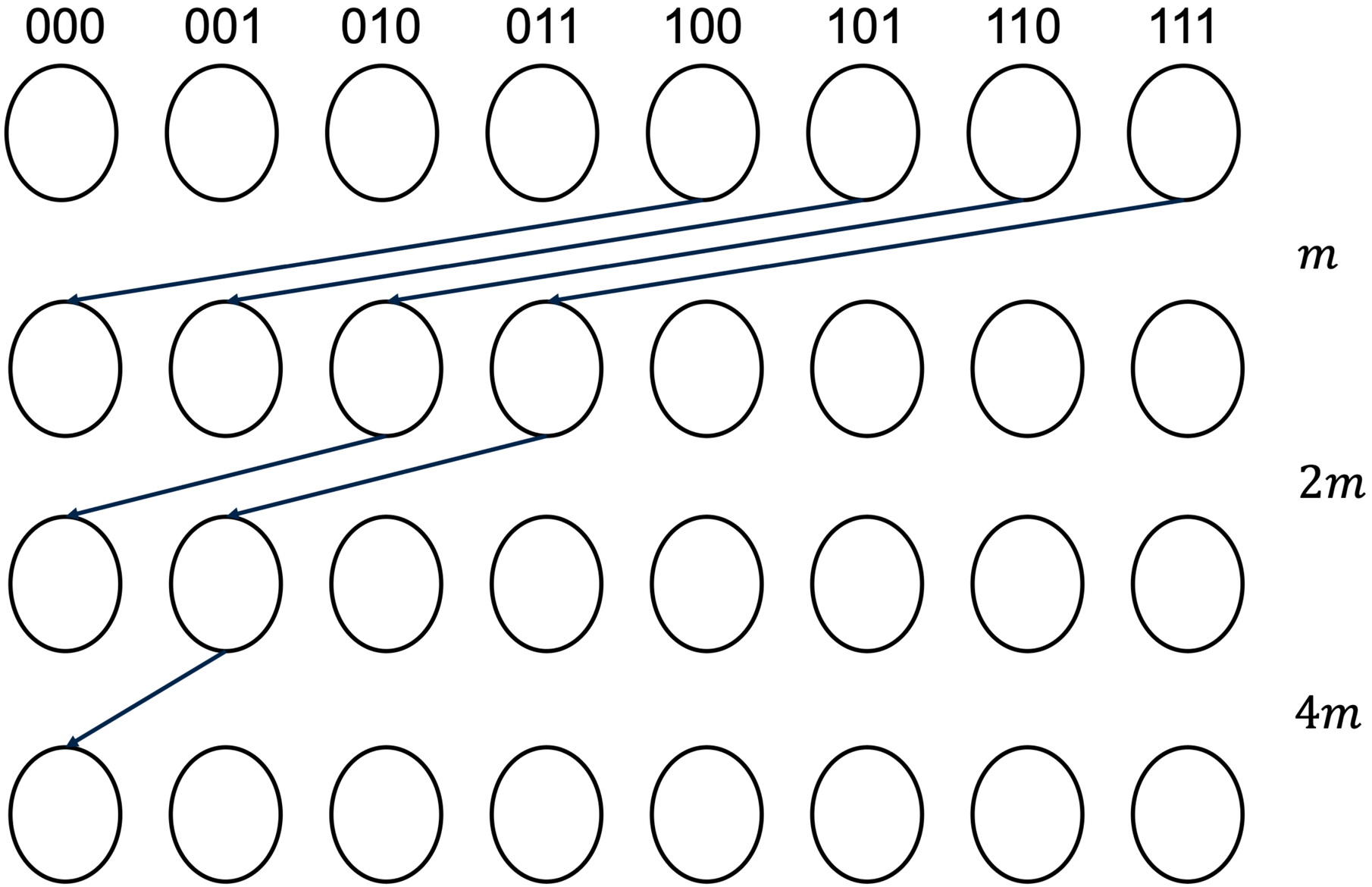
CSE 6220/CX 4220
Introduction to HPC

Lecture 8: Applications of Prefix Sums

Helen Xu
hxu615@gatech.edu

Georgia Tech College of Computing
School of Computational Science and Engineering

Super fast sidebar into Gather (for PA1)

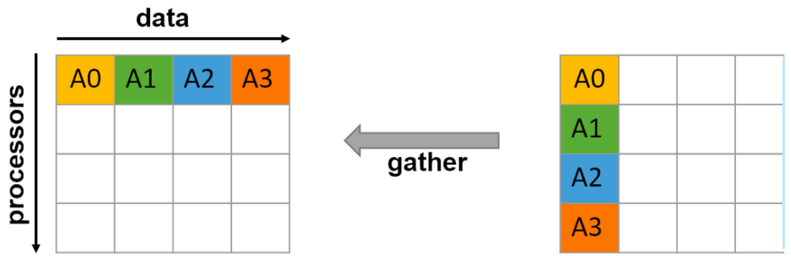


For message size 1, runtime is $(\tau + \mu)(\lg p)$

From Srinivas Aluru

Super fast sidebar into Gather (for PA1)

Gather data from all processes on one process(=root)




```
int MPI_Gather(void* sbuf, int scount, MPI_Datatype stype,
               void* rbuf, int rcount, MPI_Datatype rtype,
               int root, MPI_Comm comm);
```

```
#include <mpi.h>
#include <iostream >
#include <vector >
int main(int argc, char* argv[]) {
    int rank, size;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    std::vector<int> buf;
    if (rank == 0) buf.resize(size, -1);
    MPI_Gather(&rank  , 1, MPI_INT,
               &buf[0], 1, MPI_INT, 0, MPI_COMM_WORLD);
    if (rank == 0) std::cout << buf.back() << std::endl;
    return MPI_Finalize();
}
```

At this point, what is the content of the vector buf?

From Srinivas Aluru

Recap of Shared-Memory Parallel Prefix

Work-span examples

```
for(int i = 0; i < n; i++) {
  parallel_for(int j = 0; j < N; j++ {
    // O(1)
  }
}
```

What is the work and span?

Work-span examples

```
for(int i = 0; i < n; i++) {
  parallel_for(int j = 0; j < N; j++ {
    // O(1)
  }
}
```

What is the work and span?

Work = $\Theta(n^2)$

Span = $\Theta(n \lg n)$

Work-span examples

```
for(int i = 0; i < n; i++) {
  parallel_for(int j = 0; j < N; j++ {
    parallel_for(int k = 0; k < N; k++ {
      // O(1)
    }
  }
}
```

What is the work and span?

Work-span examples

```
for(int i = 0; i < n; i++) {
  parallel_for(int j = 0; j < N; j++ {
    parallel_for(int k = 0; k < N; k++ {
      // O(1)
    }
  }
}
```

What is the work and span?

Work = $\Theta(n^3)$

Span = $\Theta(n \cdot (\lg n + \lg n)) = \Theta(n \lg n)$

Work-span examples

```
for(int i = 0; i < n; i++) {
  parallel_for(int j = 0; j < N; j++ {
    for(int k = 0; k < N; k++ {
      // O(1)
    }
  }
}
```

What is the work and span?

Work-span examples

```
for(int i = 0; i < n; i++) {
  parallel_for(int j = 0; j < N; j++ {
    for(int k = 0; k < N; k++ {
      // O(1)
    }
  }
}
```

What is the work and span?

Work = $\Theta(n^3)$

Span = $\Theta(n(\lg n + n)) = \Theta(n^2)$

Hillis-Steele Prefix Sum

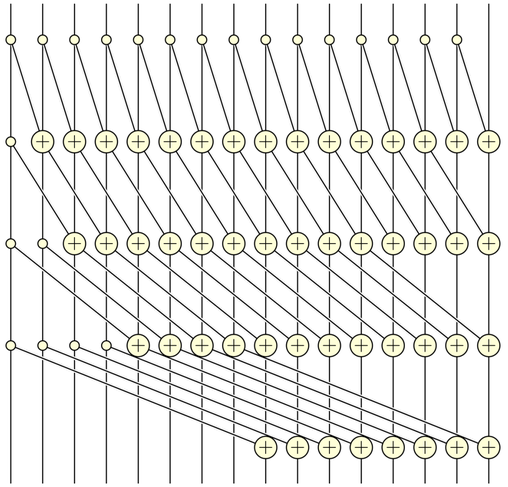
for i = 0 up to $\log(n)$:
  parallel_for j = 0 up to n-1:
    if j < 2^i:
      $x_j^{i+1} \leftarrow x_j^i$
    else:
      $x_j^{i+1} \leftarrow x_j^i + x_{j-2^i}^i$

What is the work and span?

Work = $O(n \lg n)$

Span = $O(\lg^2 n)$

Better, but still not work efficient



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

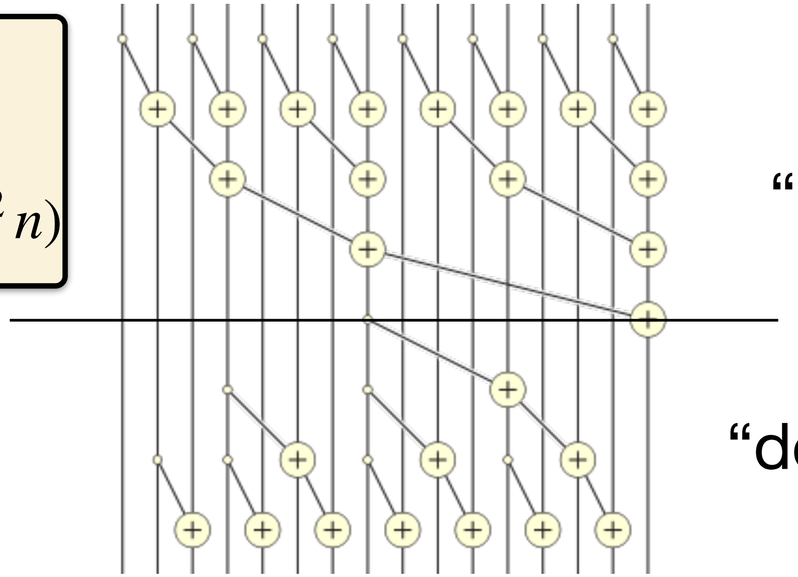
"Data-Parallel Algorithms," Hillis and Steele, CACM 1986.

Work-Efficient Parallel Prefix Pseudocode

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

Span in looping formulation follows the expression:
$\lg(n/2) + \lg(n/4) + \ldots + 1 = \Theta(\lg^2 n)$

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"

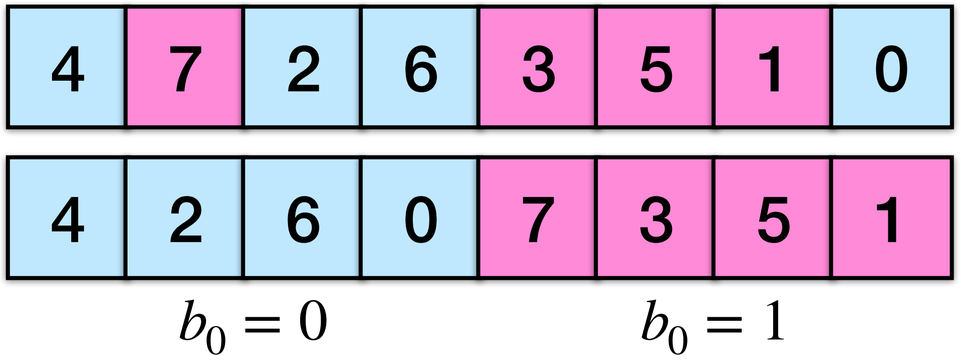
"downsweep"

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

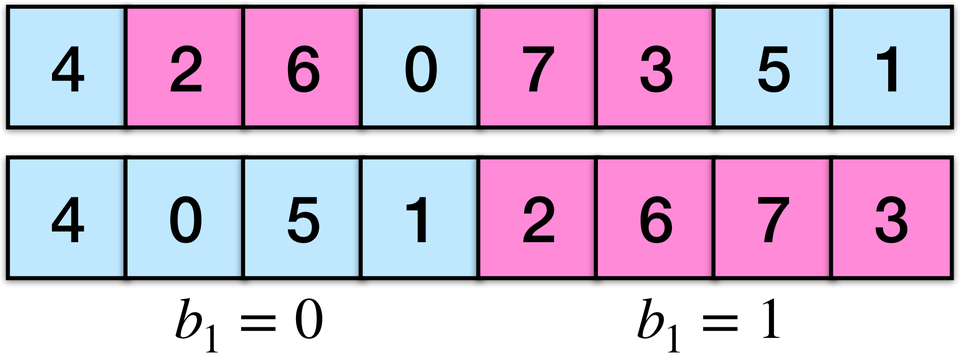
"Prefix Sums and Their Applications," Blelloch, 1990.

More Applications of Parallel Prefix

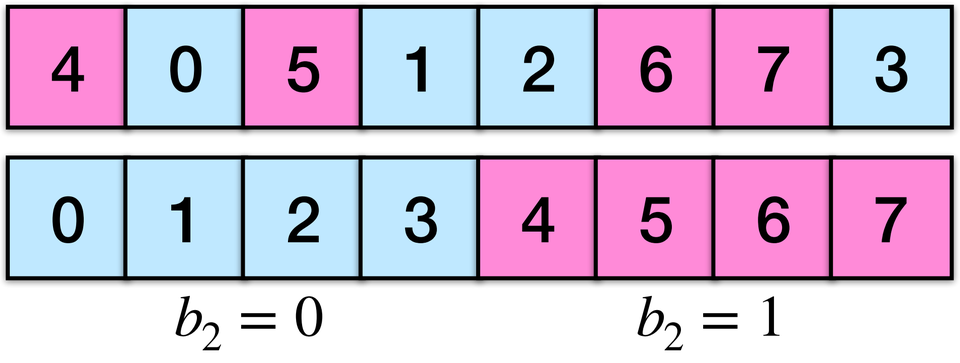
Application: Radix Sort (Serial)



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

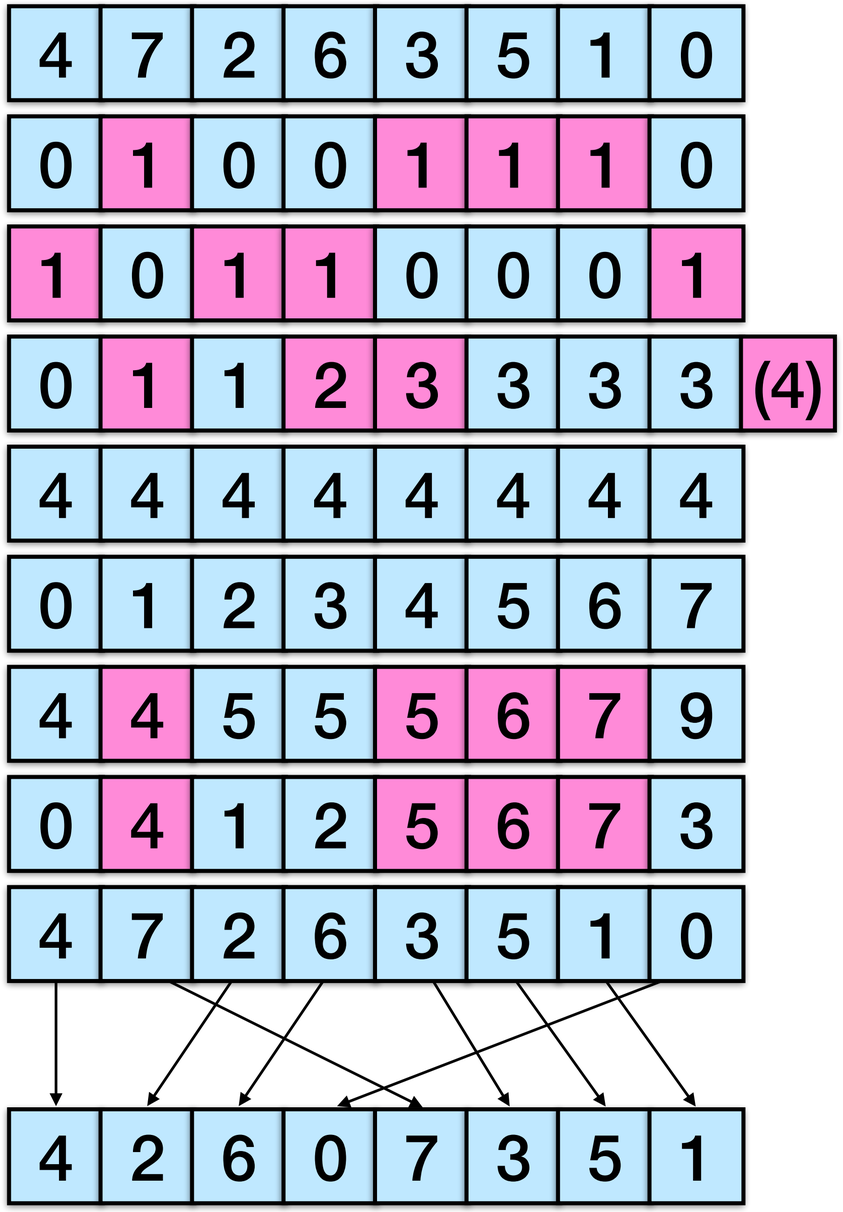


Stably sort entire array on next bit
X0X < X1X



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

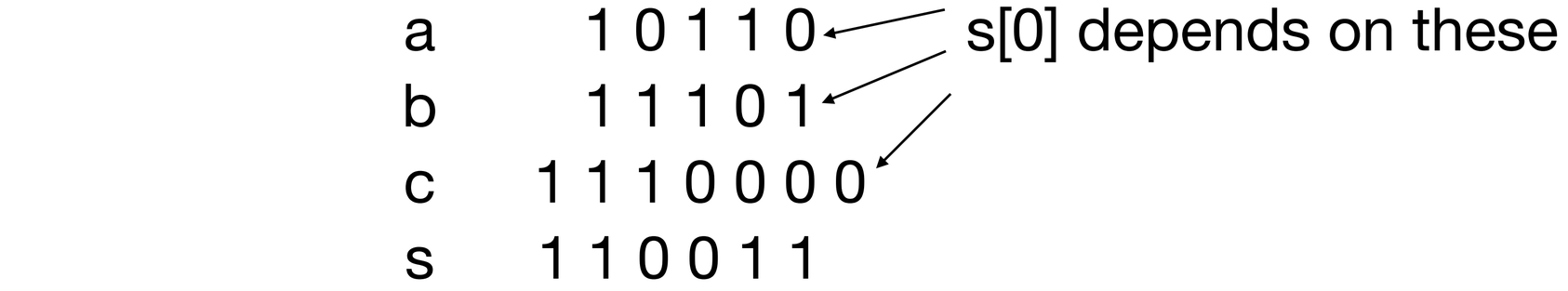
Problem: Computing sum of two n-bit binary numbers, a and b.

$a = a_{n-1}a_{n-2}\ldots a_0$ 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:

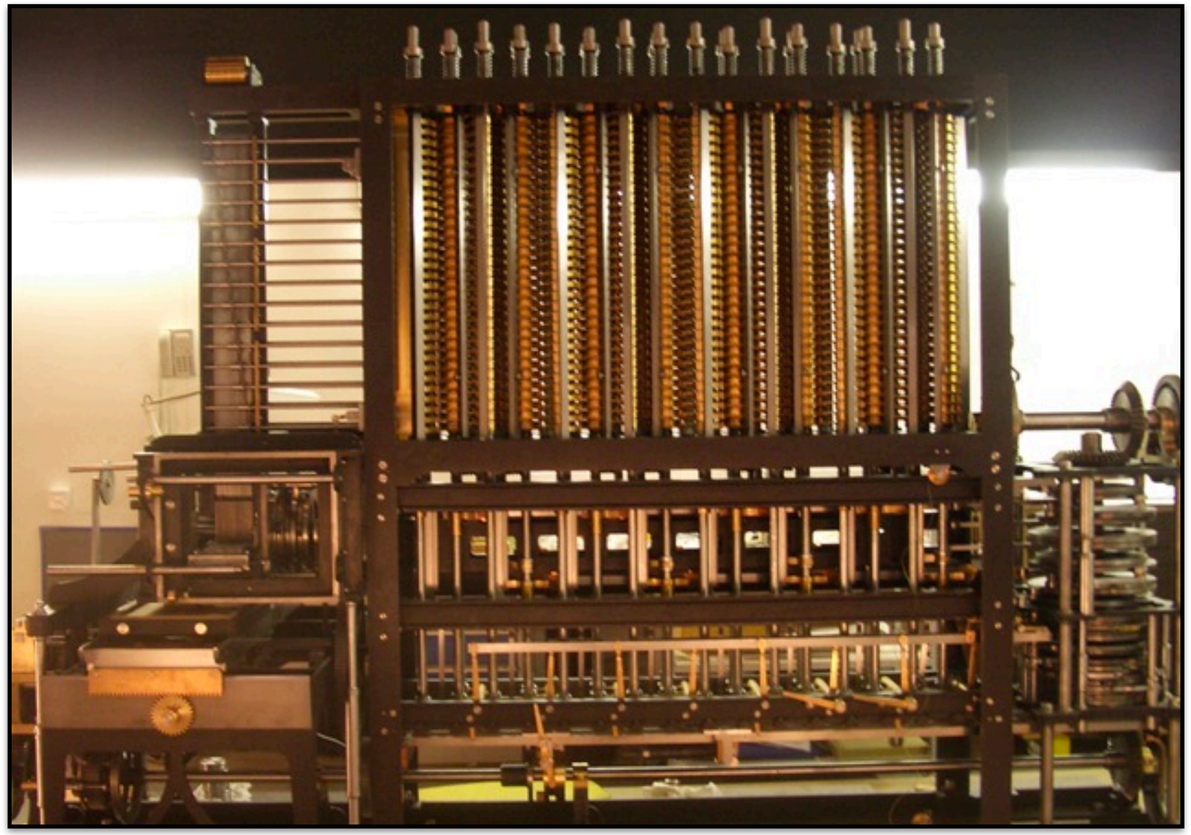
$$M[i] = \begin{pmatrix} p[i] & g[i] \\ 0 & 1 \end{pmatrix}$$

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

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.

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

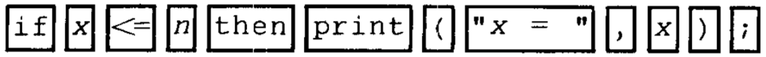


TABLE I. A Finite-State Automaton for Recognizing Tokens

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

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

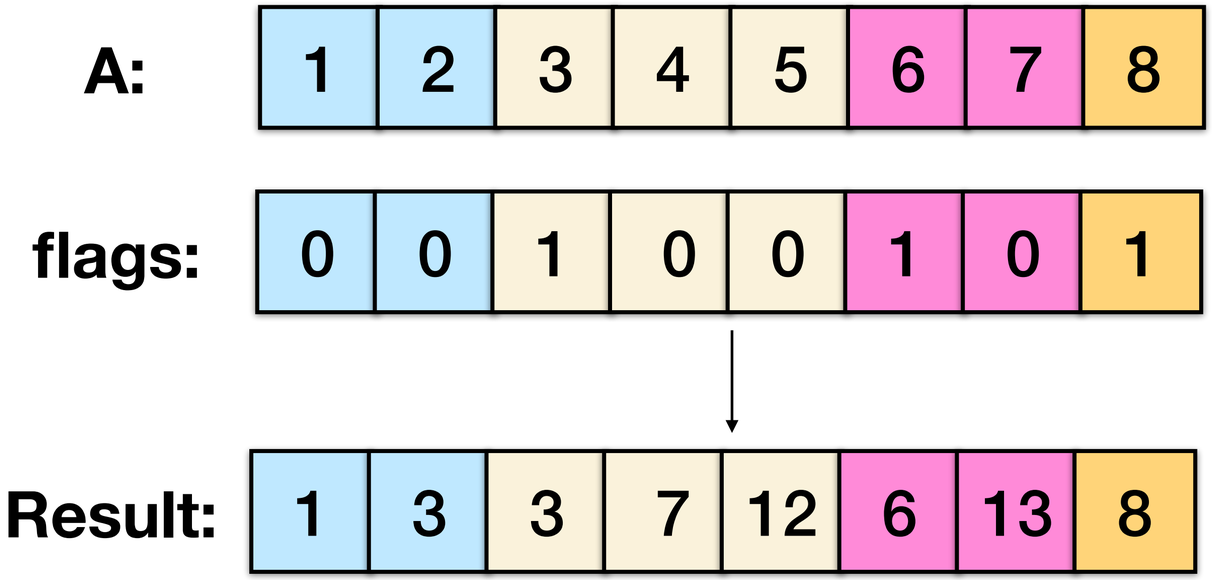
Old State	Character Read													
•	A	B	...	Y	Z	+	−	*	<	>	=	"	Space	New line
N	A	A	...	A	A	*	*	*	<	<	*	Q	N	N
A	Z	Z	...	Z	Z	*	*	*	<	<	*	Q	N	N
Z	Z	Z	...	Z	Z	*	*	*	<	<	*	Q	N	N
*	A	A	...	A	A	*	*	*	<	<	*	Q	N	N
<	A	A	...	A	A	*	*	*	<	<	=	Q	N	N
=	A	A	...	A	A	*	*	*	<	<	*	Q	N	N
Q	S	S	...	S	S	S	S	S	S	S	S	E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

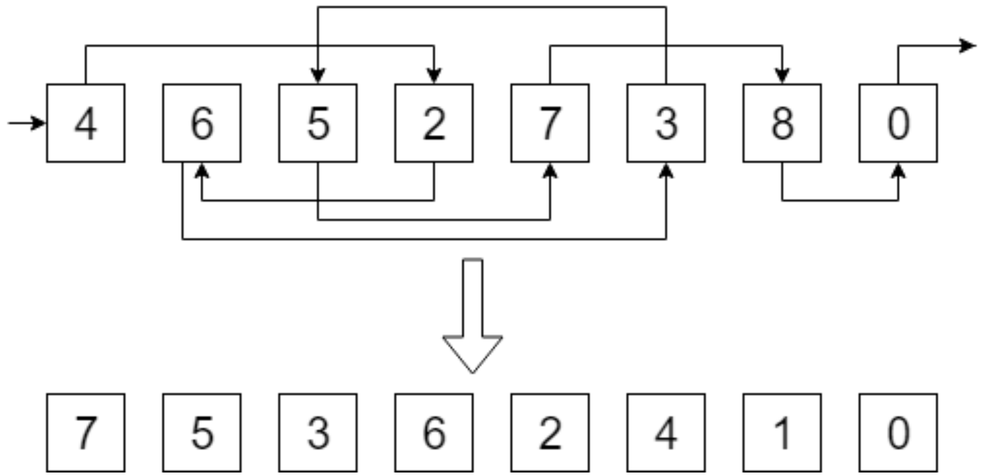
https://en.wikipedia.org/wiki/Summed-area_table

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.

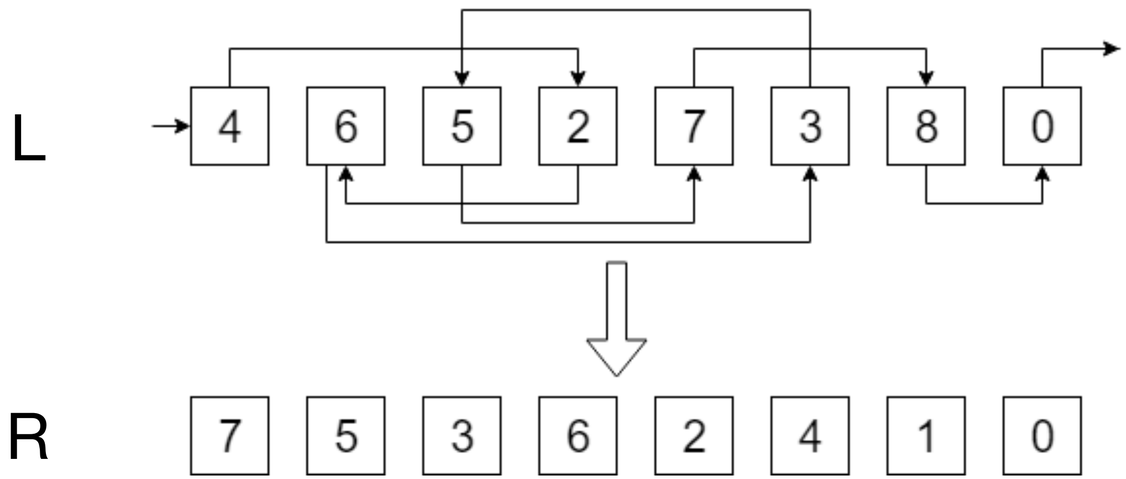


https://en.algorithmica.org/hpc/external-memory/list-ranking/

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).

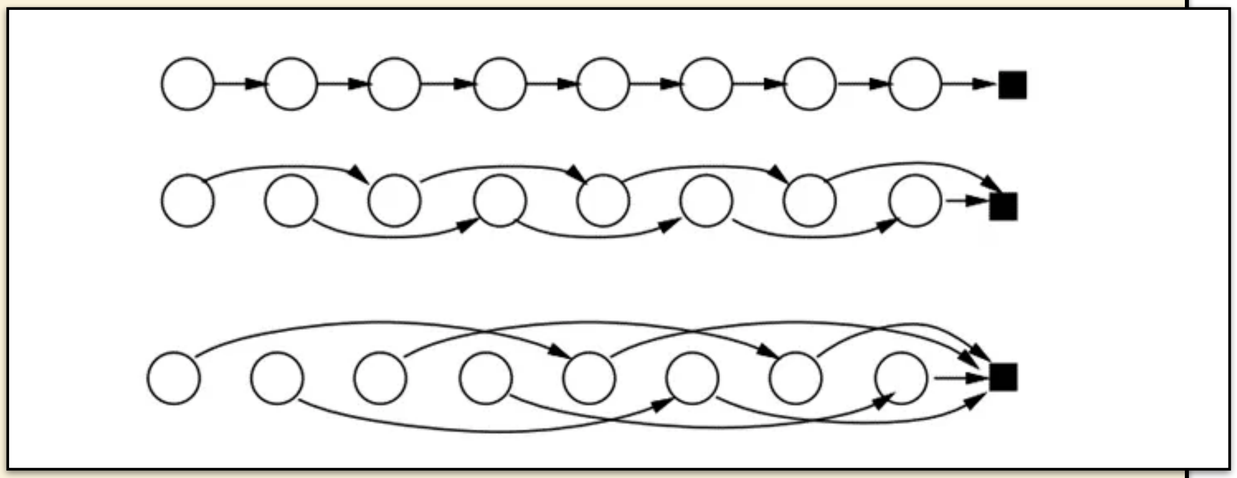
Output: A list R of length n containing the **rank** (distance from the end of the list) of each element.



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

Wyllie's Algorithm for List Ranking

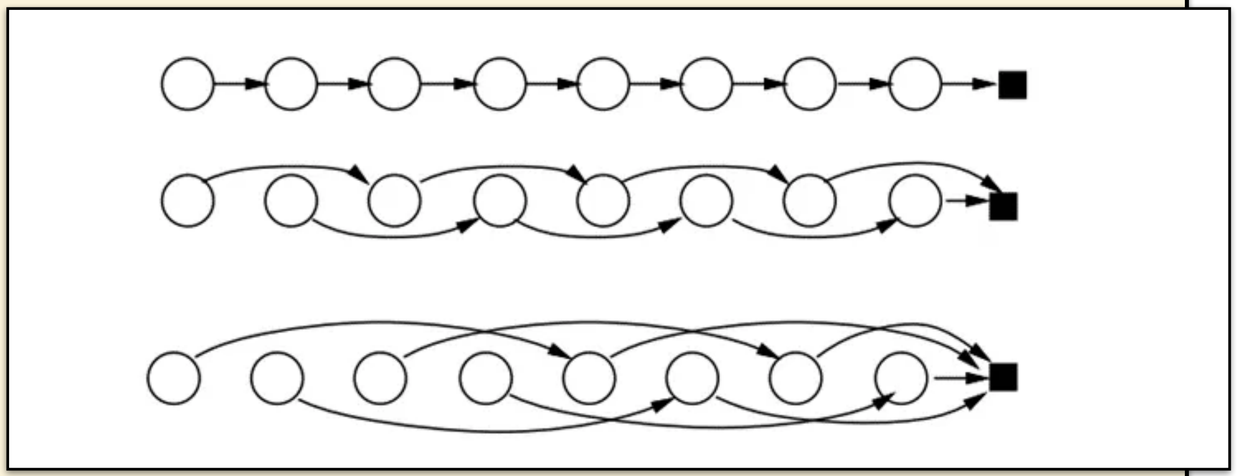
```
L = list of input successor pointers in an array
R = output list of ranks (distance from end)

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

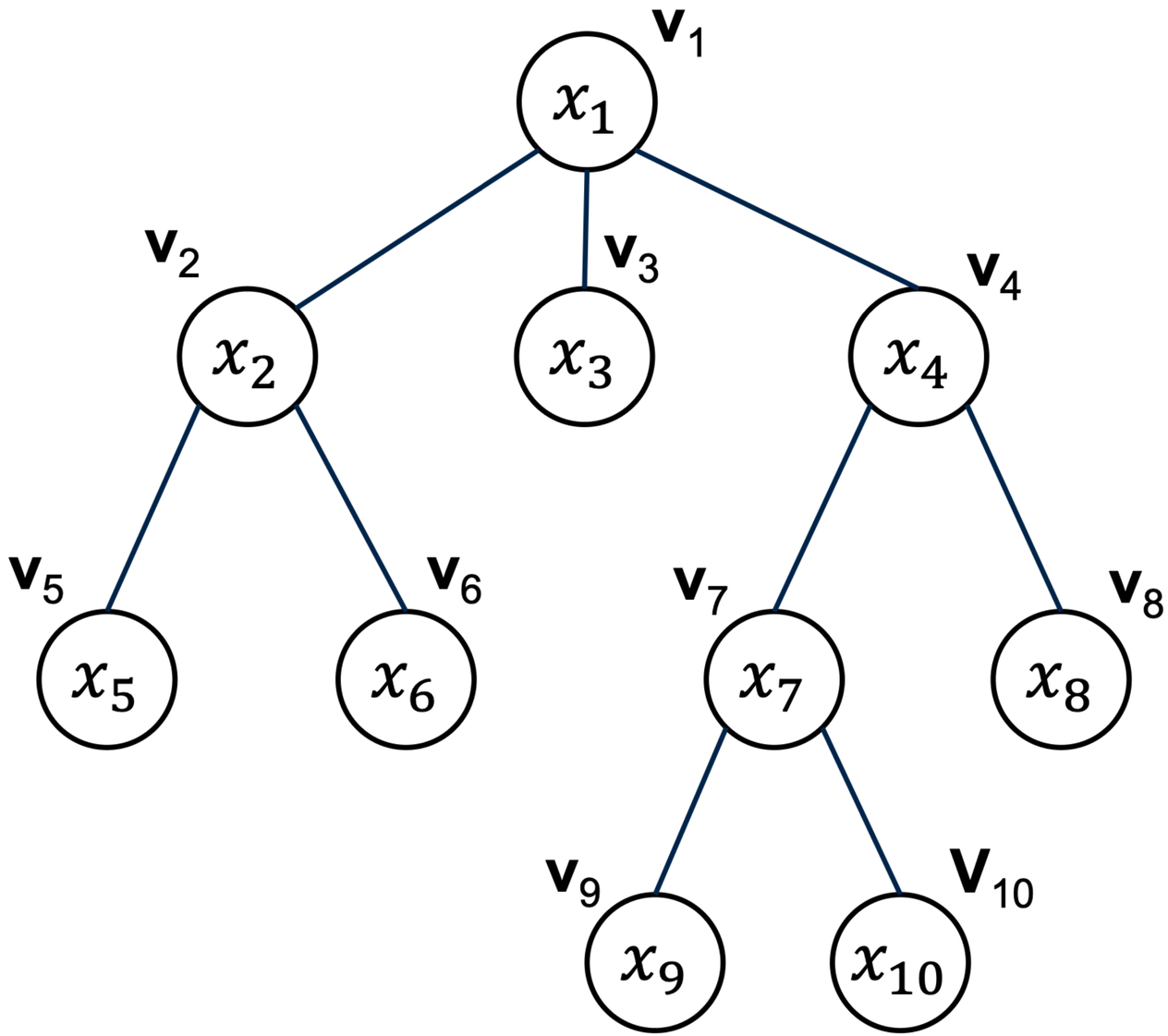
```
L = list of input successor pointers in an array
R = output list of ranks (distance from end)

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, since serial algorithm is O(n)

Work = $O(n \log n)$

Span = $O(\log^2 n)$

Application: Tree Accumulation



From Srinivas Aluru

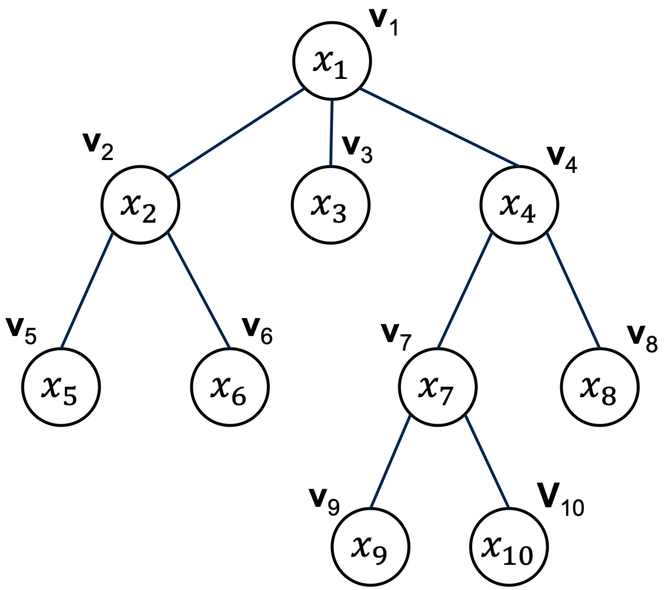
Application: Tree Accumulation

Upward Accumulation (UA): For a node v, sum all numbers in the subtree

- Example: $UA(v_4) = x_4 + x_7 + x_8 + x_9 + x_{10}$
- We want to compute it **for all nodes**

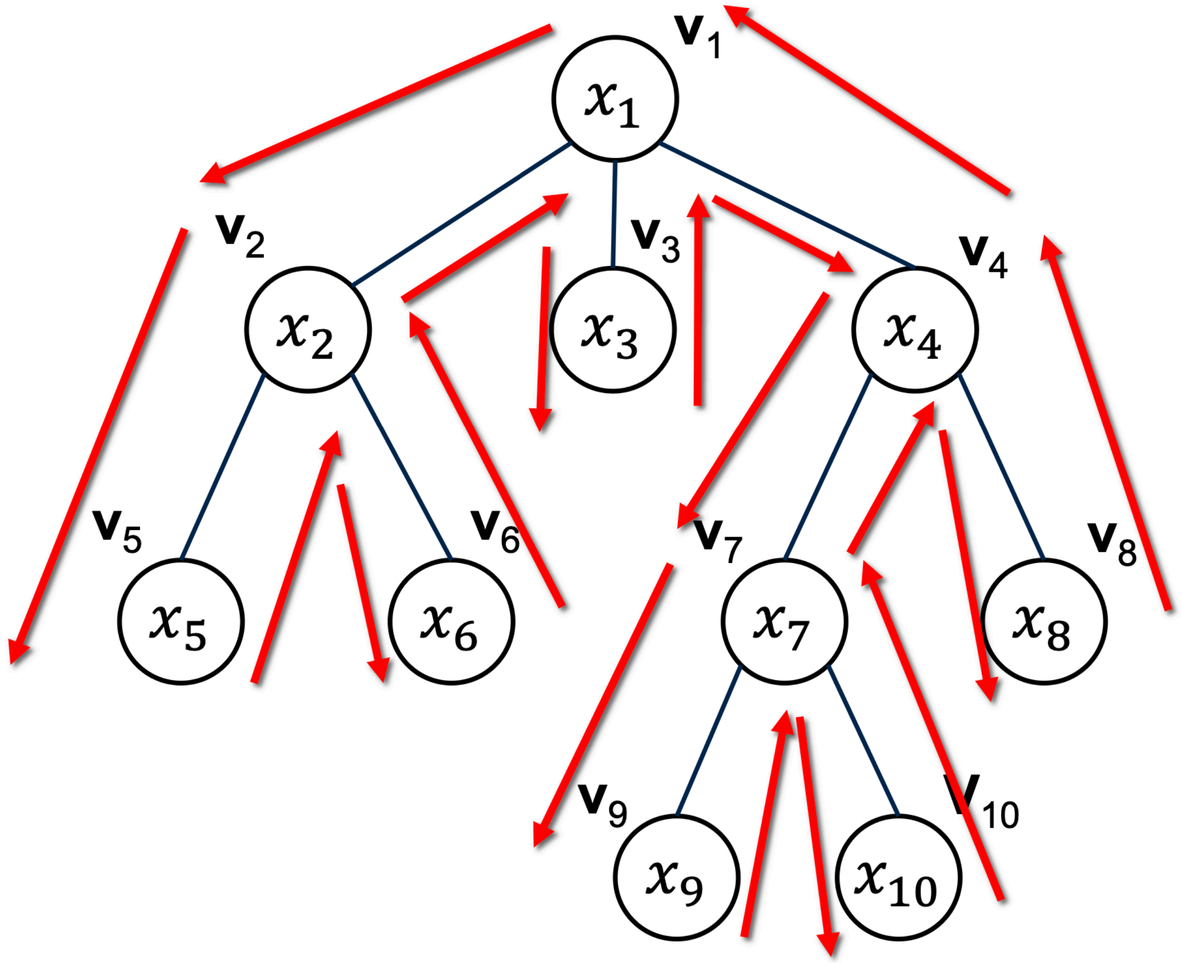
Downward Accumulation (DA): For each node v, sum all numbers in ancestors of v

- Example: $DA(v_7) = x_7 + x_4 + x_1$



Serial computation time is $\Theta(n)$ for both UA and DA

Euler Tour Example

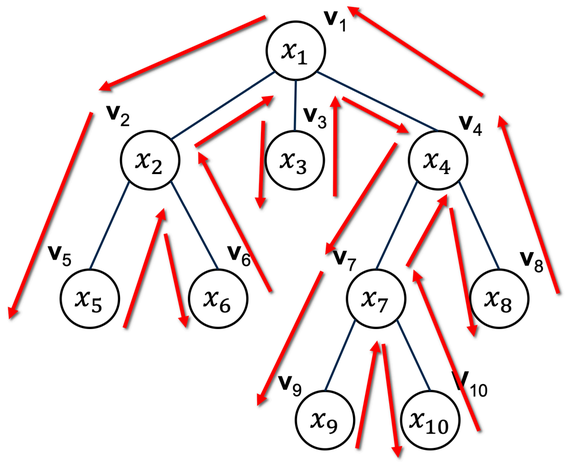


Euler Tour: $v_1\ v_2\ v_5\ v_2\ v_6\ v_2\ v_1\ v_3\ v_1\ v_4\ v_7\ v_9\ v_7\ v_{10}\ v_7\ v_4\ v_8\ v_4\ v_1$

From Srinivas Aluru

Tree Accumulation: UA

Sum in subtree



Euler Tour: $v_1\ v_2\ v_5\ v_2\ v_6\ v_2\ v_1\ v_3\ v_1\ v_4\ v_7\ v_9\ v_7\ v_{10}\ v_7\ v_4\ v_8\ v_4\ v_1$

- ET: $v_1\ v_2\ v_5\ v_2\ v_6\ v_2\ v_1\ v_3\ v_1\ v_4\ v_7\ v_9\ v_7\ v_{10}\ v_7\ v_4\ v_8\ v_4\ v_1$

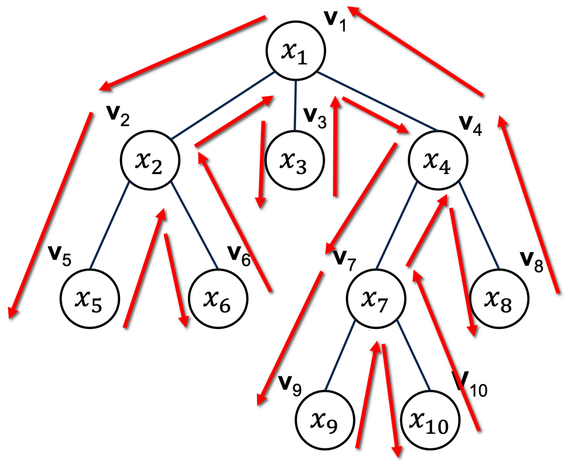
- $|ET| = 1 + 2(n-1) = 2n - 1$
- Assume ET is block partitioned among processors.
  - First occurrence of v_i put x_i
  - Otherwise put 0

v_1	v_2	v_5	v_2	v_6	v_2	v_1	v_3	v_1	v_4	v_7	v_9	v_7	v_{10}	v_7	v_4	v_8	v_4	v_1
x_1	x_2	x_5	0	x_6	0	0	x_3	0	x_4	x_7	x_9	0	x_{10}	0	0	x_8	0	0

- PS: resulting prefix sums array
- i_f: index of first occurrence of v
- i_l: index of last occurrence of v
- UA(v) = PS[i_l] - PS[i_f-1]

Tree Accumulation: UA

Sum in subtree



Euler Tour: $v_1\ v_2\ v_5\ v_2\ v_6\ v_2\ v_1\ v_3\ v_1\ v_4\ v_7\ v_9\ v_7\ v_{10}\ v_7\ v_4\ v_8\ v_4\ v_1$

- ET: $v_1\ v_2\ v_5\ v_2\ v_6\ v_2\ v_1\ v_3\ v_1\ v_4\ v_7\ v_9\ v_7\ v_{10}\ v_7\ v_4\ v_8\ v_4\ v_1$
- $|ET| = 1 + 2(n-1) = 2n - 1$
- Assume ET is block partitioned among processors.
  - First occurrence of v_i put x_i
  - Otherwise put 0

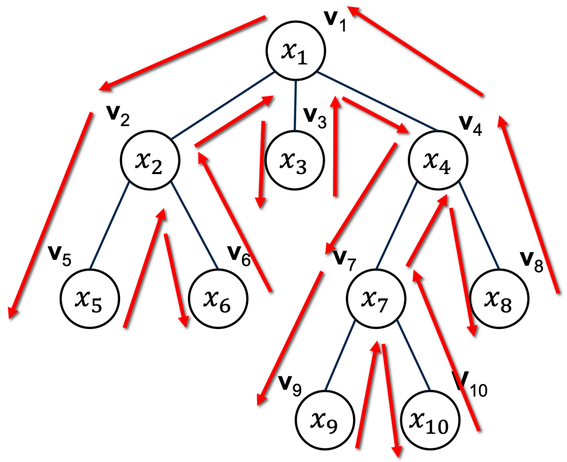
v_1	v_2	v_5	v_2	v_6	v_2	v_1	v_3	v_1	v_4	v_7	v_9	v_7	v_{10}	v_7	v_4	v_8	v_4	v_1
x_1	x_2	x_5	0	x_6	0	0	x_3	0	x_4	x_7	x_9		x_{10}	0	0	x_8	0	0

- PS: resulting prefix sums array
- i_f: index of first occurrence of v
- i_l: index of last occurrence of v
- UA(v) = PS[i_l] - PS[i_f-1]

How do we tell if an entry is the first occurrence? We need to find it in parallel and each one in O(1)

Tree Accumulation: UA

Sum in subtree



Euler Tour: $v_1\ v_2\ v_5\ v_2\ v_6\ v_2\ v_1\ v_3\ v_1\ v_4\ v_7\ v_9\ v_7\ v_{10}\ v_7\ v_4\ v_8\ v_4\ v_1$

- ET: $v_1\ v_2\ v_5\ v_2\ v_6\ v_2\ v_1\ v_3\ v_1\ v_4\ v_7\ v_9\ v_7\ v_{10}\ v_7\ v_4\ v_8\ v_4\ v_1$
- $|ET| = 1 + 2(n-1) = 2n - 1$
- Assume ET is block partitioned among processors.
  - First occurrence of v_i put x_i
  - Otherwise put 0

v_1	v_2	v_5	v_2	v_6	v_2	v_1	v_3	v_1	v_4	v_7	v_9	v_7	v_{10}	v_7	v_4	v_8	v_4	v_1
x_1	x_2	x_5	0	x_6	0	0	x_3	0	x_4	x_7	x_9		x_{10}	0	0	x_8	0	0

- PS: resulting prefix sums array
- i_f: index of first occurrence of v
- i_l: index of last occurrence of v
- UA(v) = PS[i_l] - PS[i_f-1]

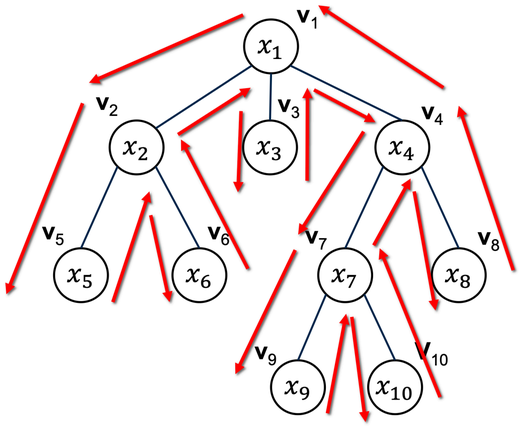
How do we tell if an entry is the first occurrence? We need to find it in parallel and each one in O(1)

Look to your left - if it is the parent, you are the first occurrence

From Srinivas Aluru

Tree Accumulation: DA

Sum up self + ancestors



Euler Tour: $v_1\ v_2\ v_5\ v_2\ v_6\ v_2\ v_1\ v_3\ v_1\ v_4\ v_7\ v_9\ v_7\ v_{10}\ v_7\ v_4\ v_8\ v_4\ v_1$

- ET: $v_1\ v_2\ v_5\ v_2\ v_6\ v_2\ v_1\ v_3\ v_1\ v_4\ v_7\ v_9\ v_7\ v_{10}\ v_7\ v_4\ v_8\ v_4\ v_1$
- Assume ET is block partitioned among processors.
  - First occurrence of v_i put x_i
  - Last occurrence of v_i put $-x_i$
  - Otherwise put 0

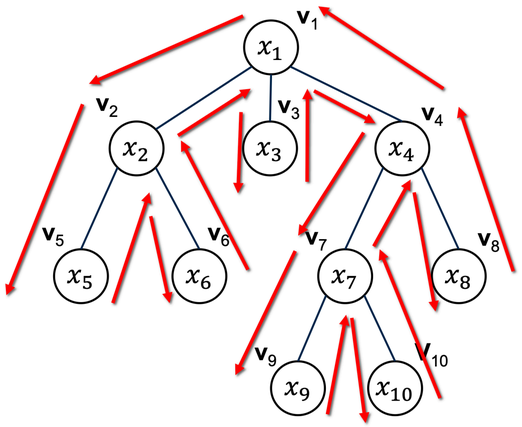
v_1	v_2	v_5	v_2	v_6	v_2	v_1	v_3	v_1	v_4	v_7	v_9	v_7	v_{10}	v_7	v_4	v_8	v_4	v_1
x_1	x_2	x_5	0	x_6	$-x_2$	0	x_3	0	x_4		x_9	0	x_{10}	$-x_7$	0	x_8	$-x_4$	$-x_1$

- PS: resulting prefix sums array
- i_f: index of first occurrence of v
- DA(v) = PS[i_f]

How do we know which is the last occurrence? We need to find it in parallel and each one in O(1)

Tree Accumulation: DA

Sum up self + ancestors



Euler Tour: $v_1\ v_2\ v_5\ v_2\ v_6\ v_2\ v_1\ v_3\ v_1\ v_4\ v_7\ v_9\ v_7\ v_{10}\ v_7\ v_4\ v_8\ v_4\ v_1$

- ET: $v_1\ v_2\ v_5\ v_2\ v_6\ v_2\ v_1\ v_3\ v_1\ v_4\ v_7\ v_9\ v_7\ v_{10}\ v_7\ v_4\ v_8\ v_4\ v_1$
- Assume ET is block partitioned among processors.
  - First occurrence of v_i put x_i
  - Last occurrence of v_i put $-x_i$
  - Otherwise put 0

v_1	v_2	v_5	v_2	v_6	v_2	v_1	v_3	v_1	v_4	v_7	v_9	v_7	v_{10}	v_7	v_4	v_8	v_4	v_1
x_1	x_2	x_5	0	x_6	$-x_2$	0	x_3	0	x_4		x_9	0	x_{10}	$-x_7$	0	x_8	$-x_4$	$-x_1$

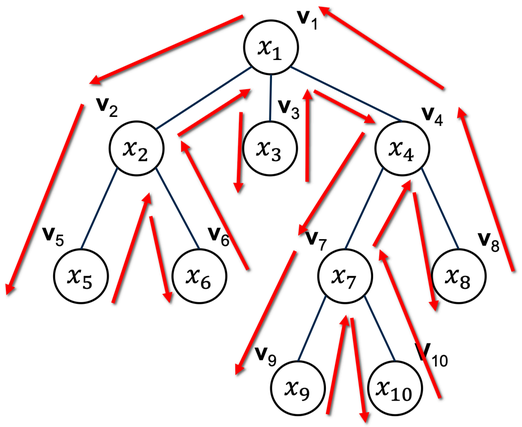
- PS: resulting prefix sums array
- i_f: index of first occurrence of v
- DA(v) = PS[i_f]

How do we know which is the last occurrence? We need to find it in parallel and each one in O(1)

Look to your right - if it is the parent, you are the last occurrence

Tree Accumulation: DA

Sum up self + ancestors



Euler Tour: $v_1\ v_2\ v_5\ v_2\ v_6\ v_2\ v_1\ v_3\ v_1\ v_4\ v_7\ v_9\ v_7\ v_{10}\ v_7\ v_4\ v_8\ v_4\ v_1$

- ET: $v_1\ v_2\ v_5\ v_2\ v_6\ v_2\ v_1\ v_3\ v_1\ v_4\ v_7\ v_9\ v_7\ v_{10}\ v_7\ v_4\ v_8\ v_4\ v_1$
- Assume ET is block partitioned among processors.
  - First occurrence of v_i put x_i
  - Last occurrence of v_i put $-x_i$
  - Otherwise put 0

v_1	v_2	v_5	v_2	v_6	v_2	v_1	v_3	v_1	v_4	v_7	v_9	v_7	v_{10}	v_7	v_4	v_8	v_4	v_1
x_1	x_2	x_5	0	x_6	$-x_2$	0	x_3	0	x_4	x_7	x_9	0	x_{10}	$-x_7$	0	x_8	$-x_4$	$-x_1$

- PS: resulting prefix sums array
- i_f: index of first occurrence of v
- DA(v) = PS[i_f]

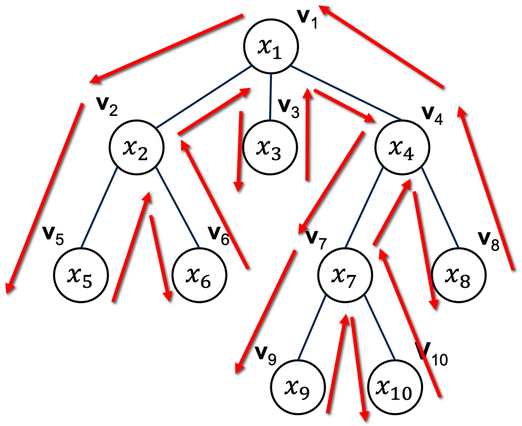
How do we know which is the last occurrence? We need to find it in parallel and each one in O(1)

Is this correct? Hint: Look at the leaves

Look to your right - if it is the parent, you are the last occurrence

Tree Accumulation: DA

Sum up self + ancestors



Euler Tour: $v_1\ v_2\ v_5\ v_2\ v_6\ v_2\ v_1\ v_3\ v_1\ v_4\ v_7\ v_9\ v_7\ v_{10}\ v_7\ v_4\ v_8\ v_4\ v_1$

- ET: $v_1\ v_2\ v_5\ v_2\ v_6\ v_2\ v_1\ v_3\ v_1\ v_4\ v_7\ v_9\ v_7\ v_{10}\ v_7\ v_4\ v_8\ v_4\ v_1$
- Assume ET is block partitioned among processors.
  - First occurrence of v_i put x_i
  - Last occurrence of v_i put $-x_i$
  - Otherwise put 0

v_1	v_2	v_5	v_2	v_6	v_2	v_1	v_3	v_1	v_4	v_7	v_9	v_7	v_{10}	v_7	v_4	v_8	v_4	v_1
x_1	x_2	x_5	0	x_6	$-x_2$	0	x_3	0	x_4		x_9	0	x_{10}	$-x_7$	0	x_8	$-x_4$	$-x_1$

How do we know which is the last occurrence? We need to find it in parallel and each one in O(1)

- PS: resulting prefix sums array
- i_f: index of first occurrence of v
- DA(v) = PS[i_f]

Is this correct? Hint: Check the leaves

Look to your right - if it is the parent, you are the last occurrence

From Srinivas Aluru

Tree Accumulation: DA

Sum up self + ancestors

- ET: $v_1\ v_2\ v_5\ v_2\ v_6\ v_2\ v_1\ v_3\ v_1\ v_4\ v_7\ v_9\ v_7\ v_{10}\ v_7\ v_4\ v_8\ v_4\ v_1$
- Assume ET is block partitioned among processors.
  - Internal nodes:
    - First occurrence of v_i put x_i
    - Last occurrence of v_i put $-x_i$
    - Otherwise put 0
  - Leaf nodes: put 0

v_1	v_2	v_5	v_2	v_6	v_2	v_1	v_3	v_1	v_4	v_7	v_9	v_7	v_{10}	v_7	v_4	v_8	v_4	v_1
x_1	x_2	0	0	0	$-x_2$	0	0	0	x_4	x_7	0	0	0	$-x_7$	0	0	$-x_4$	$-x_1$

- PS: resulting prefix sums array
- i_f: index of first occurrence of v
- If v_i is an internal node
  - DA(v_i) = PS[i_f]
- If v_i is a leaf node
  - DA(v_i) = PS[i_f] + x_i

From Srinivas Aluru

Sequence Alignment

Over all possible alignments

- An important problem in computational biology.
- DNA sequences: Strings over {A,C,G,T}.
- Goal: To find out how “evolutionarily close” the sequences are. This is tested by how “well” the sequences align.
- Alignment: Arranging characters of each sequence into columns to expose similarity.
- Gaps (-) may be inserted to denote insertions/deletions.
- Optimal Alignment: Minimum number of substitutions, insertions, and deletions needed to turn one sequence into another.

From Srinivas Aluru

Example Alignment and Score Computation

- Alignment has a score that shows quality.
- Every column of an alignment is a match, mismatch or a gap.
- Matches are preferred and hence have a positive score, others have a negative score.
- Example: If match = 1, mismatch = 0 and gap = −1, then for the following alignment

A	T	G	A	–	C	C
A	–	G	A	A	T	C
1	-1	1	1	-1	0	1

Score(ATGACC, AGAATC) = 2

Problem Definition

Input:

- Two sequences A=$a_1, a_2, \cdots, a_m$ and B= $b_1, b_2, \cdots, b_n$
- Scores for match/mismatch ($f(a, b)$) and gap (g)

Algorithm:

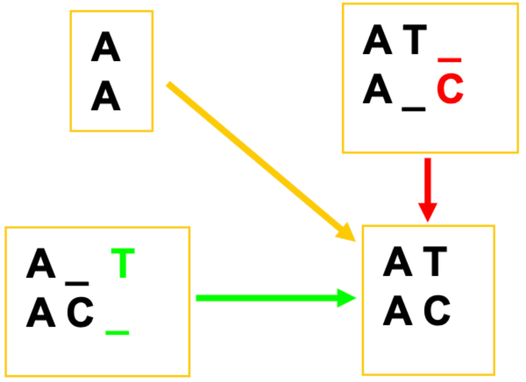
Dynamic programing solution:

- T = Table of size $(m+1) \times (n+1)$
- $T[i, j]$ = best score between $a_1, a_2, \cdots, a_i$ and $b_1, b_2, \cdots, b_j$
- $$T[i,j] = \max \begin{cases} T[i-1, j-1] + f(a_i, b_j) & \text{match/mismatch} \\ T[i-1, j] - g & \text{gap in B} \\ T[i, j-1] - g & \text{gap in A} \end{cases}$$
- Sequential time =$O(mn)$

Dynamic Programming Matrix

$$T[i,j] = \max \begin{cases} T[i-1,j-1] + f(a_i, b_j) \\ T[i-1,j] - g \\ T[i,j-1] - g \end{cases}$$

Match = 2
Substitution = 0
Gap = -1



	0	1	2	3	4	5	6
0		Gap	A	T	G	C	T
1	Gap	0	-1	-2	-3	-4	-5
2	A	-1	2 (2)	1 (0)	0	0	0
3	C	-2	1 (0)	2 (0)	1	2	1
4	C	-3	0	1 (0)	2	3	2
5	G	-4	0	0	3 (2)	2	3
6	C	-5	0	0	2	5 (2)	4
7	T	-6	0	2	1	4	7 (2)

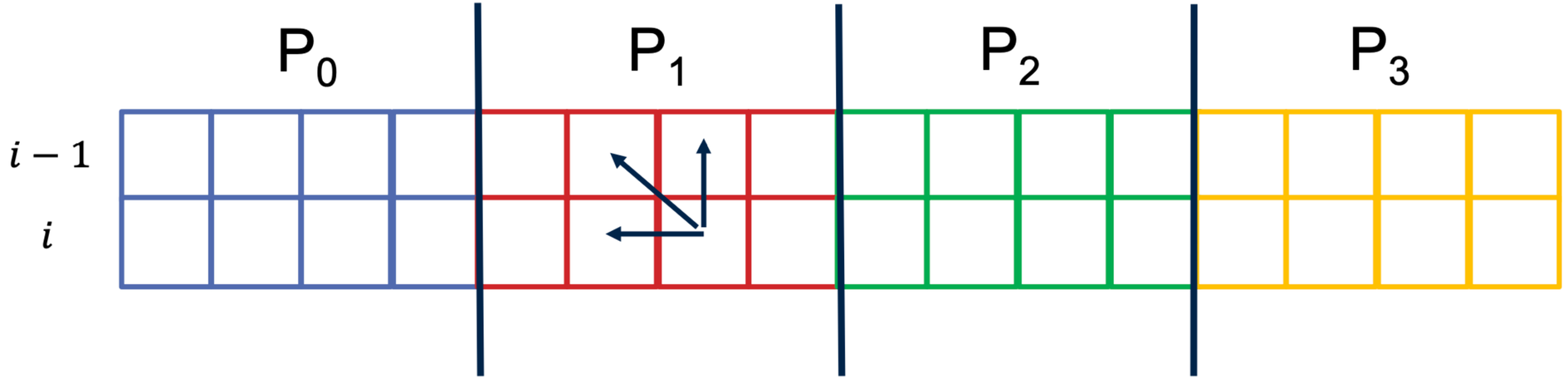
Initialize all gaps

Max score

From Srinivas Aluru

Solution Using Parallel Prefix

- We compute each row of T in parallel:



- $w[j] = \max \begin{cases} T[i-1, j-1] + f(a_i, b_j) \\ T[i-1, j] - g \end{cases}$

Can be computed in parallel upfront with previous row

- $T[i,j] = \max \begin{cases} w[j] \\ T[i, j-1] - g \end{cases}$

max is a binary associative operator, but -g poses issue for immediate use of parallel prefix

From Srinivas Aluru

Solution Using Parallel Prefix

- $x[j] = T[i,j] + j \cdot g$

j is a global index

- $x[j] = \max \begin{cases} w[j] + jg \\ T[i, j-1] - g + jg \end{cases}$
- $x[j] = \max \begin{cases} w[j] + jg \\ T[i, j-1] + (j-1)g \end{cases}$
- $x[j] = \max \begin{cases} w[j] + jg \\ x[j-1] \end{cases}$

compute x[j] using parallel prefix, then get T[i, j]

From Srinivas Aluru

Parallel Algorithm: Compute row i from i-1

1. Use right shift to send last element of $(i-1)^{th}$ row on each processor. $O(\tau + \mu)$
2. Create vector A

$$A[j] = jg + \max\begin{cases} T[i-1, j-1] + f(a_i, b_j) \\ T[i-1, j] - g \end{cases} \qquad O\left(\frac{n}{p}\right)$$

3. Compute parallel prefix on A using max as operator and store results in x $O\left(\frac{n}{p} + (\tau + \mu)\, log\, p\right)$
4. Compute $T[i,j]$'s using $O\left(\frac{n}{p}\right)$

$$T[i,j] = x[j] - jg$$

Computation Time = $O\left(\frac{mn}{p} + m \log p\right)$

Communication Time = $O\left((\tau + \mu) m\, log\, p\right)$

Efficient when $p = O\left(\frac{n}{\log n}\right)$

Needs to be done for each row

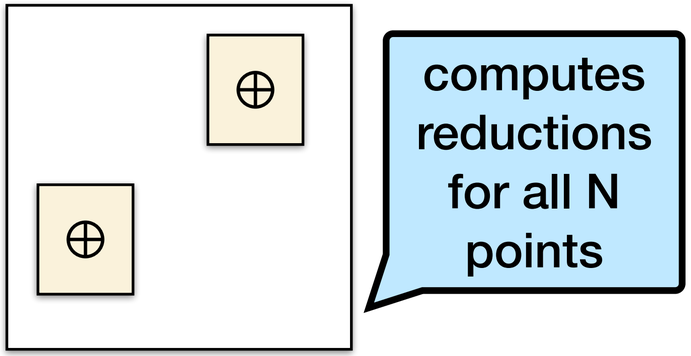
Application: Tensor Region Sums

Several scientific computing applications involve reducing **many (potentially overlapping) regions** of a tensor to a single value for each region, using a binary associative operator $\oplus$.

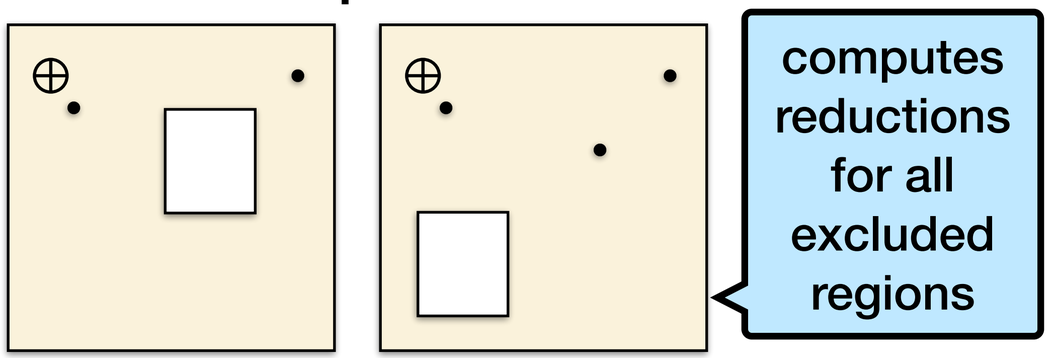
Inclusion: The **summed-area table** (SAT) method preprocesses an image to answer queries about the sum in rectangular subregions of a tensor [C84].

Exclusion: The essence of the **fast multipole method** (FMM) is a reduction of a subregion's elements, excluding elements too close [BG97, CRW93, D00].

Summed-area Table



Fast Multipole Method



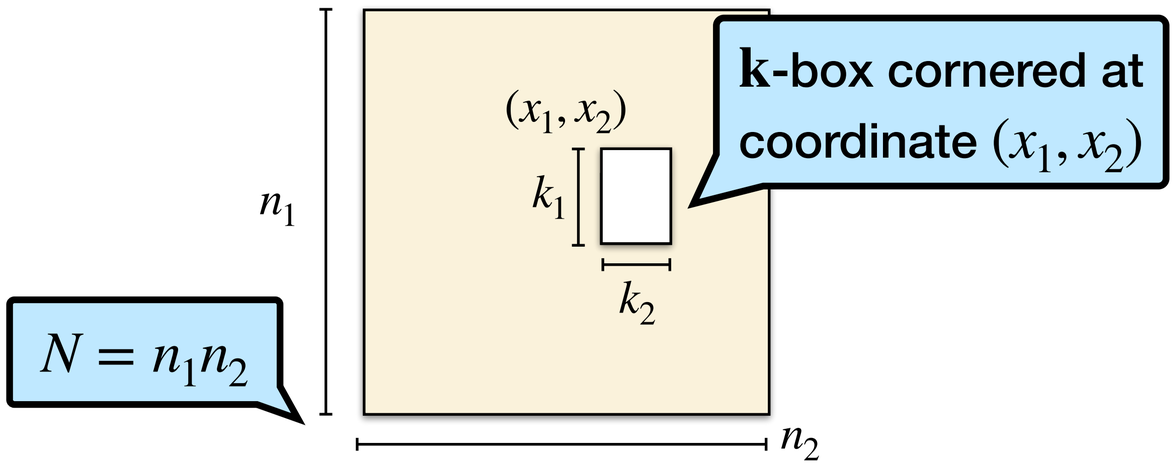
Binary associative operator

Excluded-sums problem

The excluded-sums problem [DDELP05] underlies applications that require **reducing regions of a tensor** to a single value using $\oplus$.

In 2D, it takes as input an $n_1 \times n_2$ matrix A and "box size" $\mathbf{k} = (k_1, k_2)$ where $k_1 \leq n_1, k_2 \leq n_2$.

The problem involves reducing the excluded region outside of **every** $\mathbf{k}$-box in the matrix.



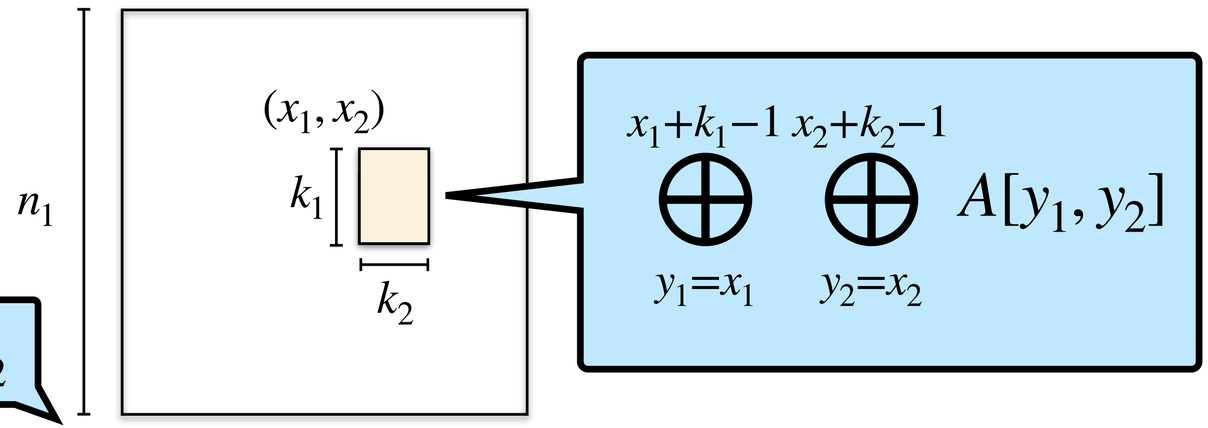
Binary associative operator

Included-sums Problem

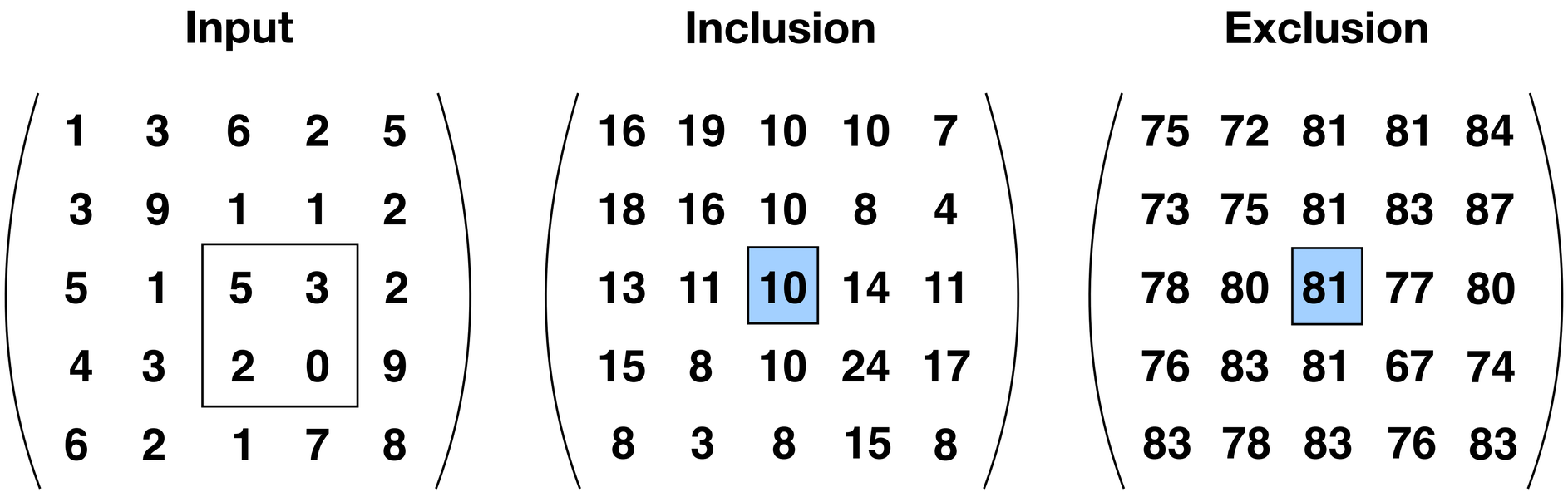
The included-sums problem takes the same input as the excluded-sums problem.

In 2D, the included sum at coordinate (x_1, x_2) involves reducing (accumulating with $\oplus$) all elements **in the k-box** cornered at (x_1, x_2).

Can be computed straightforwardly with four nested loops in $\Theta(n_1 n_2 k_1 k_2)$ time.

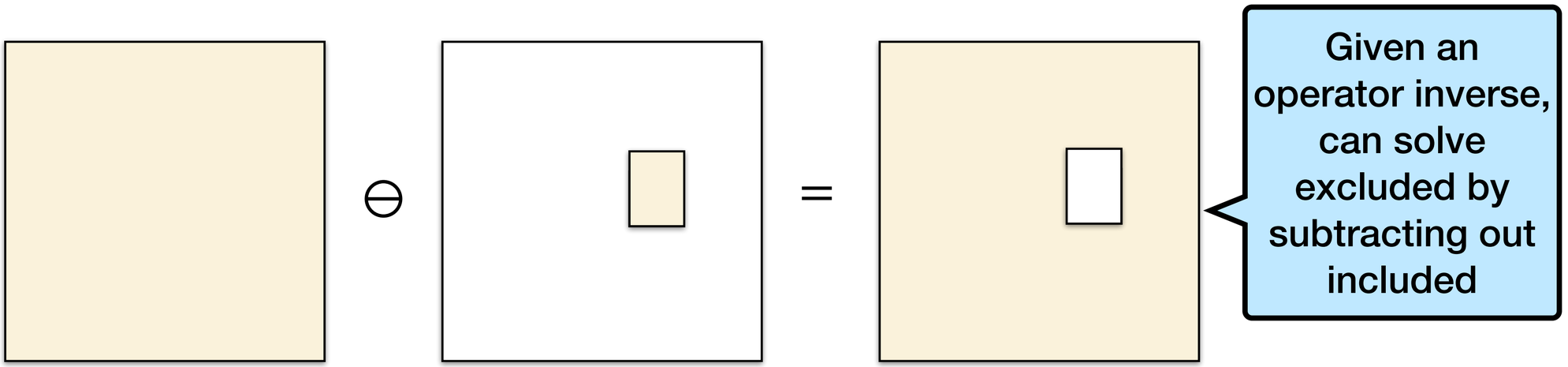


Inclusion and Exclusion Example



We present an example with addition for ease of understanding, but in general an algorithm for these problems should work with general operators.

Included and Excluded Sums With and Without Operator Inverse



This approach fails for **operators without inverse** such as max, or the FMM's functions, which may exhibit singularities [DDELP05].

We refine the included- and excluded-sums problems into **weak** and **strong** versions. The weak version requires an operator inverse, while the strong version does not.

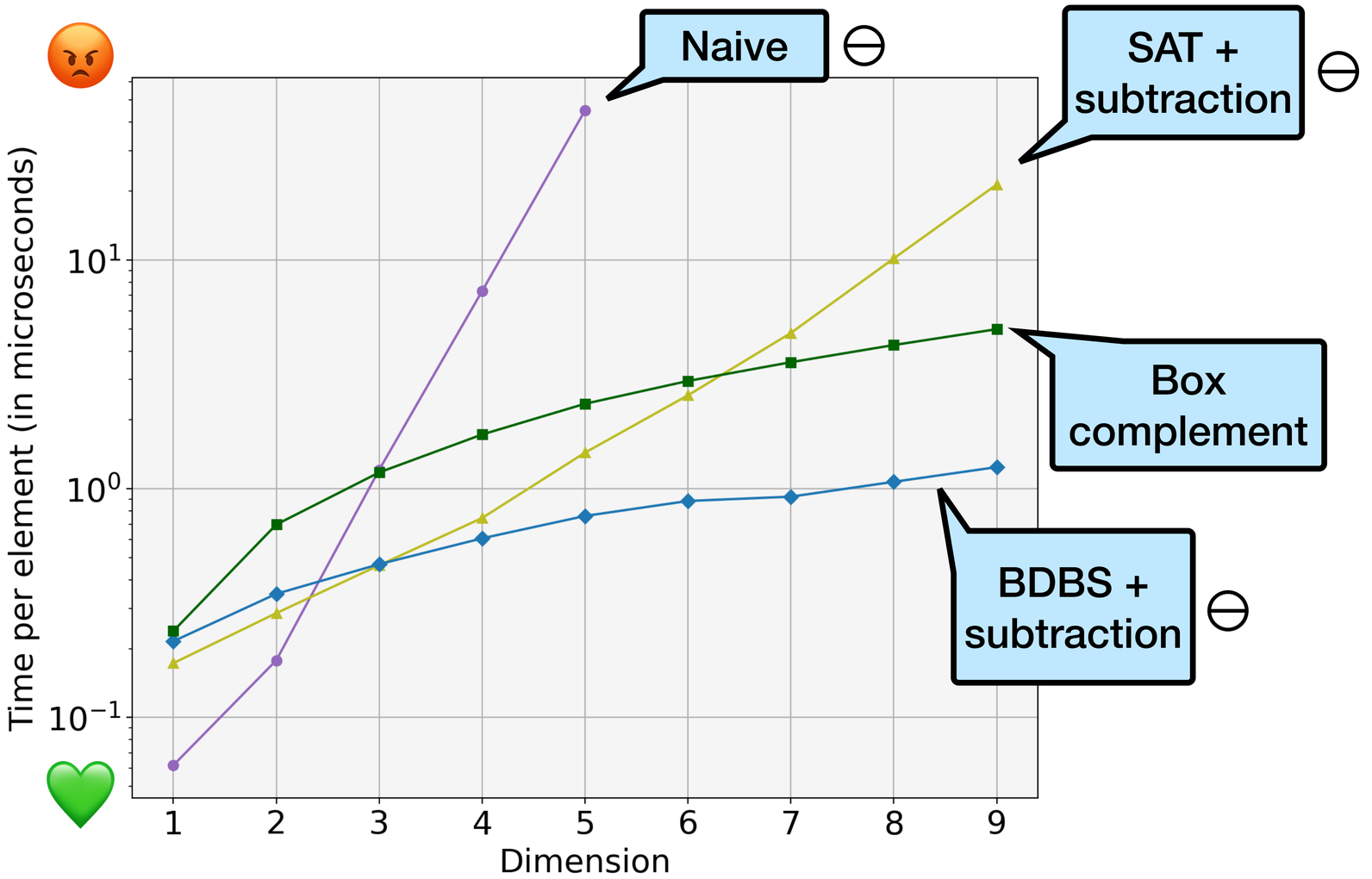
Algorithmic Bounds

Given a d-dimensional tensor with N elements:

Algorithm	Problem	Weak/Strong	Time	Space
Summed-area table	Included	Weak 😐	$\Theta(2^d N)$ 😐	$\Theta(N)$ 💚
Corners(c)	Excluded	Strong 💚	$\Omega(2^d N)$ 😐	$\Theta(cN)$ 😡
Bidirectional box-sum (BDBS)	Included	Strong 💚	$\Theta(dN)$ 💚	$\Theta(N)$ 💚
Box complement	Excluded	Strong 💚	$\Theta(dN)$ 💚	$\Theta(N)$ 💚

"Multidimensional Included and Excluded Sums," Xu, Fraser, and Leiserson. ACDA 2021.

Weak and Strong Excluded Sums in Higher Dimensions

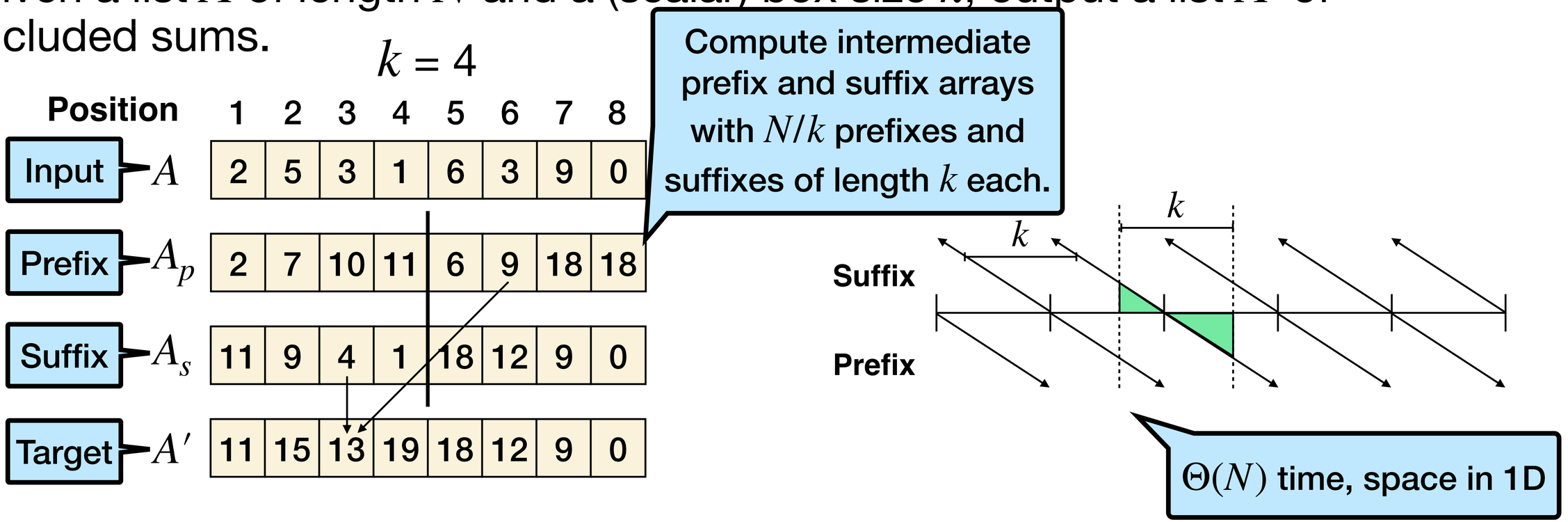


"Multidimensional Included and Excluded Sums," Xu, Fraser, and Leiserson. ACDA 2021.

Bidirectional Box-sum Algorithm for Strong Included Sums

We will start with the bidirectional box-sum algorithm (BDBS) in one dimension then show how to extend the technique to higher dimensions.

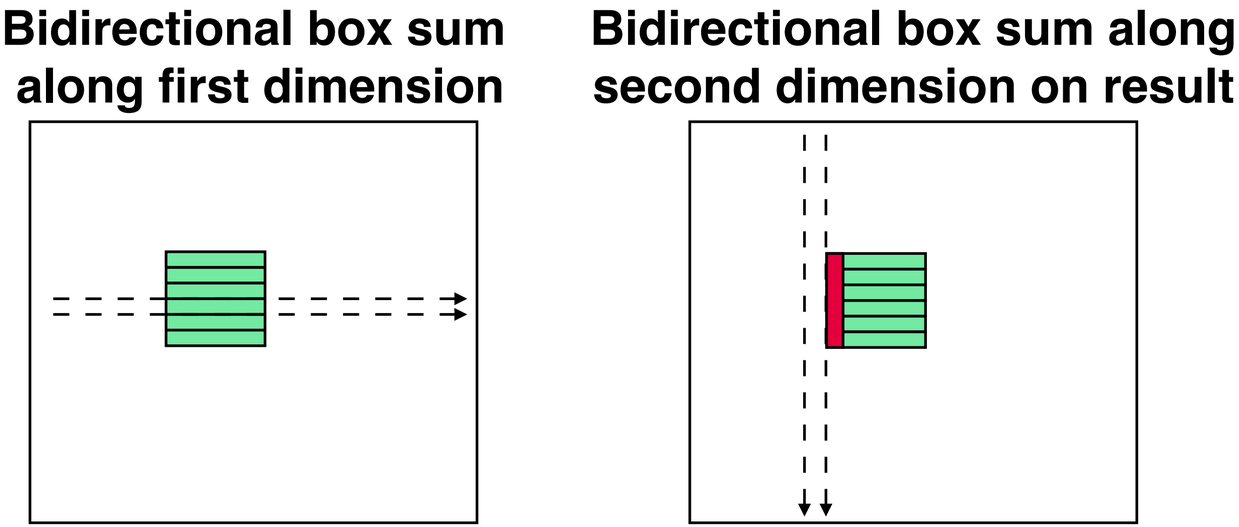
Given a list A of length N and a (scalar) box size k, output a list A' of included sums.



"Multidimensional Included and Excluded Sums," Xu, Fraser, and Leiserson. ACDA 2021.

Multidimensional Bidirectional Box Sum

The BDBS technique extends into arbitrary dimensions by performing the prefixes and suffixes along each dimension in turn.

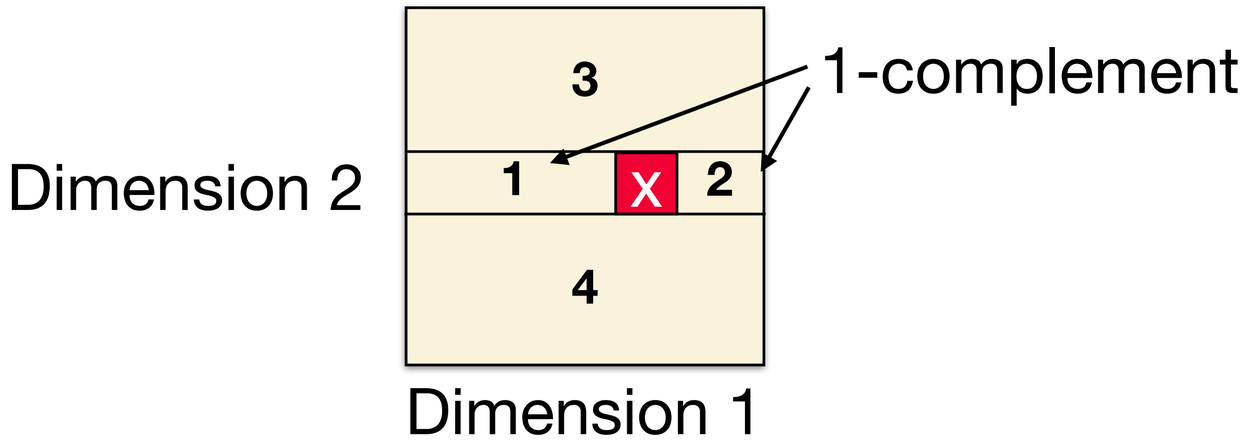


Given a d-dimensional tensor with N elements, BDBS solves the strong included-sums problem in $\Theta(dN)$ time and $\Theta(N)$ space.

Formulating the Excluded Sum as the Box Complement

Given a d-dimensional tensor and a "box size", we will first sketch how to decompose the excluded region for each point into $2d$ disjoint regions.

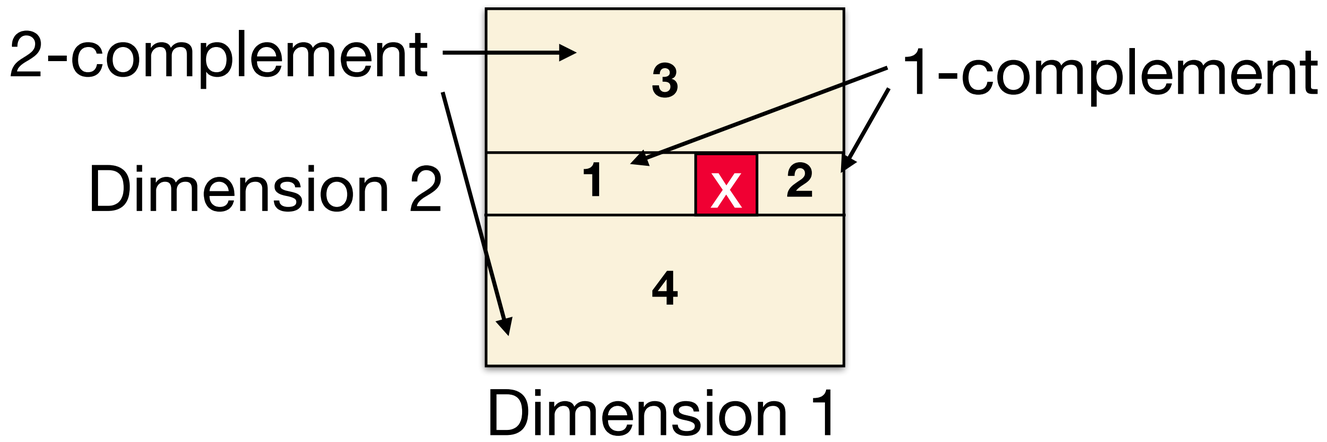
At a high level, the "i-complement" of a box such that there is some coordinate in dimension $j \in [1,i]$ that is "out of range" in dimension j, and the coordinates are "in range" for all dimensions $m \in [i+1,d]$.



Formulating the Excluded Sum as the Box Complement

Given a d-dimensional tensor and a "box size", we will first sketch how to decompose the excluded region for each point into $2d$ disjoint regions.

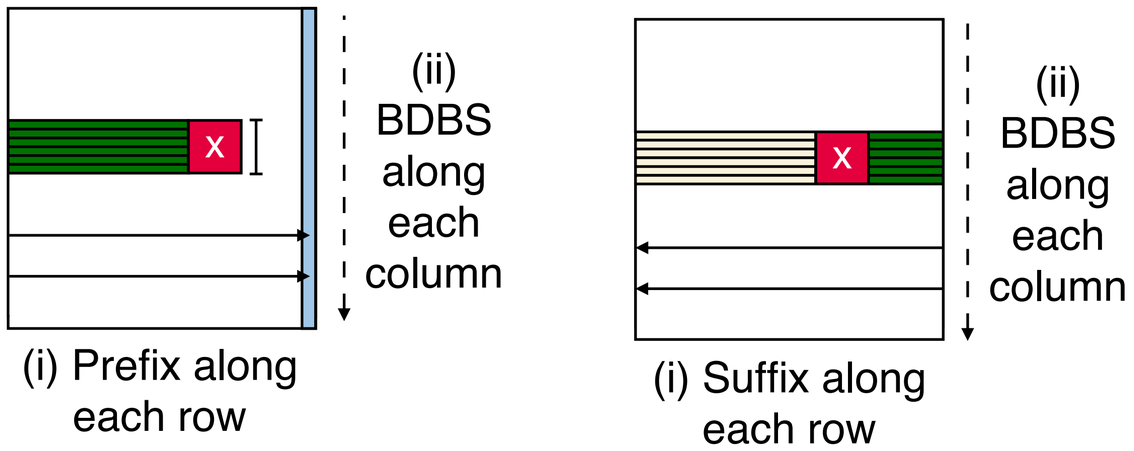
At a high level, the "i-complement" of a box such that there is some coordinate in dimension $j \in [1,i]$ that is "out of range" in dimension j, and the coordinates are "in range" for all dimensions $m \in [i+1,d]$.



Box-Complement Algorithm for Strong Excluded Sums

The box-complement algorithm uses **dimension reduction** to compute the "i-complement" for all $i = 1, \ldots, d$.

The **BDBS algorithm** for included sums is a major subroutine in the box-complement algorithm to sum up elements "in the range."

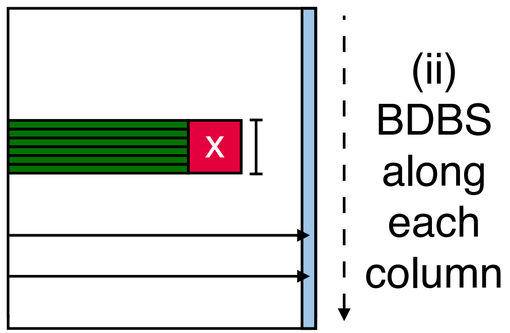


"Multidimensional Included and Excluded Sums," Xu, Fraser, and Leiserson. ACDA 2021.

Box-Complement Algorithm for Strong Excluded Sums

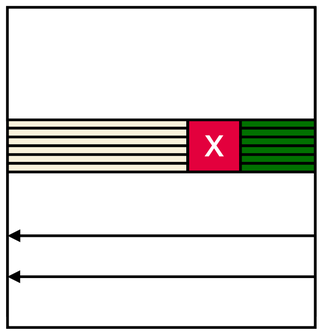
The box-complement algorithm uses **dimension reduction** to compute the "i-complement" for all $i = 1,\ldots,d$.

The **BDBS algorithm** for included sums is a major subroutine in the box-complement algorithm to sum up elements "in the range."



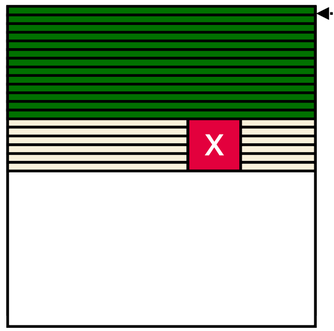
(ii) BDBS along each column

(i) Prefix along each row

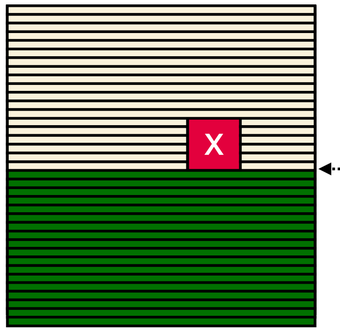


(ii) BDBS along each column

(i) Suffix along each row



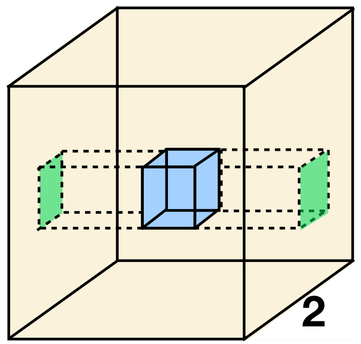
Prefix



Suffix

62

Extending the Box-complement Algorithm to Higher Dimensions

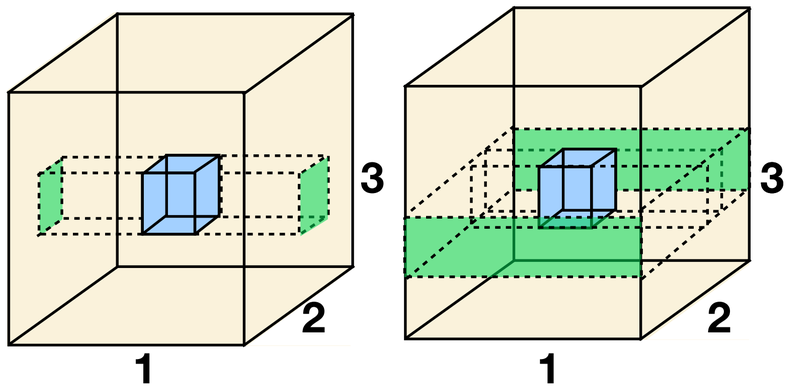


Full Prefix / Suffix Dimensions: 1

Included Sum Dimensions: 2, 3

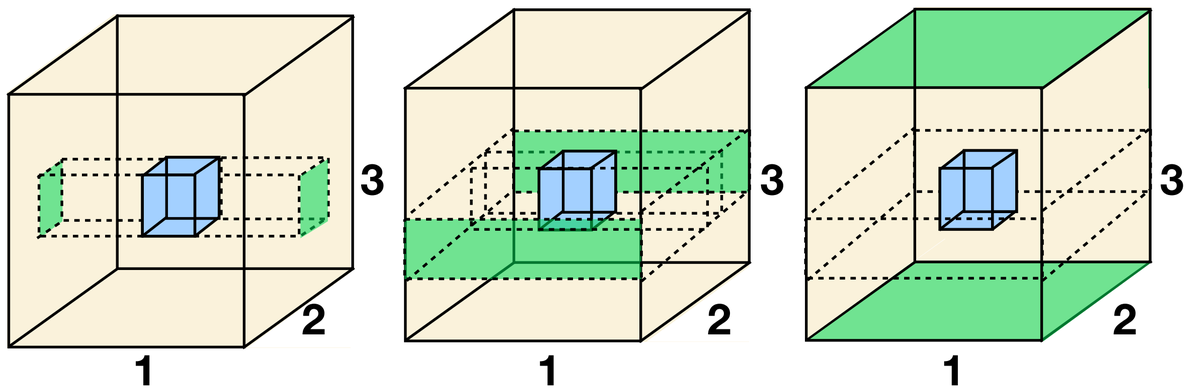
"Multidimensional Included and Excluded Sums," Xu, Fraser, and Leiserson. ACDA 2021.

Extending the Box-complement Algorithm to Higher Dimensions



Full Prefix / Suffix Dimensions:	**1**	**1, 2**
Included Sum Dimensions:	**2, 3**	**3**

Extending the Box-complement Algorithm to Higher Dimensions



Full Prefix / Suffix Dimensions:	**1**	**1, 2**	**1, 2, 3**
Included Sum Dimensions:	**2, 3**	**3**	**None**

Each dimension-reduction step takes $\Theta(N)$ time and reuses the same temporaries, for a total of $\Theta(dN)$ time and $\Theta(N)$ space.

"Multidimensional Included and Excluded Sums," Xu, Fraser, and Leiserson. ACDA 2021.