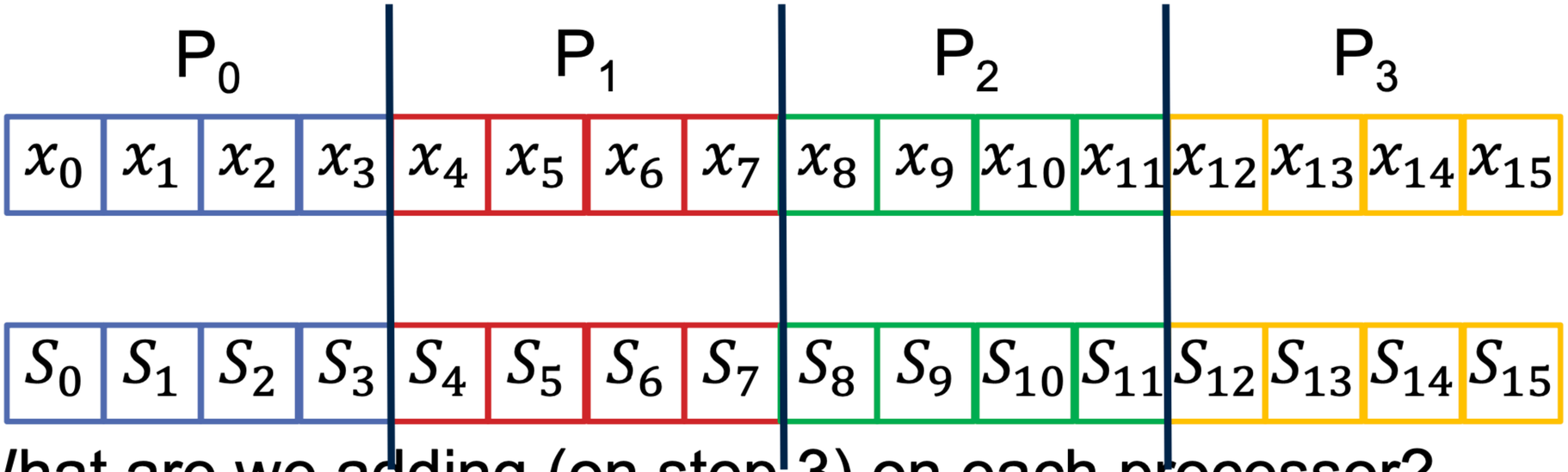
Computation time = $\Theta(\frac{n}{p} + \lg p)$

Communication time = $\Theta((\tau + \mu)\lg p)$

Bandwidth in s/byte for convenience

From Srinivas Aluru

Algorithm Example



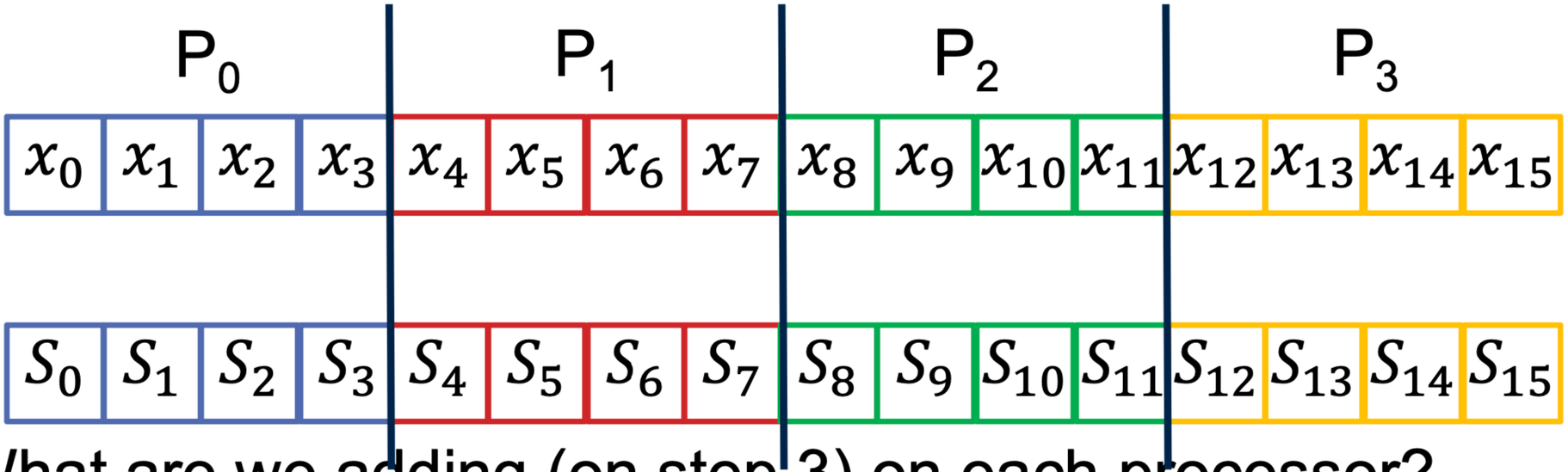
- What are we adding (on step 3) on each processor?
  - P_0 adds S_3
  - P_1 adds S_7
  - P_2 adds S_{11}
  - P_3 adds S_{15}

Should not be adding end to self

Ideas about how to fix it?

From Srinivas Aluru

Algorithm Example



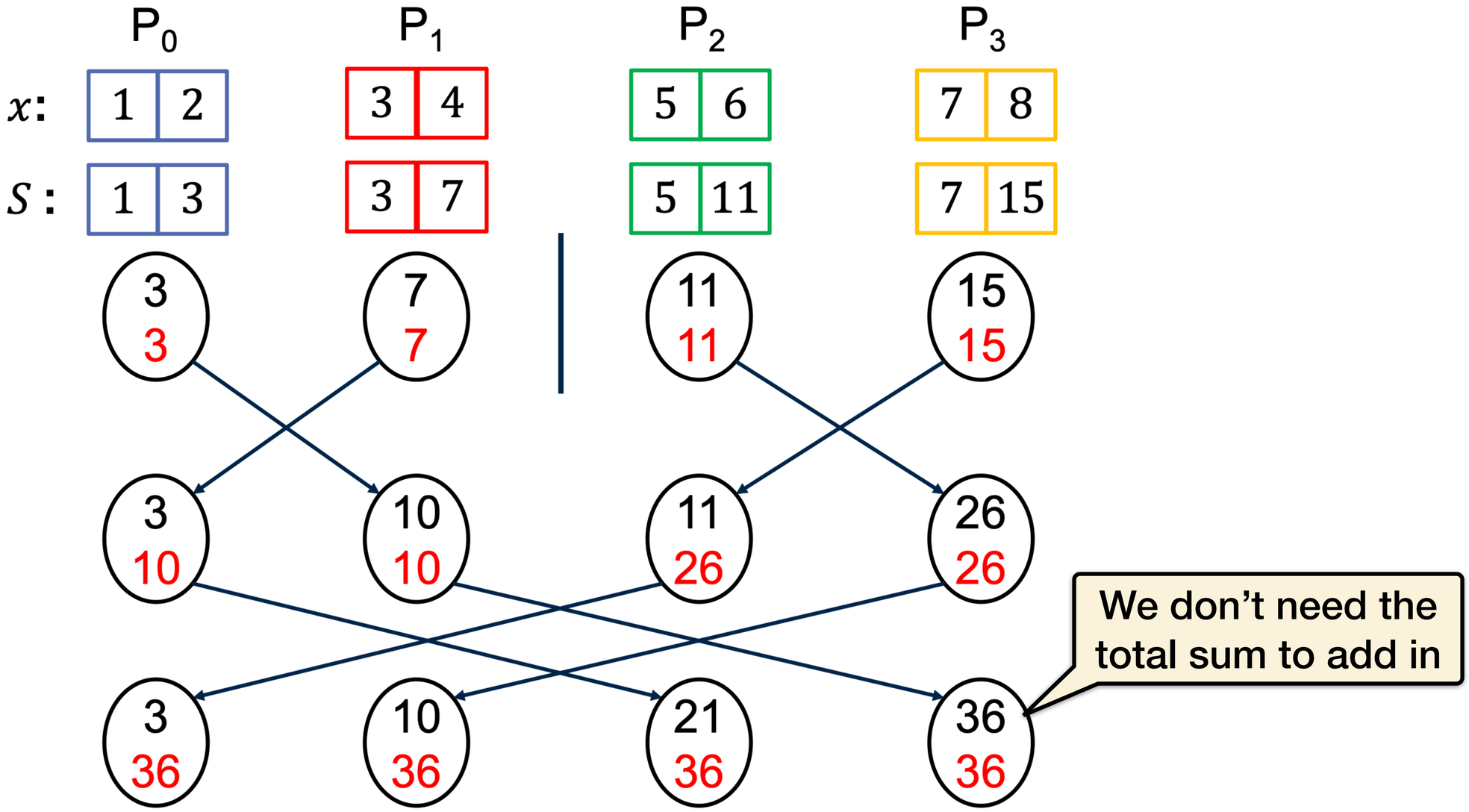
- What are we adding (on step 3) on each processor?
  - P_0 adds S_3
  - P_1 adds S_7
  - P_2 adds S_{11}
  - P_3 adds S_{15}

Should not be adding end to self

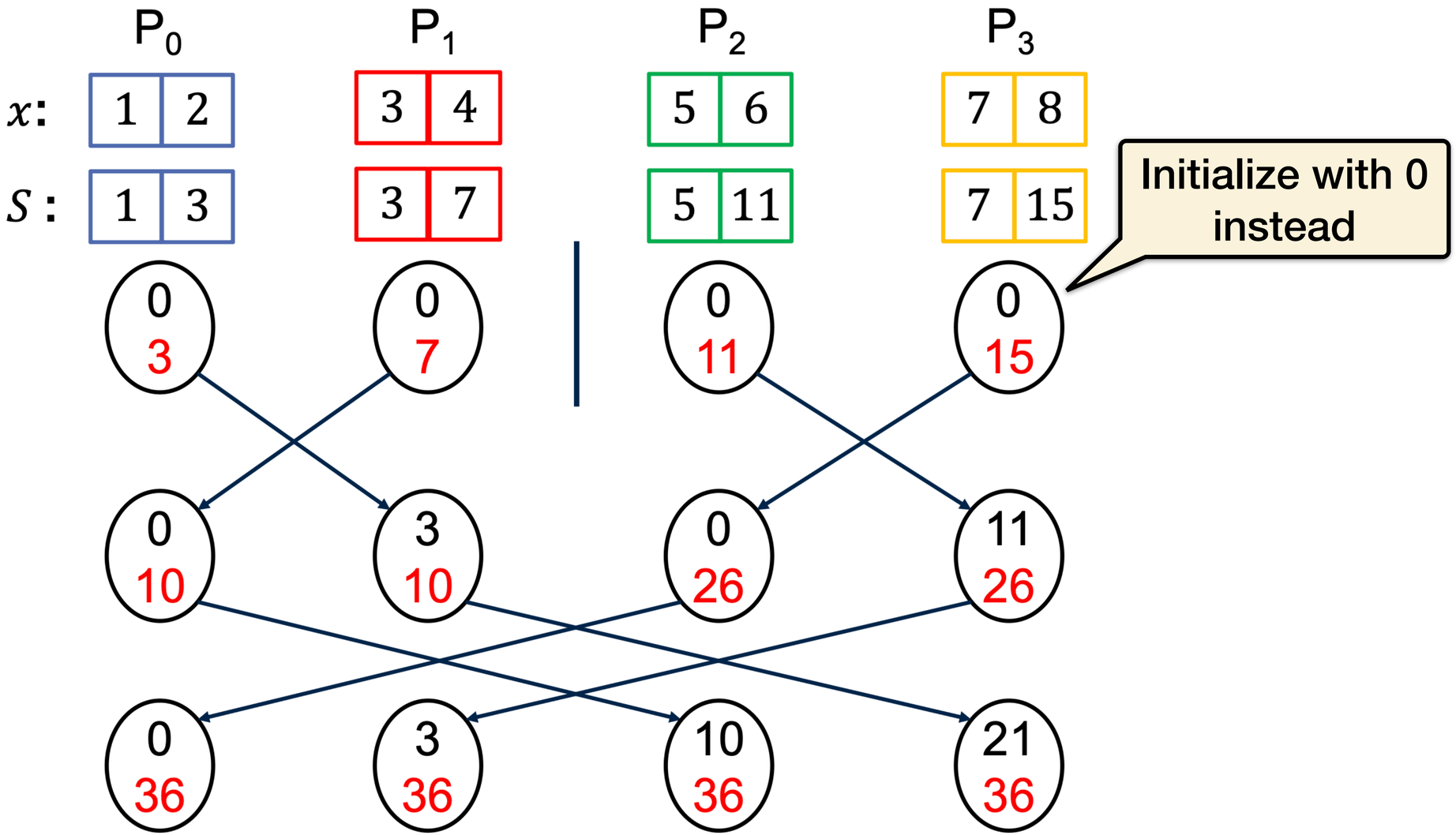
Potential solution: Exclusive scan

From Srinivas Aluru

With Outer Inclusive Scan



With Outer Exclusive Scan



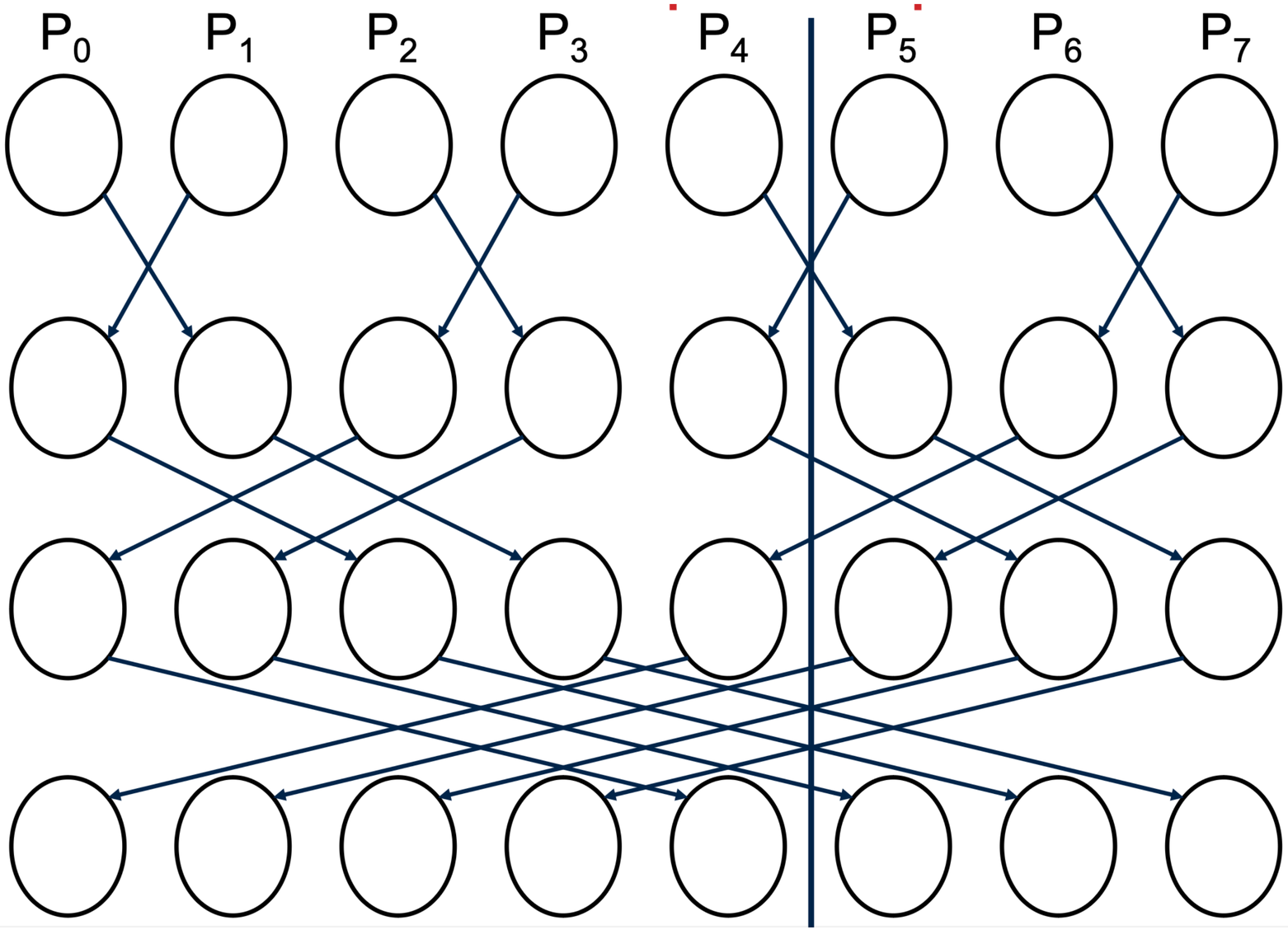
Generalizing Parallel Prefix

If n is not divisible by p, assign some processors with 1 more element than the others.

What if p is not a power of 2?

- Find p' = a power of 2 such that $\frac{p'}{2} < p < p'$.
- Run your code like you have p' processors.
- Ignore communications to/from non-existing processors (i.e., rank $\geq p$)

Parallel Prefix on Non-Power-of-2 Processors



From Srinivas Aluru

Applications of Parallel Prefix

A Partial List of Applications for Parallel Prefix

- Adding two n-bit integers in $O(\log n)$ time
- Evaluating polynomials
- Solving recurrences
- Radix sort
- “2D parallel prefix” for image segmentation
- Traversing linked lists
- and many others!

Application: Fibonacci via Matrix Multiply Prefix

$$F_{n+1} = F_n + F_{n-1}$$

$$\begin{pmatrix} F_{n+1} \\ F_n \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} F_n \\ F_{n-1} \end{pmatrix}$$

Can compute all F_n by `matmul_prefix` on

$$\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}, \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}, \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}, \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}, \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \cdots$$

$$\downarrow$$

$$\begin{pmatrix} 2 & 1 \\ 1 & 1 \end{pmatrix}, \begin{pmatrix} 3 & 2 \\ 2 & 1 \end{pmatrix}, \begin{pmatrix} 5 & 3 \\ 3 & 2 \end{pmatrix}, \begin{pmatrix} 8 & 5 \\ 5 & 3 \end{pmatrix}, \begin{pmatrix} 13 & 8 \\ 8 & 5 \end{pmatrix} \cdots$$

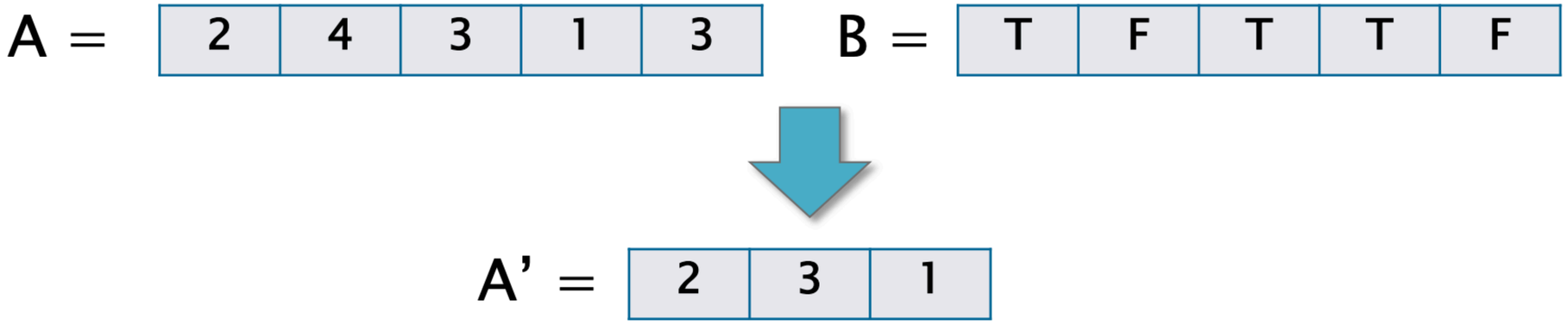
Select the upper left entry

The same idea works for any linear recurrence (e.g., random number generation).

Proof in “Prefix Sums and Their Applications,” Blelloch, 1990.

Application: Stream Compression (aka Filter)

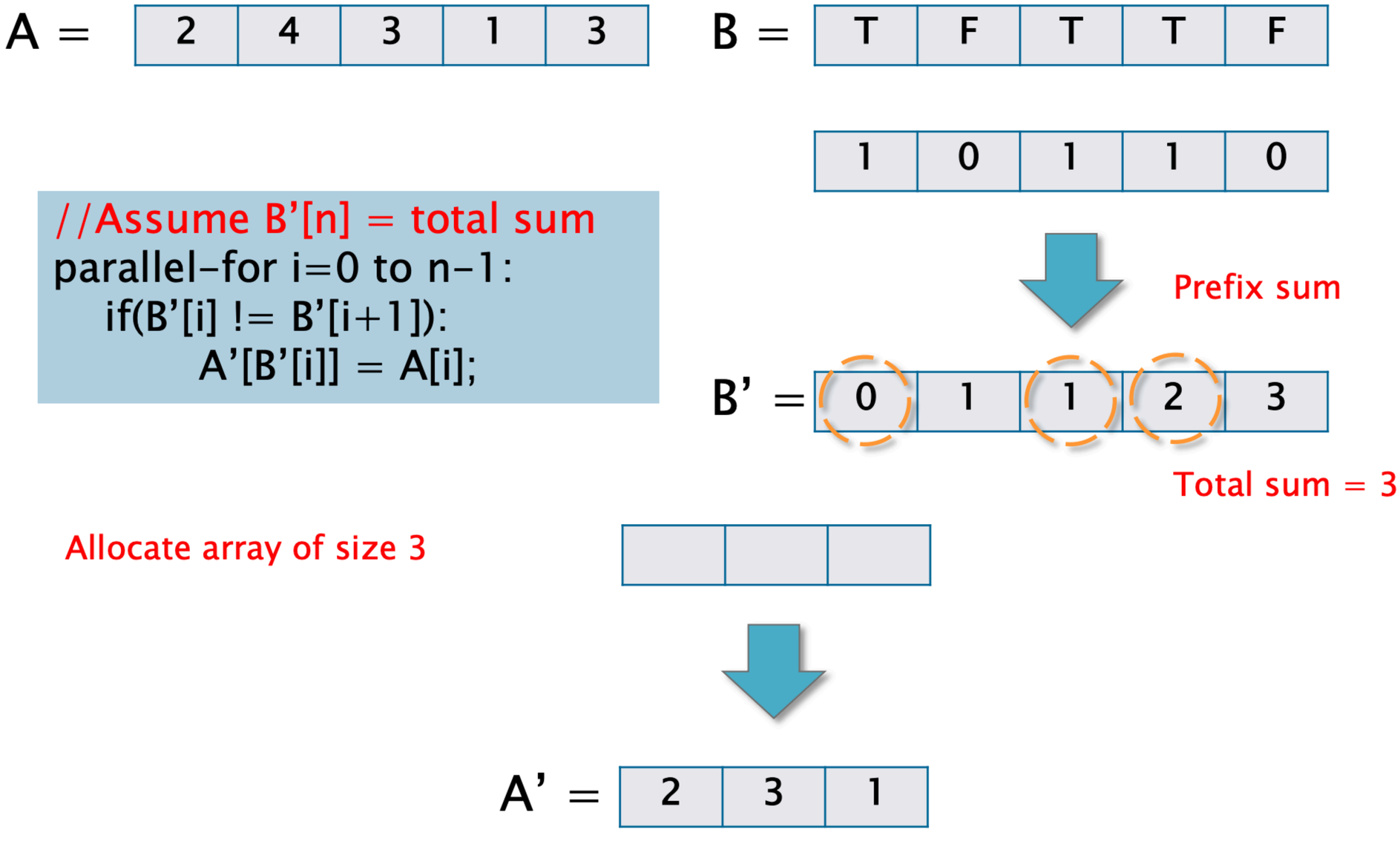
- Definition: Given a sequence $A=[x_0, x_1,\ldots, x_{n-1}]$ and a Boolean array of flags $B[b_0, b_1,\ldots, b_{n-1}]$, output an array A' containing just the elements A[i] where B[i] = true (maintaining relative order)
- Example:



- Can you implement filter using prefix sum?

Filter Implementation

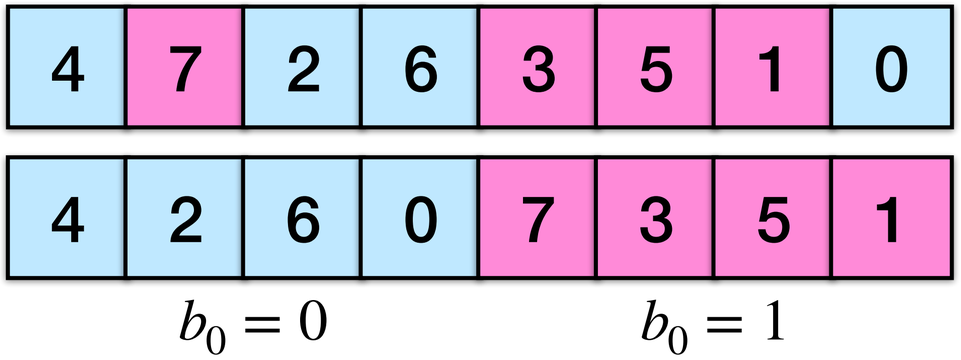
Can use to filter on some condition



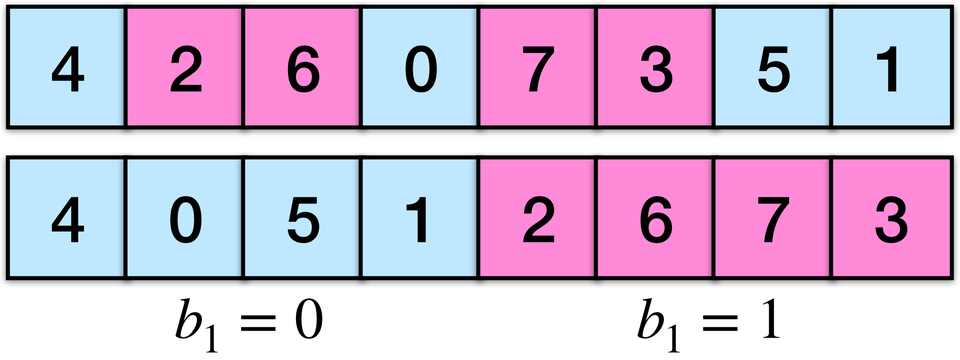

```
//Assume B'[n] = total sum
parallel-for i=0 to n-1:
    if(B'[i] != B'[i+1]):
        A'[B'[i]] = A[i];
```

FIRST LECTURE ENDED HERE

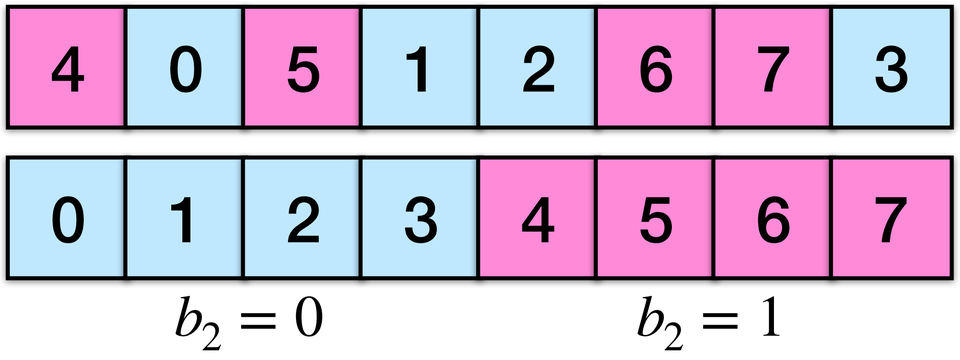
Application: Radix Sort (Serial)



Sort on least significant bit (b_0 in $b_2 b_1 b_0$)
XX0 < XX1 (evens before odds)

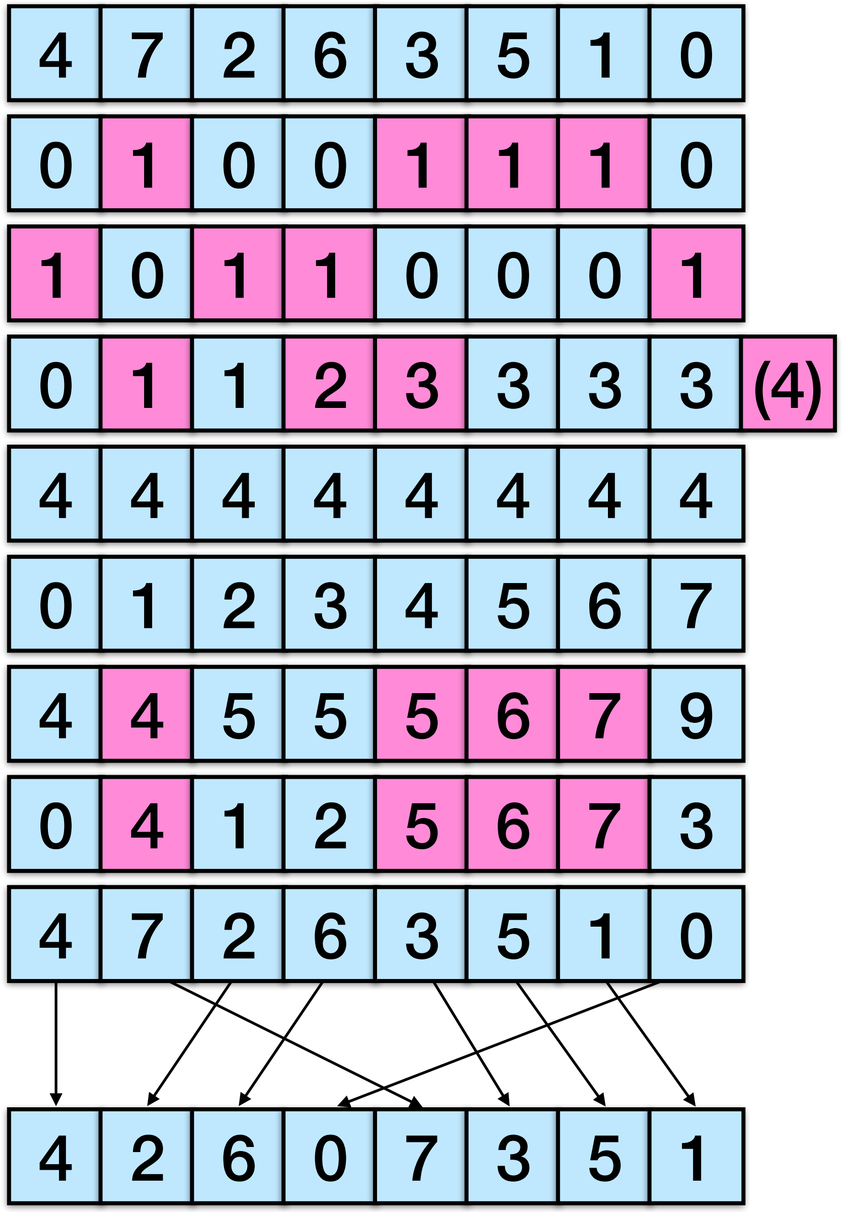


Stably sort entire array on next bit
X0X < X1X



Stably sort entire array on next bit
0XX < 1XX

Application: Data-Parallel Radix Sort



Input

Odds = last bit of each element

Evens = complement of odds

Even_positions = exclusive scan of evens

totalEvens = broadcast last element

index = constant array of 0 .. n

odd_positions = #evens + indx - even_pos

pos = get positions using masked assignment

Scatter input according to pos

(repeat with next bit until you are out of bits)

Analyzing Data-Parallel Radix Sort

ALGORITHM: RADIX_SORT(A, b)

1 **for** i **from** 0 **to** $b-1$
2 $\quad$ FLAGS $:= \{(a >> i) \ mod \ 2 : a \in A\}$
3 $\quad$ NOTFLAGS $:= \{1 - b : b \in$ FLAGS$\}$
4 $\quad$ $R_0 :=$ SCAN(NOTFLAGS)
5 $\quad$ $s_0 :=$ SUM(NOTFLAGS)
6 $\quad$ $R_1 :=$ SCAN(FLAGS)
7 $\quad$ $R := \{$**if** FLAGS$[j] = 0$ **then** $R_0[j]$ **else** $R_1[j] + s_0 : j \in [0..|A|)\}$
8 $\quad$ $A := A \leftarrow \{(R[j], A[j]) : j \in [0..|A|)\}$
9 **return** A

What is the work and span?

Analyzing Data-Parallel Radix Sort

$O(\log n)$ span

```
ALGORITHM: RADIX_SORT(A, b)
1  for i from 0 to b − 1
2     FLAGS := {(a >> i) mod 2 : a ∈ A}
3     NOTFLAGS := {1 − b : b ∈ FLAGS}
4     R_0 := SCAN(NOTFLAGS)
5     s_0 := SUM(NOTFLAGS)
6     R_1 := SCAN(FLAGS)
7     R := {if FLAGS[j] = 0 then R_0[j] else R_1[j] + s_0 : j ∈ [0..|A|)}
8     A := A ← {(R[j], A[j]) : j ∈ [0..|A|)}
9  return A
```

What is the work and span?

Work: There are b iterations of the for loop, and iteration takes $O(n)$ work, so the total work is $O(bn)$.

Span: There are b iterations of the for loop, and iteration has $O(\log n)$ span, so the total span is $O(b \log n)$.

Application: Adding n-bit Integers

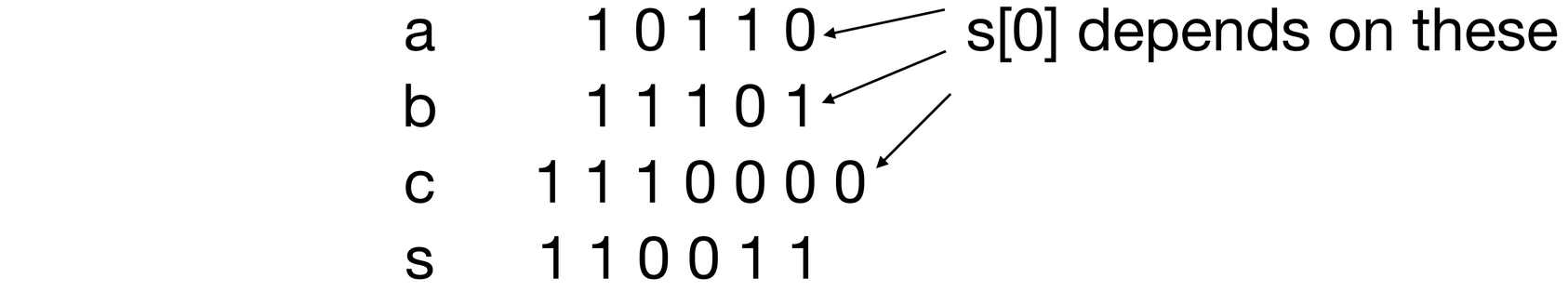
Problem: Computing sum of two n-bit binary numbers, a and b.

$a = a_{n-1}a_{n-2}\ldots a_0$ and $b = b_{n-1}b_{n-2}\ldots b_0$

$s = a + b = s_n s_{n-1}\ldots s_0$ (using carry bit array $c = c_{n-1}, \ldots, c_0, c_{-1}$)

```
c[-1] = 0 // rightmost carry bit
for i = 0 to n - 1: //compute right to left
  s[i] = (a[i] xor b[i]) xor c[i-1] // one or three 1s
  c[i] = ((a[i] xor b[i]) and c[i-1]) or (a[i] and b[i]) // next carry bit
```

Example:
a = 22
b = 29



Goal: Compute all c_i in $O(\log n)$ span via parallel prefix

Application: Adding n-bit Integers

```
c[-1] = 0 // rightmost carry bit
for i = 0 to n - 1: //compute right to left
  c[i] = ((a[i] xor b[i]) and c[i-1]) or (a[i] and b[i]) // next carry bit
```

Idea: Split carry bit into two cases that indicate information about the carry-out (c_i) regardless of carry-in (c_{i-1}):

- Generate (g_i): This column will generate a carry-out **whether or not** the carry-in is 1.

$$g_i = a_i \,\&\&\, b_i$$

- Propagate (p_i): This column will propagate a carry-in **if there is one** to the carry-out.

$$p_i = a_i \,||\, b_i$$

can be computed in parallel

$$c_i = g_i + p_i c_{i-1}$$

Carry Lookahead Logic

Idea: Define each carry-in in terms of p_i, g_i and the initial carry in c_{i-1} and not in terms of carry chain (i.e., unwind the recursion):

- $c_0 = g_0 + p_0 c_{-1}$
- $c_1 = g_1 + p_1 c_0 = g_1 + p_1 g_0 + p_1 p_0 c_{-1}$
- …

Can be expressed with 2-by-2 boolean matrix multiplication:

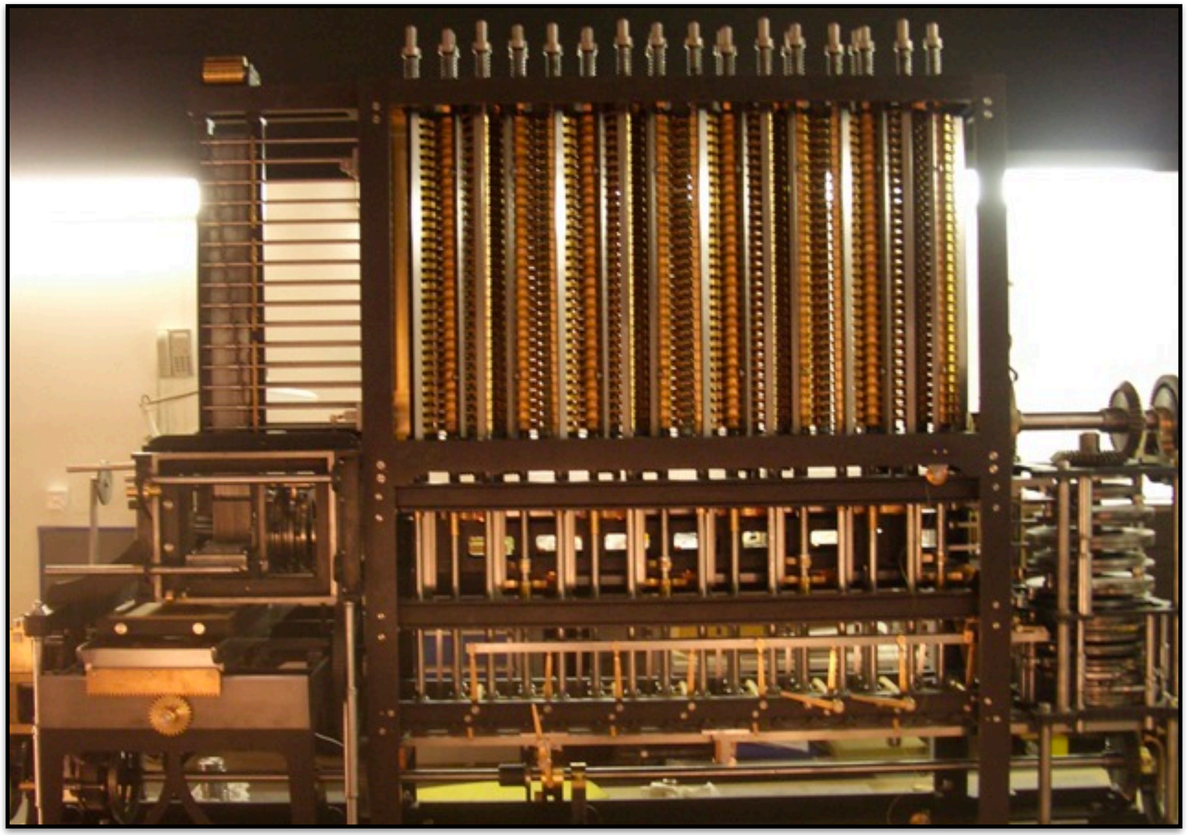
$$M[i] = \begin{pmatrix} p[i] & g[i] \\ 0 & 1 \end{pmatrix}$$

$$\begin{pmatrix} c[i] \\ 1 \end{pmatrix} = M[i] \times M[i-1] \times \ldots \times M[0] \times \begin{pmatrix} 0 \\ 1 \end{pmatrix}$$

Carry-lookahead addition is used in all computers

This idea is used in all hardware

The design goes back to Babbage in the 1800s:



https://community.cadence.com/cadence_blogs_8/b/breakfast-bytes/posts/carry2

Application: Lexical Analysis

Lexical analysis **divides a long string of characters into tokens** - often the first thing a compiler does when processing a program.

Suppose we have a regular language - we can represent it with a **finite-state automaton** that begins in a certain state and makes transitions between states based on the characters read.

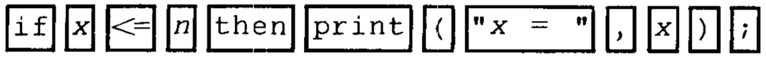


TABLE I. A Finite-State Automaton for Recognizing Tokens

Old State	Character Read													
•	A	B	...	Y	Z	+	−	*	<	>	=	"	Space	New line
N	A	A	...	A	A	*	*	*	<	<	*	Q	N	N
A	Z	Z	...	Z	Z	*	*	*	<	<	*	Q	N	N
Z	Z	Z	...	Z	Z	*	*	*	<	<	*	Q	N	N
*	A	A	...	A	A	*	*	*	<	<	*	Q	N	N
<	A	A	...	A	A	*	*	*	<	<	=	Q	N	N
=	A	A	...	A	A	*	*	*	<	<	*	Q	N	N
Q	S	S	...	S	S	S	S	S	S	S	S	E	S	S
S	S	S	...	S	S	S	S	S	S	S	S	E	S	S
E	E	E	...	E	E	*	*	*	<	<	*	S	N	N

Seems to depend on the previous state, which depends on characters read up to some point

Goal: Perform lexical analysis in parallel

"Data-Parallel Algorithms," Hillis and Steele, CACM 1986.

Application: Lexical Analysis

Idea: replace every character in the string with the array representation of its **state-to-state function** (column).

Then perform a parallel-prefix operation with ⊕ as the **array composition**. Each character becomes an array representing the **state-to-state function** for that prefix.

Use the **initial state** to index into the arrays.

TABLE I. A Finite-State Automaton for Recognizing Tokens

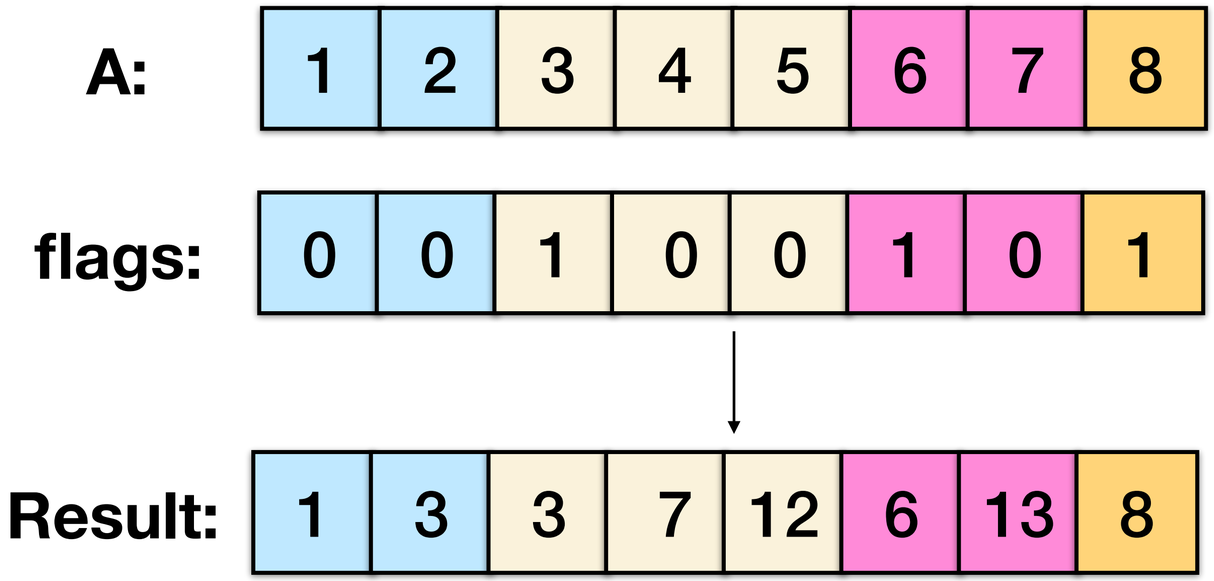
Old State	Character Read													
•	A	B	...	Y	Z	+	−	*	<	>	=	"	Space	New line
N	A	A	...	A	A	*	*	*	<	<	*	Q	N	N
A	Z	Z	...	Z	Z	*	*	*	<	<	*	Q	N	N
Z	Z	Z	...	Z	Z	*	*	*	<	<	*	Q	N	N
*	A	A	...	A	A	*	*	*	<	<	*	Q	N	N
<	A	A	...	A	A	*	*	*	<	<	=	Q	N	N
=	A	A	...	A	A	*	*	*	<	<	*	Q	N	N
Q	S	S	...	S	S	S	S	S	S	S	S	E	S	S
S	S	S	...	S	S	S	S	S	S	S	S	E	S	S
E	E	E	...	E	E	*	*	*	<	<	*	S	N	N

Example of state-to-state function for the space character

"Data-Parallel Algorithms," Hillis and Steele, CACM 1986.

Application: Segmented Scans

Inputs: value array, flag array, associative operator $\oplus$



Can be used to parallelize sparse-matrix vector multiply (SpMV)

Application: Image Processing

- A **summed-area table** is an algorithm/data structure for quickly generating the sum of values in some rectangular subset of a grid.
- Often used in image processing [Crow, 84].
- Computes **prefix sums in both dimensions**, and then **inclusion-exclusion on the corners** to compute the sum within any rectangular area.

$$I(x, y) = \sum_{x' \le x, y' \le y} i(x', y')$$

$$A = (x_0, y_0), B = (x_1, y_0), C = (x_0, y_1), D = (x_1, y_1)$$

$$\sum_{x_0 < x \le x_1, y_0 < y \le y_1} i(x, y) = I(D) + I(A) - I(B) - I(C)$$

1.

31	2	4	33	5	36
12	26	9	10	29	25
13	17	21	22	20	18
24	23	15	16	14	19
30	8	28	27	11	7
1	35	34	3	32	6

2.

31	33	37	70	75	111
43	71	84	127	161	222
56	101	135	200	254	333
80	148	197	278	346	444
110	186	263	371	450	555
111	222	333	444	555	666

15 + 16 + 14 + 28 + 27 + 11 =
101 + 450 – 254 – 186 = 111

https://en.wikipedia.org/wiki/Summed-area_table

Application: Image Processing

- A **summed-area table** is an algorithm/data structure for quickly generating the sum of values in some rectangular subset of a grid.
- Often used in image processing [Crow, 84].
- Computes **prefix sums in both dimensions**, and then **inclusion-exclusion on the corners** to compute the sum within any rectangular area.

$$I(x, y) = \sum_{x' \le x, y' \le y} i(x', y')$$

$$A = (x_0, y_0), B = (x_1, y_0), C = (x_0, y_1), D = (x_1, y_1)$$

$$\sum_{x_0 < x \le x_1, y_0 < y \le y_1} i(x, y) = I(D) + I(A) - I(B) - I(C)$$

Requires inverse

1.

31	2	4	33	5	36
12	26	9	10	29	25
13	17	21	22	20	18
24	23	15	16	14	19
30	8	28	27	11	7
1	35	34	3	32	6

2.

31	33	37	70	75	111
43	71	84	127	161	222
56	101	135	200	254	333
80	148	197	278	346	444
110	186	263	371	450	555
111	222	333	444	555	666

15 + 16 + 14 + 28 + 27 + 11 =
101 + 450 – 254 – 186 = 111

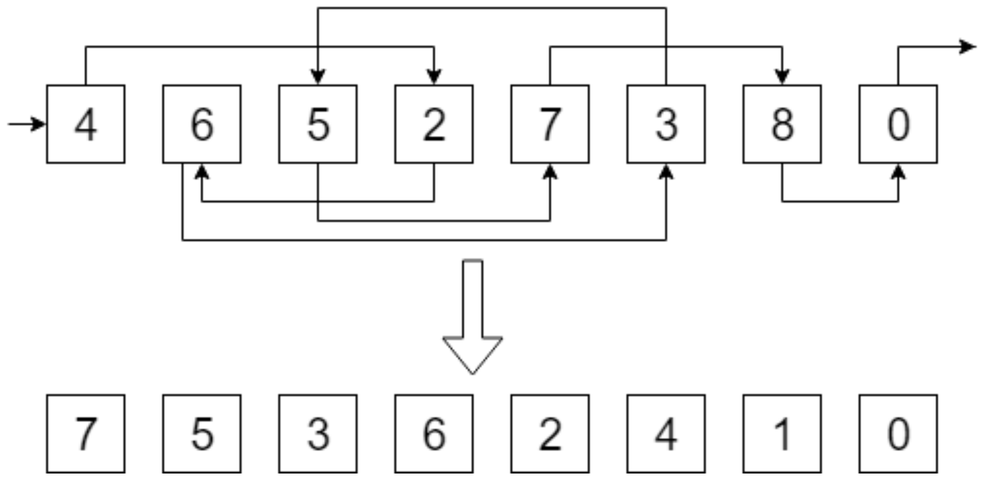
https://en.wikipedia.org/wiki/Summed-area_table

Application: List Ranking

- The list ranking problem is to determine the **distance from every node in a singly-linked list to the end**.

- In serial, the problem can be solved by traversing the list in $O(n)$ time.

- List ranking underlies many applications like prefix sum over linked lists, and tree algorithms.

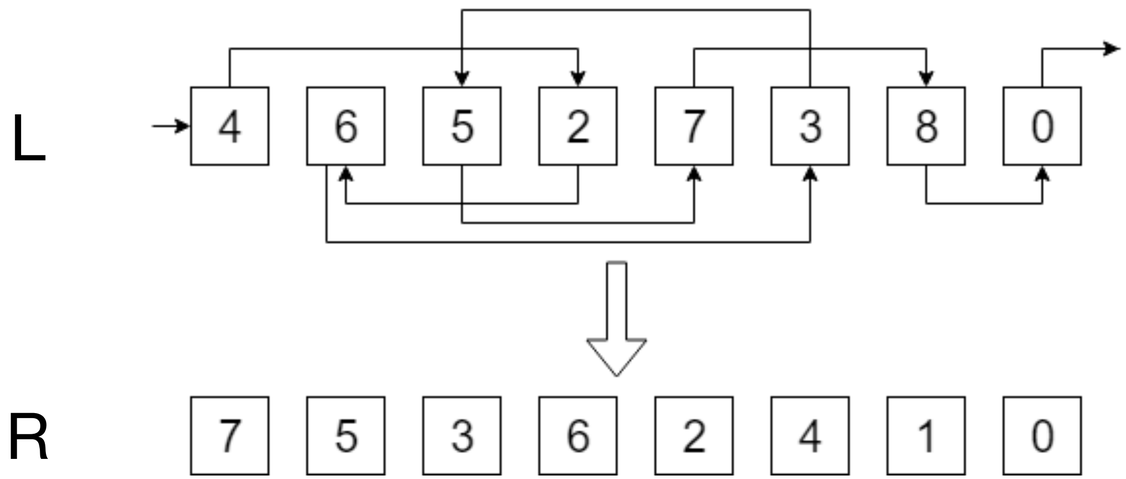


https://en.algorithmica.org/hpc/external-memory/list-ranking/

Application: List Ranking

Input: A linked list L of n nodes with an array S specifying their order. That is, every entry in the list L is a pointer to the index of its successor (or 0, if it is the end node).

Output: A list R of length n containing the rank (distance from the end of the list) of each element.



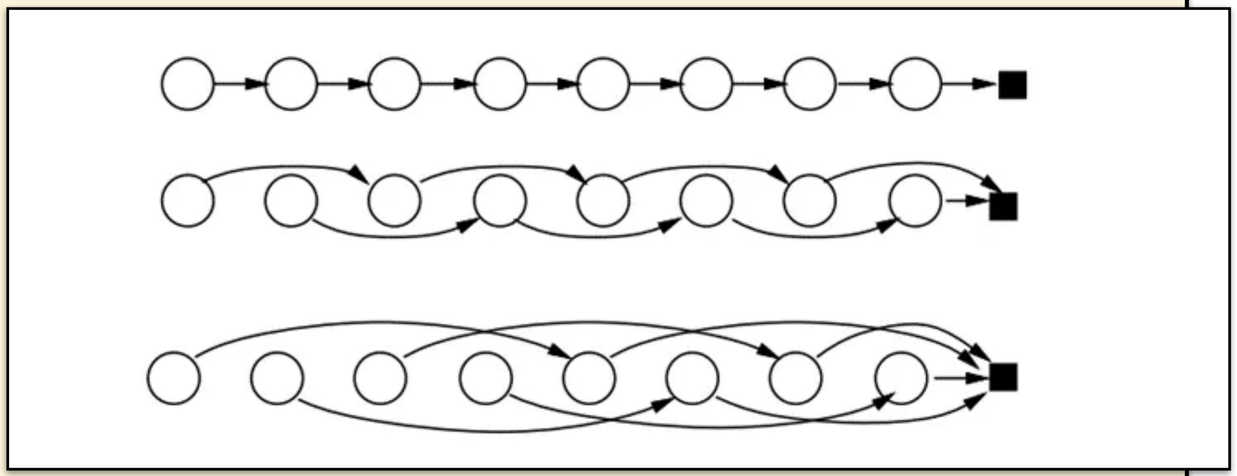
https://en.algorithmica.org/hpc/external-memory/list-ranking/

Wyllie's Algorithm for List Ranking

```
L = list of input successor pointers in an array
R = output list of ranks (distance from end)

parallel_for i = 0 to n-1:
  if L[i] != 0: R[i] = 1
  else: R[i] = 0

for j = 0 to ceil(lg n) - 1:
  temp, temp2;
  parallel_for i = 0 to n-1:
    temp[i] = R[L[i]];
    temp2[i] = L[L[i]];
  parallel_for i = 0 to n-1:
    R[i] = R[i] + temp[i];
    L[i] = temp2[i];
```



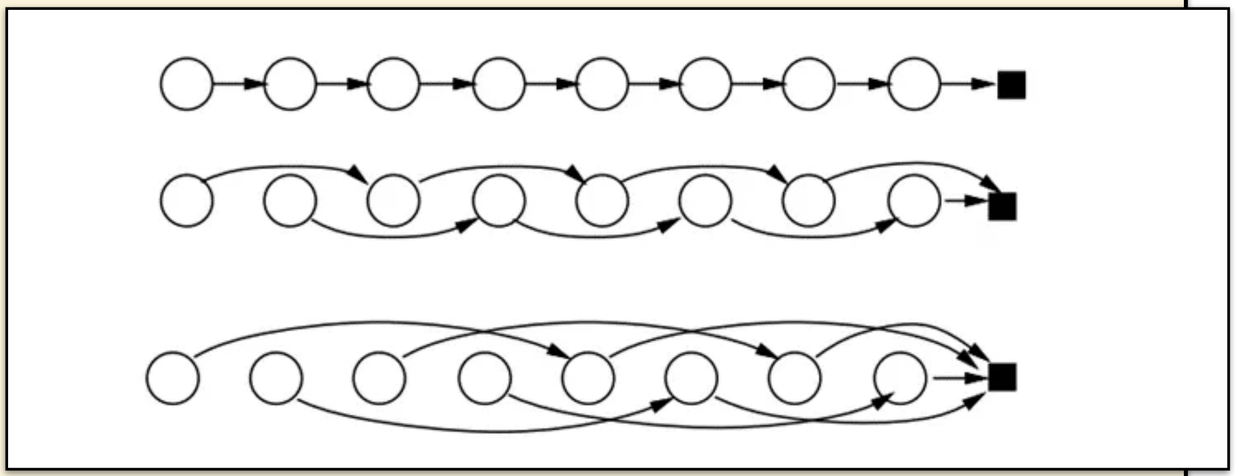
https://sneha-gupta180902.medium.com/parallel-list-ranking-c455e2849ab

Wyllie's Algorithm for List Ranking

```
L = list of input successor pointers in an array
R = output list of ranks (distance from end)

parallel_for i = 0 to n-1:
  if L[i] != 0: R[i] = 1
  else: R[i] = 0

for j = 0 to ceil(lg n) - 1:
  temp, temp2;
  parallel_for i = 0 to n-1:
    temp[i] = R[L[i]];
    temp2[i] = L[L[i]];
  parallel_for i = 0 to n-1:
    R[i] = R[i] + temp[i];
    L[i] = temp2[i];
```



What is the work and the span?

https://sneha-gupta180902.medium.com/parallel-list-ranking-c455e2849ab

Work-Span Analysis

```
L = list of input successor pointers in an array
R = output list of ranks (distance from end)

parallel_for i = 0 to n-1:
  if L[i] != 0: R[i] = 1
  else: R[i] = 0

for j = 0 to ceil(lg n) - 1:
  temp, temp2;
  parallel_for i = 0 to n-1:
    temp[i] = R[L[i]];
    temp2[i] = L[L[i]];
  parallel_for i = 0 to n-1:
    R[i] = R[i] + temp[i];
    L[i] = temp2[i];
```

Not work-efficient: sequential algorithm only requires O(n) work

Work = $O(n \log n)$

Span = $O(\log n)$