Chapter 15. Parallel Algorithms

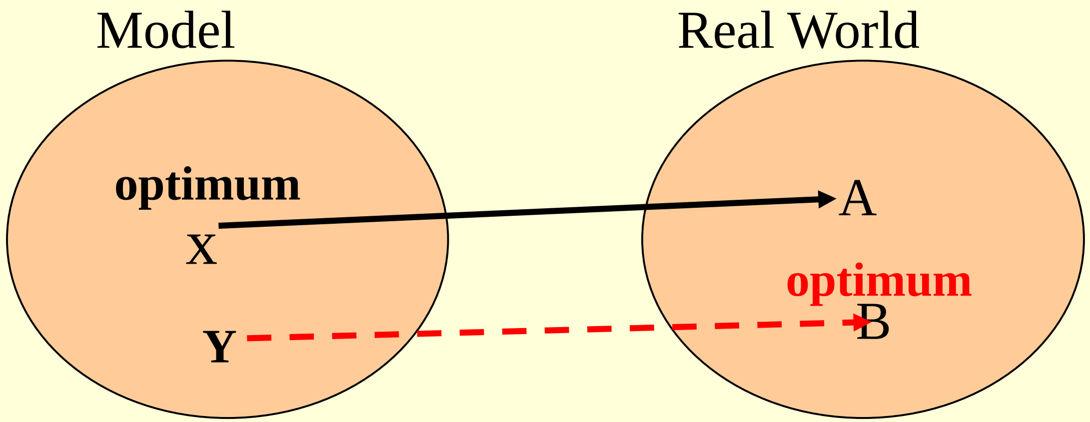
slides are freely borrowed from the web

Computation Models

- Goal of computation model is to provide a realistic representation of the costs of programming.
- Model provides algorithm designers and programmers a measure of algorithm complexity which helps them decide what is "good" (i.e., efficient)

Goal for Modeling

- We want to develop computational models which accurately represent the cost and performance of programs
- If model is poor, optimum in model may not coincide with optimum observed in practice



Models of Computation

What's a model good for??

- Provides a way to think about computers. Influences design of:
  - Architectures
  - Languages
  - Algorithms
- Provides a way of estimating how well a program will perform.

  Cost in model should be roughly same as cost of executing program

The Random Access Machine Model

RAM model of serial computers:

- Memory is a sequence of words, each capable of containing an integer.
- Each memory access takes one unit of time
- Basic operations (add, multiply, compare) take one unit time.
- Instructions are not modifiable
- Read-only input tape, write-only output tape

What about parallel computers

- RAM model is generally considered a very successful "bridging model" between programmer and hardware.
- "Since RAM is so successful, let's generalize it for parallel computers ..."

PRAM [Parallel Random Access Machine]

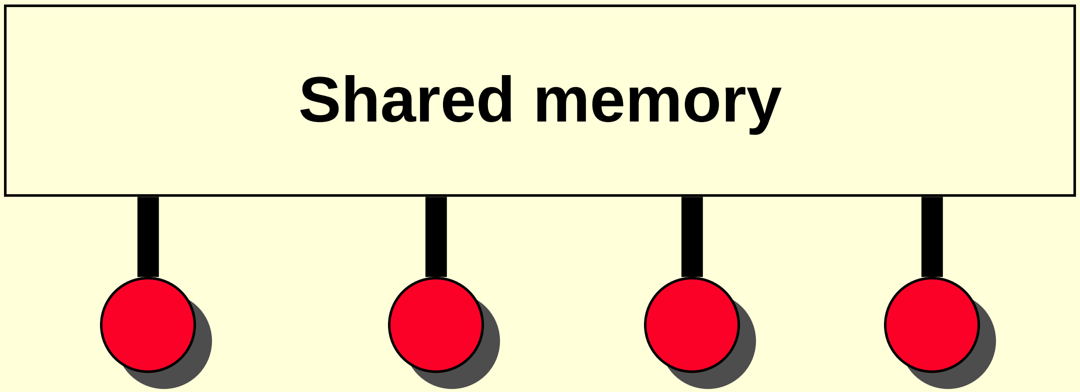
(Introduced by Fortune and Wyllie, 1978)

PRAM composed of:

- P processors, each with its own unmodifiable program.
- A single shared memory composed of a sequence of words, each capable of containing an arbitrary integer.
- a read-only input tape.
- a write-only output tape.

PRAM model is a synchronous, MIMD, shared address space parallel computer.

PRAM model of computation



- *p* processors, each with local memory
- Synchronous operation
- Shared memory reads and writes
- Each processor has unique id in range [1..*p*]
- At each unit of time, a processor is either active or idle (depending on id)

Common Simplifying Assumptions

- Infinite number of processors
- Any memory location is uniformly accessible from any processor
- Infinite amount of shared memory
- SIMD (Single Instruction Multiple Data)
  - All processors execute same program
  - At each time step, all processors execute same instruction on different data ("data-parallel")

Variants of PRAM model

	Exclusive Write	**Concurrent Write**
Exclusive Read	EREW	ERCW
Concurrent Read	CREW	CRCW

More PRAM taxonomy

- Different protocols can be used for reading and writing shared memory.
  - EREW - exclusive read, exclusive write

    A program isn't allowed to have two processors access the same memory location at the same time.
  - CREW - concurrent read, exclusive write
  - ERCW - exclusive read, concurrent write
  - CRCW - concurrent read, concurrent write

    Needs protocol for arbitrating write conflicts
- PRAM can emulate a message-passing machine by partitioning memory into private memories.

Why study PRAM algorithms?

- Well-developed body of literature on design and analysis of such algorithms
- Baseline model of concurrency
- Explicit model
  - Specify operations at each step
  - Scheduling of operations on processors
- Robust design paradigm

Work-Span paradigm

- Higher-level abstraction for PRAM algorithms
- Span (Critical Path Length): T(n)
  - Most expensive path from beginning to end of a PRAM algorithm
- Work: W(n)
  - Total amount of processor time to complete executing the algorithm
- *Work-efficient* if $W(n) = \Theta(T_S(n))$

optimal sequential algorithm's time complexity

Designing PRAM algorithms

- Balanced trees
- Pointer jumping
- Divide and conquer
- . . .

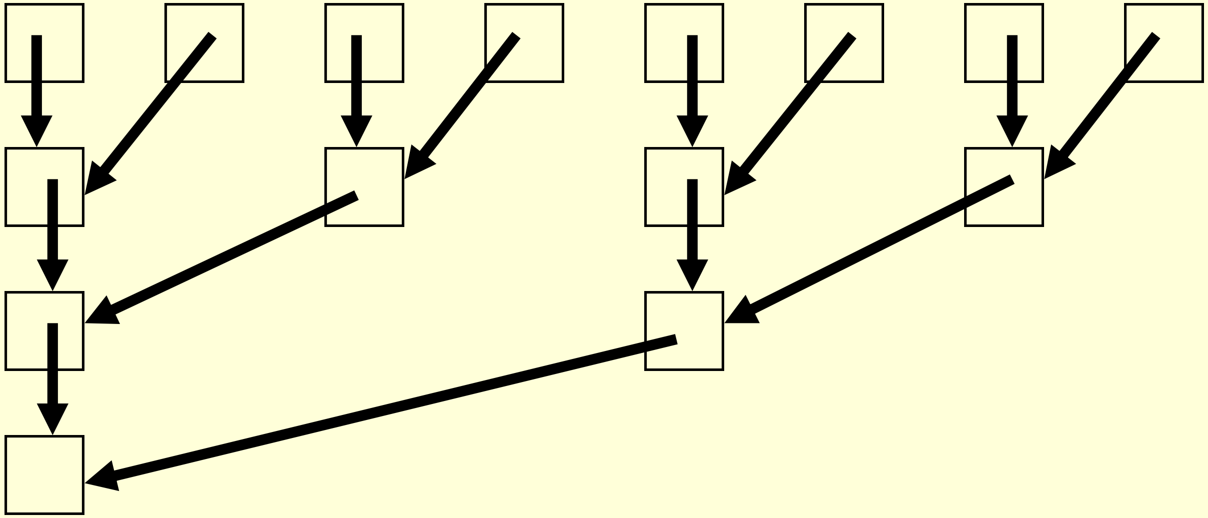
Balanced trees

- Key idea: Build balanced binary tree on input data, sweep tree up and down
- "Tree" not a data structure, often a control structure (e.g., recursion)

Parallel Sum

Parallel Sum

- Given: Sequence $\boldsymbol{a}$ of $n = 2^k$ elements
- Given: Binary associative operator +
- Compute: $S = \boldsymbol{a}_1 + \ldots + \boldsymbol{a}_n$



Parallel sum pseudo code

```
integer  B[1..n]
forall i in 1 : n  do
    B[i] := a_i
enddo
for h = 1 to k do
    forall i in 1 : n/2^h do
        B[i] := B[2i-1] + B[2i]
    enddo
enddo
S := B[1]
```

Interesting Points

- Global program: no references to processor ID
- Contains both serial and concurrent operations
- Semantics of ***forall***
  - In each iteration of the *forall* loop, a processor does the addition or sits idle depending on its ID
- Order of additions different from sequential order: associativity critical

Analysis of parallel sum

- Algorithm is correct
  - $\Theta(\lg n)$ steps
  - $\Theta(n)$ work
    - In total, n/2 + n/4 + … + 1 = n-1 additions
- EREW model
- If *n* not power of 2, pad to next power

Complexity measures of parallel sum

$$T(n) = 1 + k + 1 = \Theta(\lg n)$$

$$W(n) = n + \sum_{h=1}^{k} \frac{n}{2^h} + 1$$
$$= \Theta(n)$$

- Span is O(logn): Concurrent execution reduces number of steps (shorter critical path length)
- Work: O(n)
- Speedup: O(n/logn)
  - Optimal serial sum algorithm is O(n)

Parallel Prefix Sum

What is prefix sum ?

- Input: Sequence x of $n = 2^k$ elements, binary associative operator +
- Output: Sequence s of $n = 2^k$ elements, with $s_k = x_1 + \ldots + x_k$
- Example:

  x = [1, 4, 3, 5, 6, 7, 0, 1]

  s = [1, 5, 8, 13, 19, 26, 26, 27]

(Inclusive) Prefix-Sum (Scan) Definition

Definition: *The* all-prefix-sums *operation takes a binary associative operator* $\oplus$*, and an array of n elements*

$$[x_0, x_1, \ldots, x_{n-1}],$$

and returns the array

$$[x_0, (x_0 \oplus x_1), \ldots, (x_0 \oplus x_1 \oplus \ldots \oplus x_{n-1})].$$

Example: If $\oplus$ is addition, then the all-prefix-sums operation on the array [3 1 7 0 4 1 6 3], would return [3 4 11 11 15 16 22 25].

Inclusive Scan Application Example

- Assume we have a 100-inch sandwich to feed 10
- We know how many inches each person wants
  - [3 5 2 7 28 4 3 0 8 1]
- How do we cut the sandwich quickly?
- How much will be left?

- Method 1: cut the sections sequentially: 3 inches first, 5 inches second, 2 inches third, etc.
- Method 2: calculate Prefix scan and cut in parallel
  - [3, 8, 10, 17, 45, 49, 52, 52, 60, 61] (39 inches left)

Typical Applications of Scan

- Scan is a simple and useful parallel building block
  - Convert recurrences from sequential:

```
for(j=1;j<n;j++)
    out[j] = out[j-1] + f(j);
```

  - into parallel:

```
forall(j) { temp[j] = f(j) };
scan(out, temp);
```

- Useful for many parallel algorithms:

•Radix sort
•Quicksort
•String comparison
•Lexical analysis
•Stream compaction
•Polynomial evaluation
•Solving recurrences
•Tree operations
•Histograms
•Etc.

Other Applications

- Assigning space in farmers market
- Allocating memory to parallel threads
- Allocating memory buffer for communication channels
- …

An Inclusive Sequential Prefix-Sum

Given a sequence $[x_0, x_1, x_2, \ldots]$

Calculate output $[y_0, y_1, y_2, \ldots]$

Such that

$$y_0 = x_0$$

$$y_1 = x_0 + x_1$$

$$y_2 = x_0 + x_1 + x_2$$

$$\ldots$$

Using a recursive definition

$$y_i = y_{i-1} + x_i$$

A Work Efficient C Implementation

```
y[0] = x[0];
for (i=1; i < Max_i; i++)
    y[i] = y[i-1] + x[i];
```

Computationally efficient:

N additions needed for N elements - **O(N)**

A Naïve Inclusive Parallel Scan

- Assign one thread to calculate each y element
- Have every thread add up all x elements needed for the y element

$$y_0 = x_0$$

$$y_1 = x_0 + x_1$$

$$y_2 = x_0 + x_1 + x_2$$

Span & Work of this naïve algorithm?

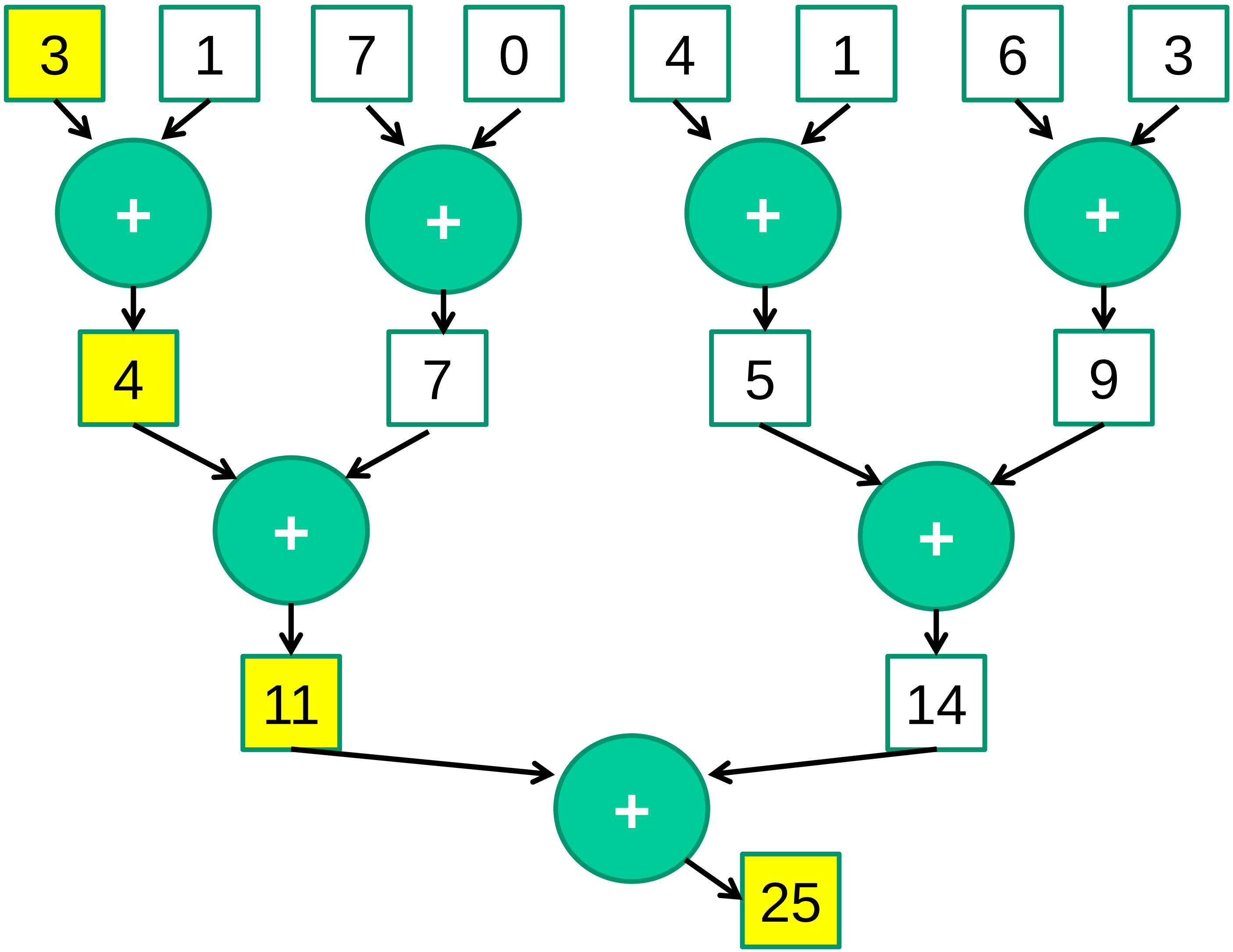
Parallel programming is easy as long as you don't care about performance.

Improving Efficiency

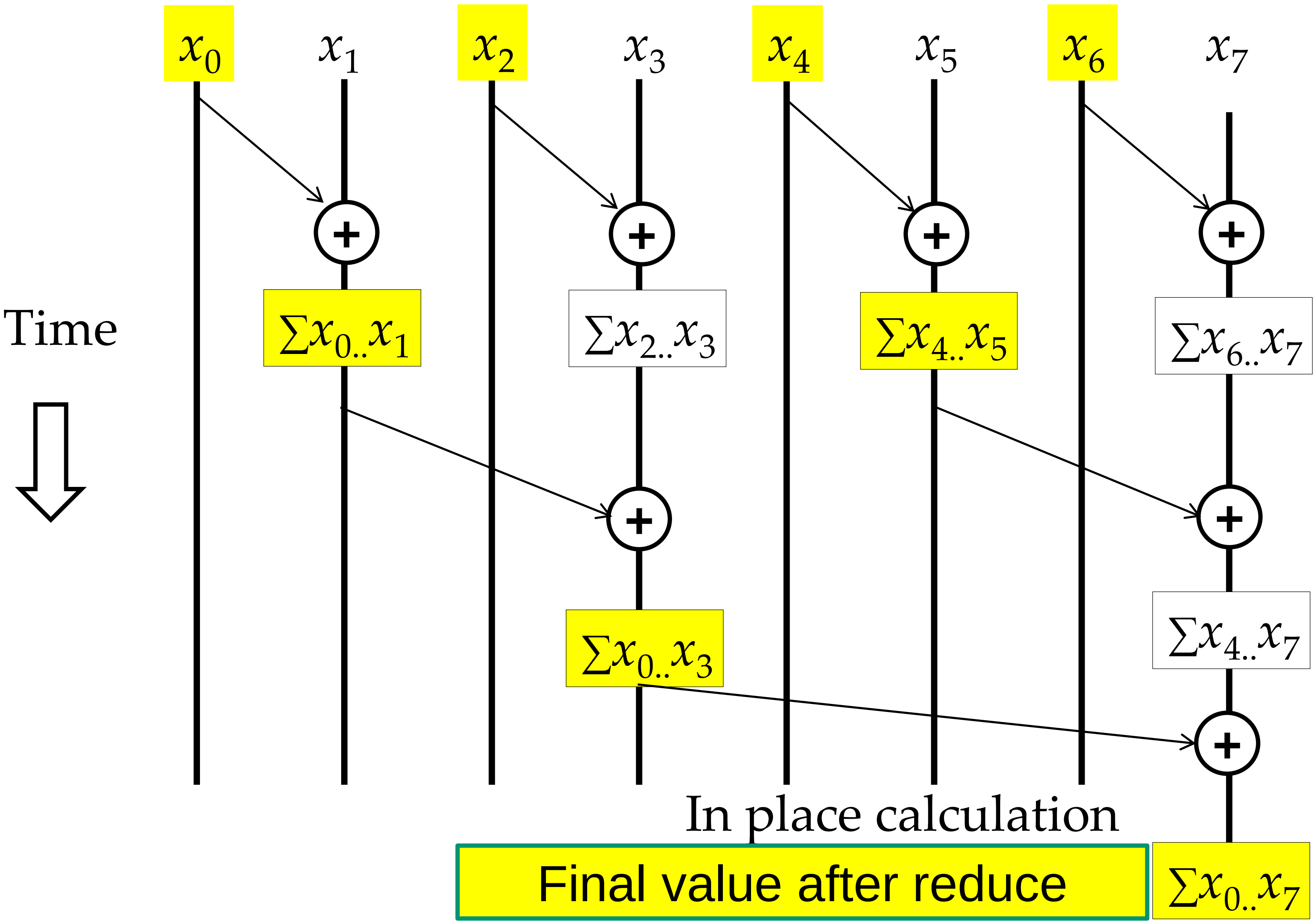
- A common parallel algorithm pattern:

  Balanced Trees
  - Build a balanced binary tree on the input data and sweep it to and from the root
  - Tree is not an actual data structure, but a concept to determine what each thread does at each step
- For scan:
  - Traverse down from leaves to root building partial sums at internal nodes in the tree
    - Root holds sum of all leaves
  - Traverse back up the tree building the scan from the partial sums

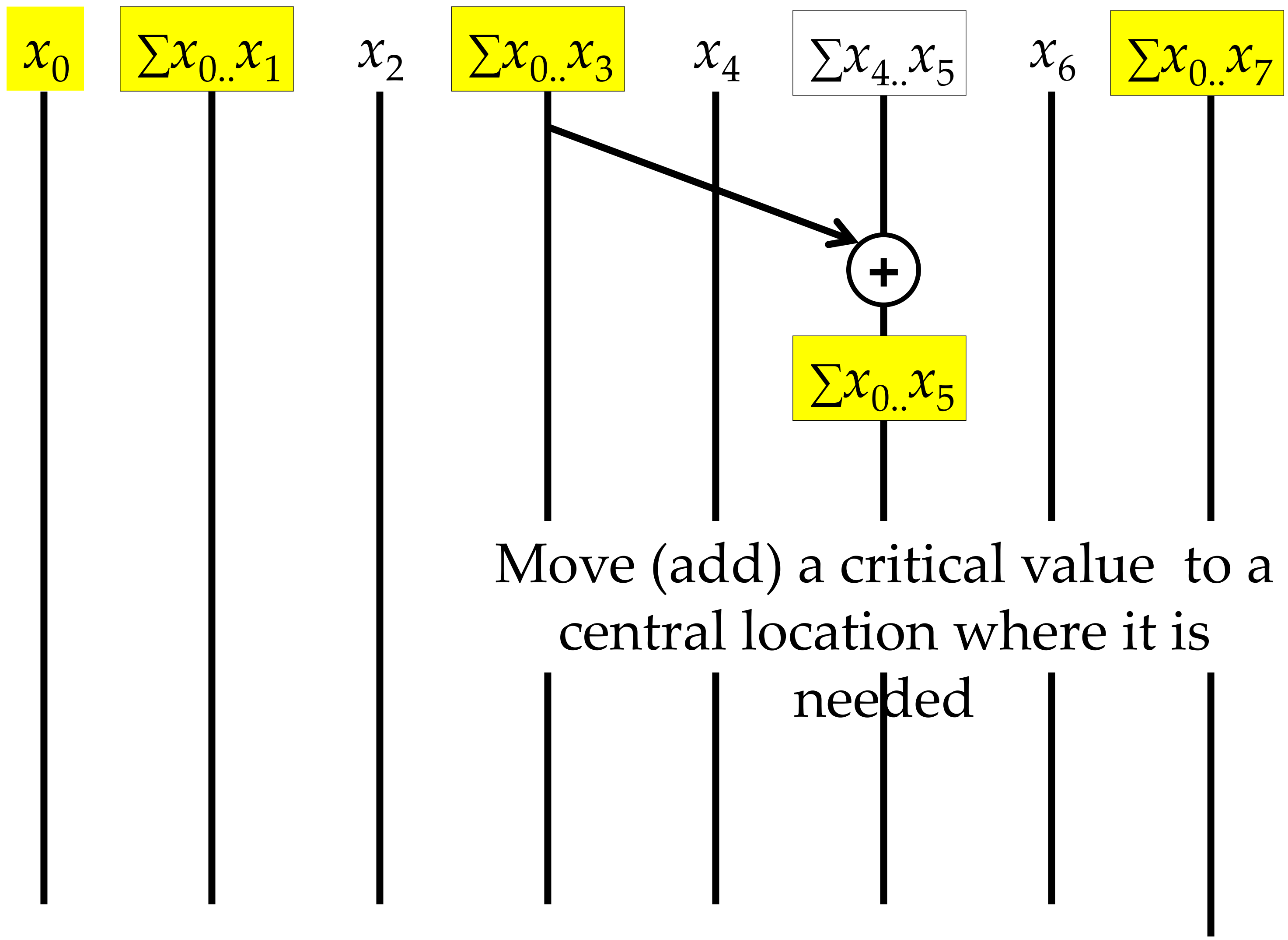
Let's Look at the Reduction Tree Again



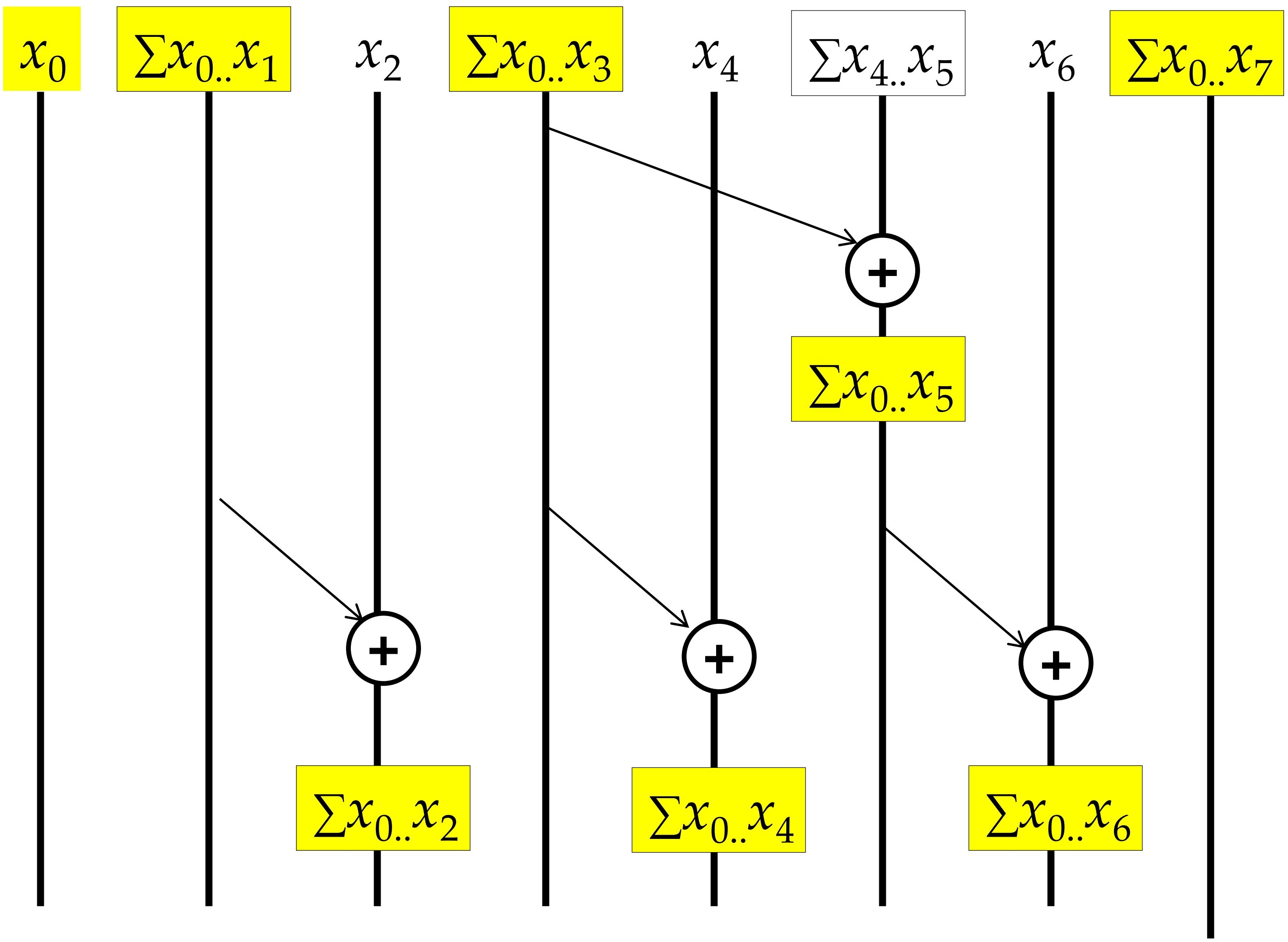
Parallel Scan – Reduction Step



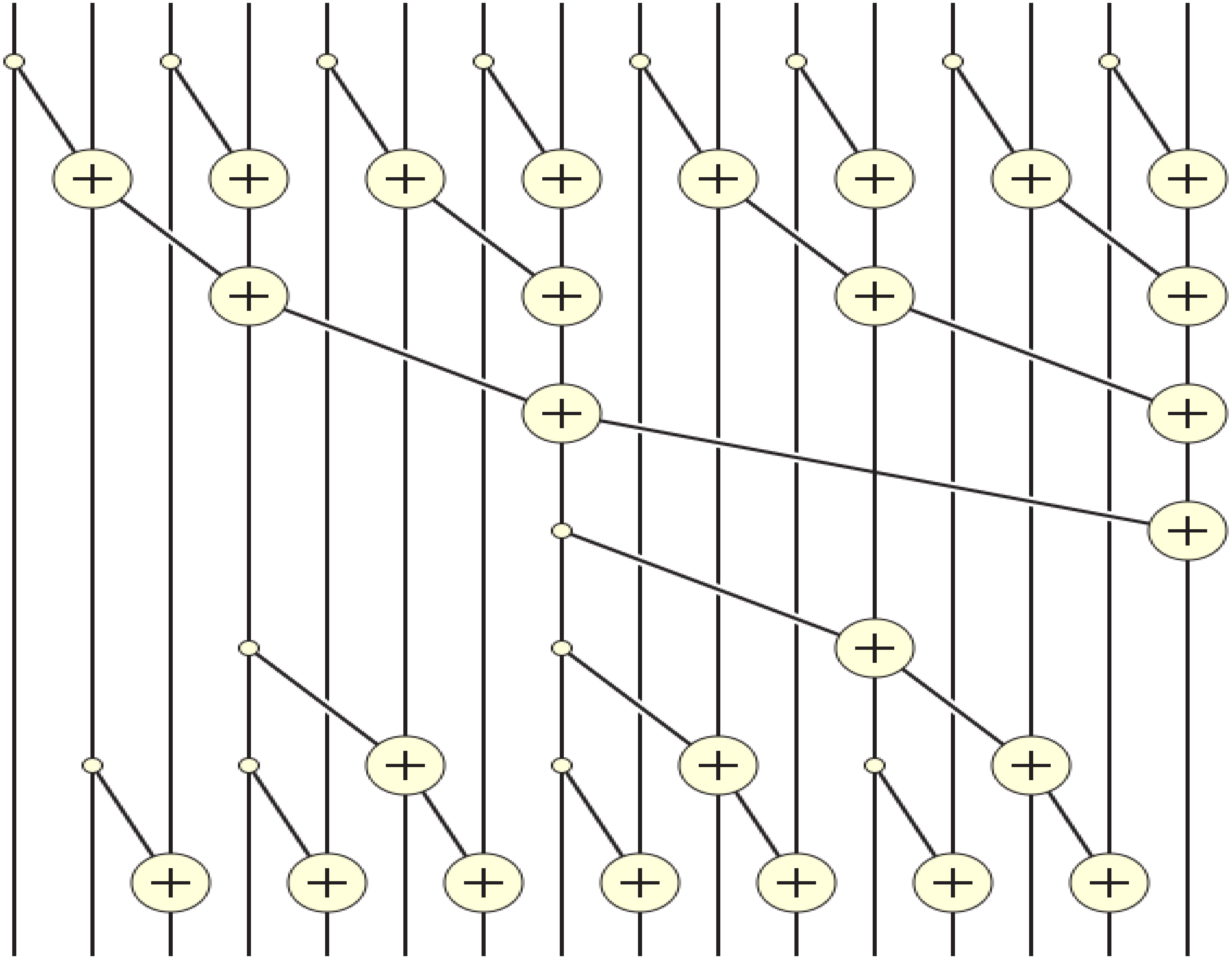
Inclusive Post Scan Step



Inclusive Post Scan Step



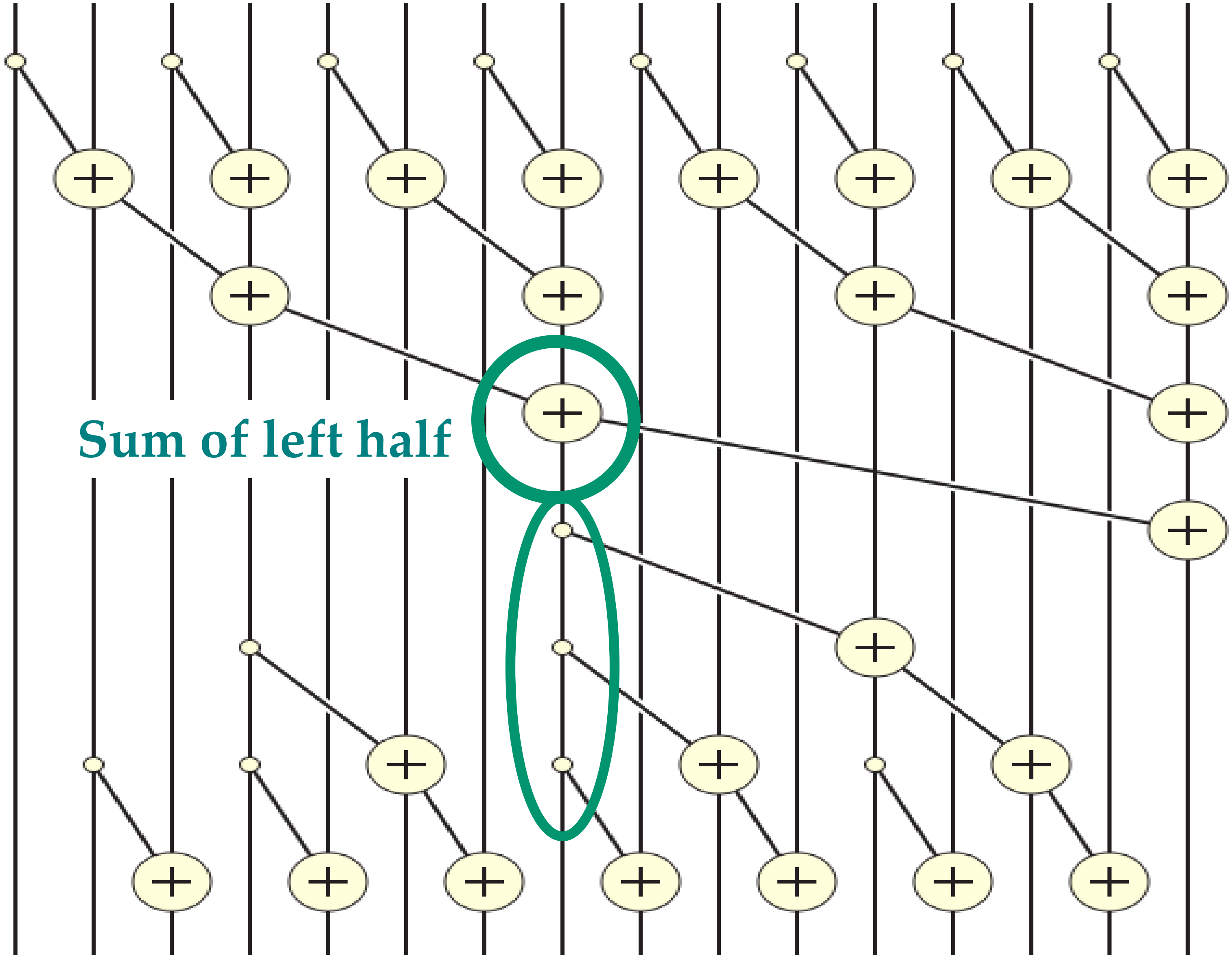
Putting it Together



Parallel Prefix Sum Implementations

- There are many multicore CPU implementations of parallel prefix sum
- CUDA implementation: http.developer.nvidia.com/GPUGems3/gpugems3_ch39.html

Putting it together



(Exclusive) Prefix-Sum (Scan) Definition

Definition: *The* all-prefix-sums *operation takes a binary associative operator* $\oplus$, *and an array of n elements*

$$[x_0, x_1, \ldots, x_{n-1}],$$

and returns the array

$$[0, x_0, (x_0 \oplus x_1), \ldots, (x_0 \oplus x_1 \oplus \ldots \oplus x_{n-2})].$$

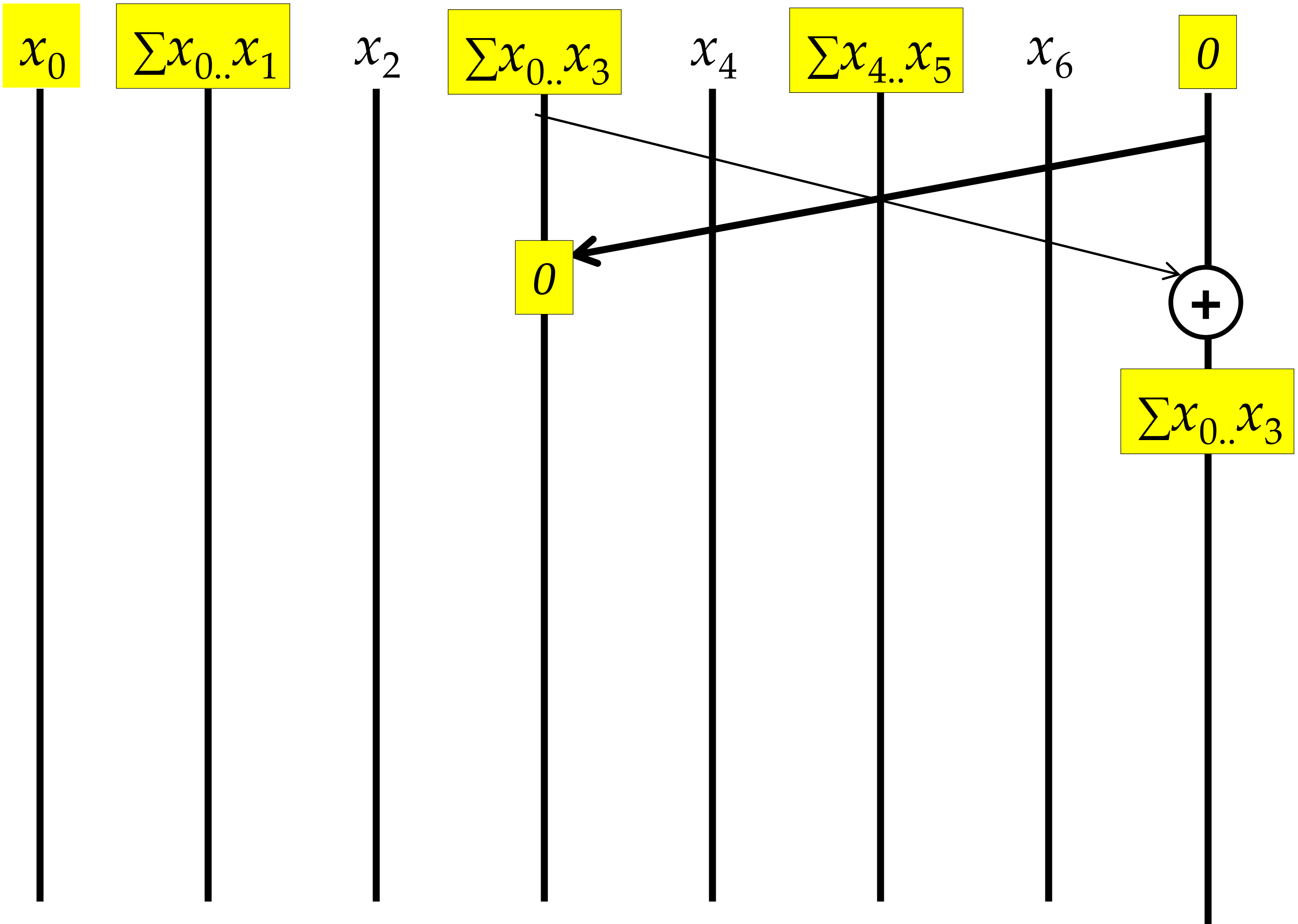
Example: If $\oplus$ is addition, then the all-prefix-sums operation on the array [3 1 7 0 4 1 6 3],
would return [0 3 4 11 11 15 16 22].

Why Exclusive Scan

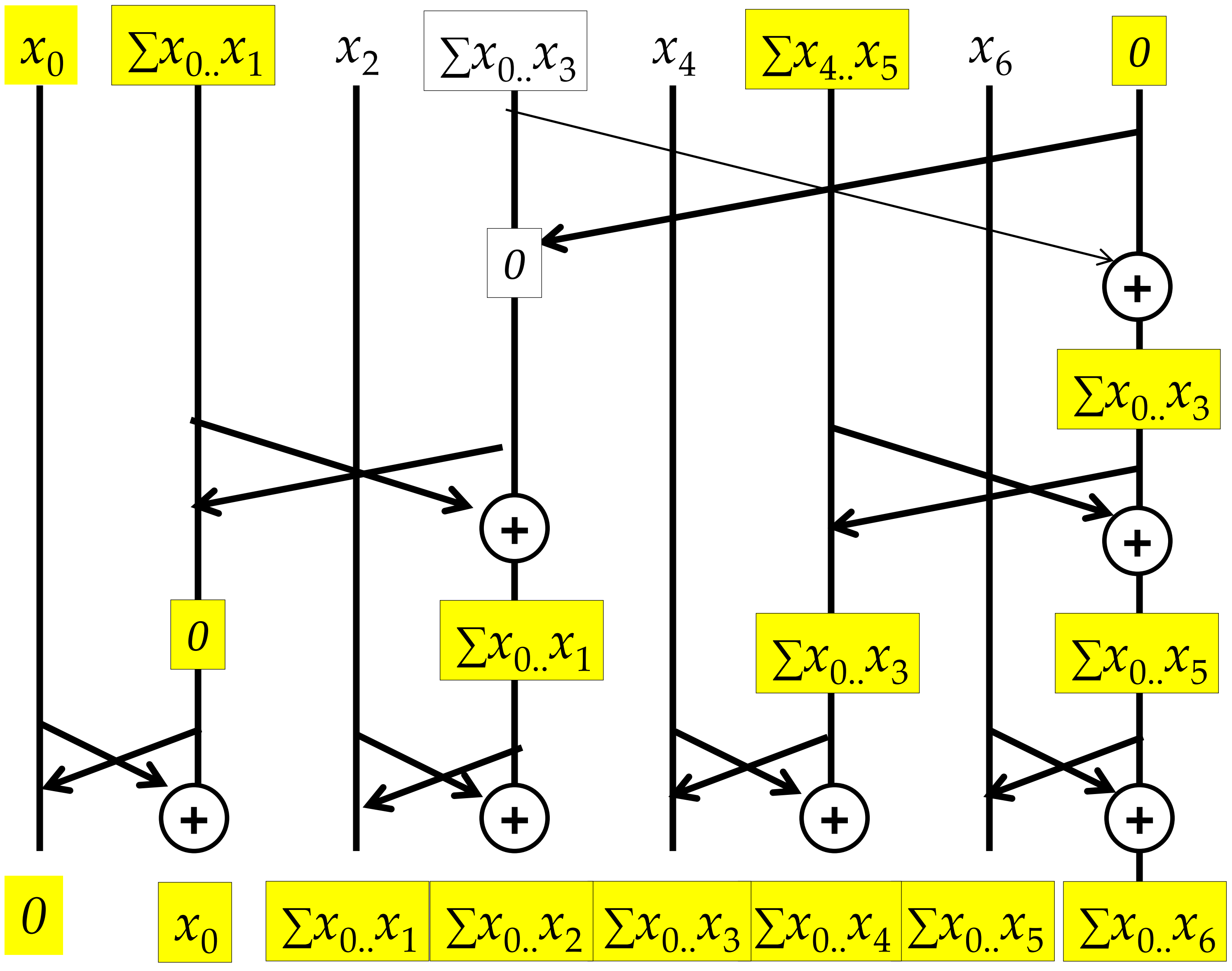
- To find the beginning address of allocated buffers
- Inclusive and Exclusive scans can be easily derived from each other; it is a matter of convenience

	[3 1 7 0 4 1 6 3]
Exclusive	[0 3 4 11 11 15 16 22]
Inclusive	[3 4 11 11 15 16 22 25]

Exclusive Post Scan Step (Add-move Operation)



Exclusive Post Scan Step



Work Analysis

- The parallel Inclusive Scan executes 2*log(n) parallel iterations. Thus, Span is **O(logn)**
  - log(n) in reduction and log(n) in post scan
  - The iterations do n/2, n/4,..1, 1, …., n/4, n/2 adds
  - Total adds: 2* (n-1) → **O(n)** work

- The total number of adds is no more than twice that done in the efficient sequential algorithm
  - The benefit of parallelism can easily overcome the 2X work when there is sufficient hardware

A problem hard to parallelize: Filter

Filter

[Non-standard terminology]

Given an array `input`, produce an array `output` containing only elements such that `f(elt)` is `true`

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

Looks hard to parallelize
- Finding elements for the output is easy
- But getting them in the right place is hard

Spring 2010 CSE332: Data Abstractions 1

- Slide source: CSE332, University of Washington

Prefix sum to rescue

- O(logn) span, O(n) work

1. Use a parallel map to compute a bit-vector for true elements

```
input   [17, 4, 6, 8, 11, 5, 13, 19, 0, 24]
bits    [1,  0, 0, 0,  1, 0,  1,  1, 0,  1]
```

2. Do parallel-prefix sum on the bit-vector

```
bitsum [1,  1, 1, 1,  2, 2,  3,  4, 4,  5]
```

3. Use a parallel map to produce the output

```
output [17, 11, 13, 19, 24]
```

```
output = new array of size bitsum[n-1]
if(bitsum[0]==1) output[0] = input[0];
FORALL(i=1; i < input.length; i++)
  if(bitsum[i] > bitsum[i-1])
    output[bitsum[i]-1] = input[i];
```

Parallel quicksort: Algorithm 1

	Best / expected case *work*
1. **Pick a pivot element**	**O(1)**
2. **Partition all the data into:**	**O(n)**
A. **The elements less than the pivot**	
B. **The pivot**	
C. **The elements greater than the pivot**	
3. **Recursively sort A and C**	**2T(n/2)**

Easy: Do the two recursive calls in parallel

- Work: unchanged of course $O(n \log n)$
- Span: Now $O(n) + 1T(n/2) = O(n)$
- So parallelism (i.e., work/span) is $O(\log n)$

- O(log n) speedup: Sort 10^9 elements 30 times faster

Parallel quicksort: Algorithm 2

Parallel partition (not in place)

Partition all the data into:

A. The elements less than the pivot
B. The pivot
C. The elements greater than the pivot

- This is just two filters!
  - We know a filter is $O(n)$ work, $O(\log n)$ span
  - Filter elements less than pivot into left side of `aux` array
  - Filter elements great than pivot into right size of `aux` array
  - Put pivot in-between them and recursively sort
  - With a little more cleverness, can do both filters at once but no effect on asymptotic complexity
- With $O(\log n)$ span for partition, the total span for quicksort is
  $O(\log n) + 1T(n/2) = O(\log^2 n)$

- O(n/log n) speedup!
- How much is the total work W(n) of this quicksort algorithm?

Parallel List Ranking

List Ranking

- List ranking problem
  - Given a singly linked list L with n objects, for each node, compute the distance to the end of the list
- If d denotes the distance
  - $$\text{node.d} = \begin{cases} 0 & \text{if node.next = nil} \\ \text{node.next.d} + 1 & \text{otherwise} \end{cases}$$
- Serial algorithm: O(n)
- Parallel algorithm
  - Assign one processor to each node
  - Assume there are as many processors as list objects
  - For each node i, do
    1. i.d = i.d + i.next.d
    2. i.next = i.next.next // pointer jumping

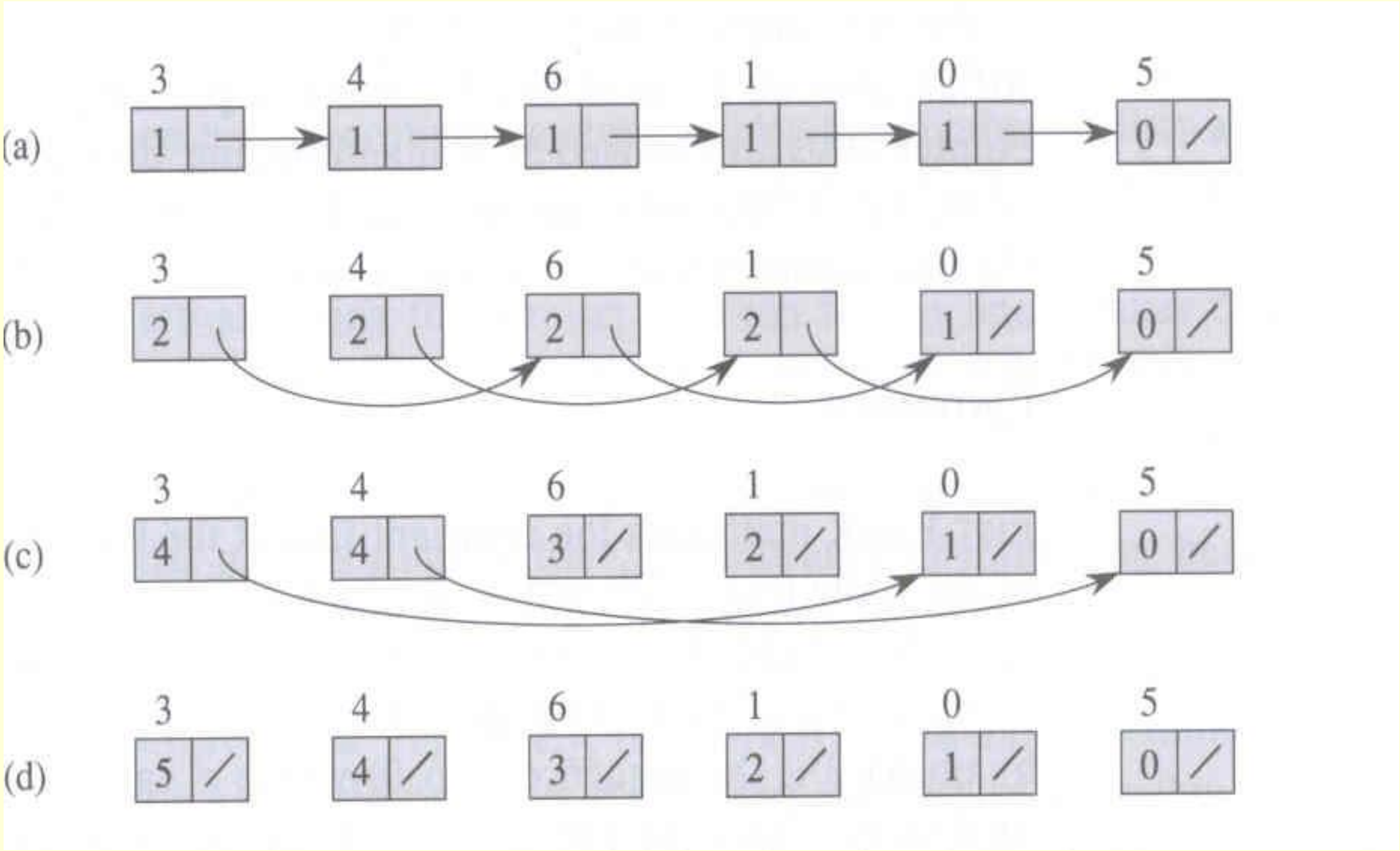
List Ranking via Pointer Jumping

- List_ranking(L)

```
for each node i, in parallel do
    if i.next = nil then i.d = 0
    else i.d = 1
while exists a node i, such that i.next != nil do
    for each node i, in parallel do
        if i.next != nil then
            i.d = i.d + i.next.d            // i updates i itself
            i.next = i.next.next
```

- Analysis
  - After a pointer jumping, a list is transformed into two (interleaved) lists
  - After that, four (interleaved) lists
  - Each pointer jumping doubles the number of lists and halves their length
  - After $\lceil \log n \rceil$, all lists contain only one node
  - Total time: $O(\log n)$

List Ranking - Example



List Ranking - Discussion

- Synchronization is important
  - In step 8 (i.next = i.next.next), all processors must read right hand side before any processor write left hand side
- The list ranking algorithm is EREW
  - If we assume in step 7 (i.d = i.d + i.next.d) all processors read i.d and then read i.next.d
  - If j.next = i, i and j do not read i.d concurrently
- Work performance
  - performs $O(n \log n)$ work since n processors in $O(\log n)$ time
- Work efficient
  - A PRAM algorithm is work efficient w.r.t another algorithm if two algorithms are within a constant factor
  - Is the link ranking algorithm work-efficient w.r.t the serial algorithm?
    - No, because $O(n \log n)$ versus $O(n)$
- Speedup
  - $S = n / \log n$

Parallel Prefix on a List (4)

- Running time (Span): $O(\log n)$
  - After $\lceil \log n \rceil$, all lists contain only one node
- Work performed: $O(n \log n)$
- Speedup
  - $S = n / \log n$

References

- Joseph JaJa, Introduction to Parallel Algorithms, Addison Wesley
- OpenMP: http://openmp.org/wp/
- CUDA Programming Guide:

http://docs.nvidia.com/cuda/cuda-c-programming-guide/#axzz4eumRgF4w