Advanced Algorithms

Module 2

Parallel Computing

Why Parallel Computing?

- Save time, resources, memory, ...
- Who is using it?
  - Academia
  - Industry
  - Government
  - Individuals?
- Two practical motivations:
  - Application requirements
  - Architectural concerns.
- Why now?
  - Most computers including laptops are multi-core!
  - Need to therefore study how to use parallel computers.

Conventional Wisdom in Computer Architecture

- Power Wall + Memory Wall + ILP Wall = Brick Wall
- Old CW: Uniprocessor performance 2X / 1.5 yrs
- New CW: Uniprocessor performance only 2X / 5 yrs?

The Academic Interest

- Algorithmics and compelxity
    - How to design parallel algorithms?
    - What are good theoretical models for parallel computing?
    - How to analyze parallel algorithms?
    - Can every sequential algorithm be parallelized?
    - What are some complexity classes wrt parallel computing?

The Academic Interest

- Systems and Programming
  - How to write parallel programs?
  - What are some tools and environments.
  - How to convert algorithms to efficient implementations.
  - What are the differences to sequential programming?
  - What are the performance measures?
  - Can sequential programs be automatically converted to parallel programs?

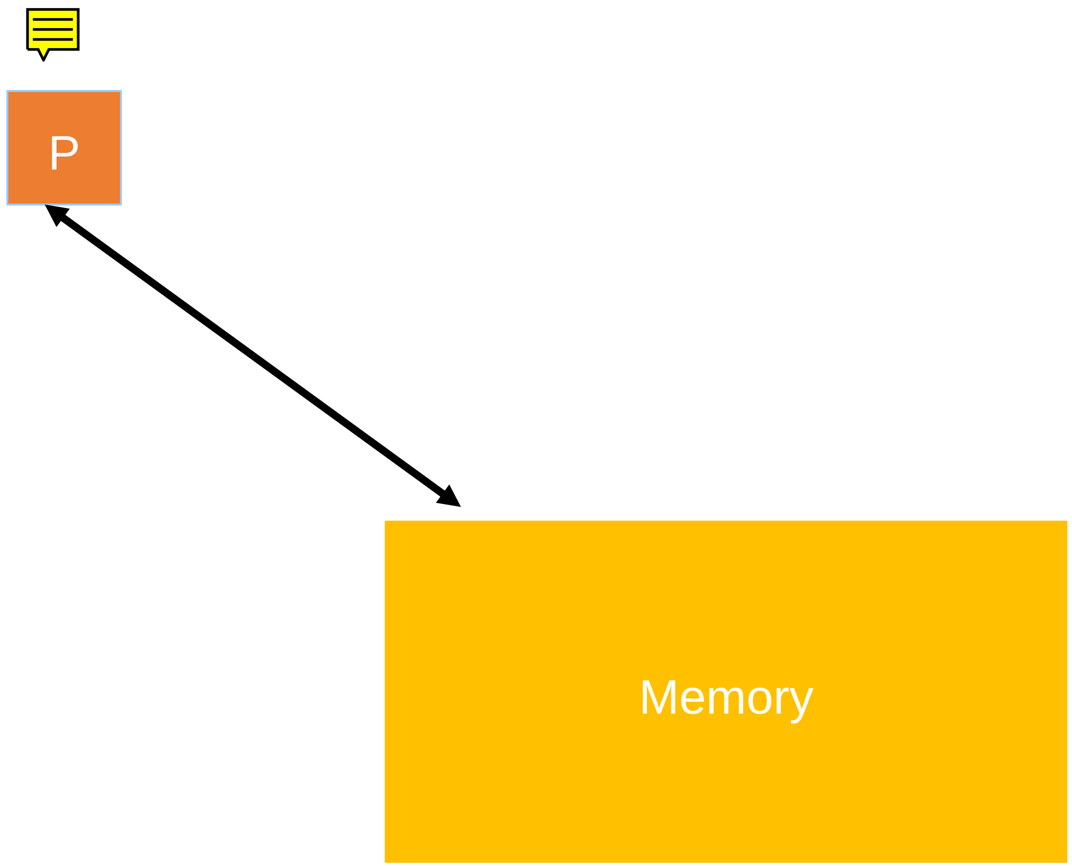
The Academic Interest

- Architectures
  - What are standard architectural designs?
  - What new issues are raised due to multiple cores?
  - Downstream concerns
    - Does a programmer have to worry about this?
    - How to support the systems software as architecture changes?

The Course Coverage

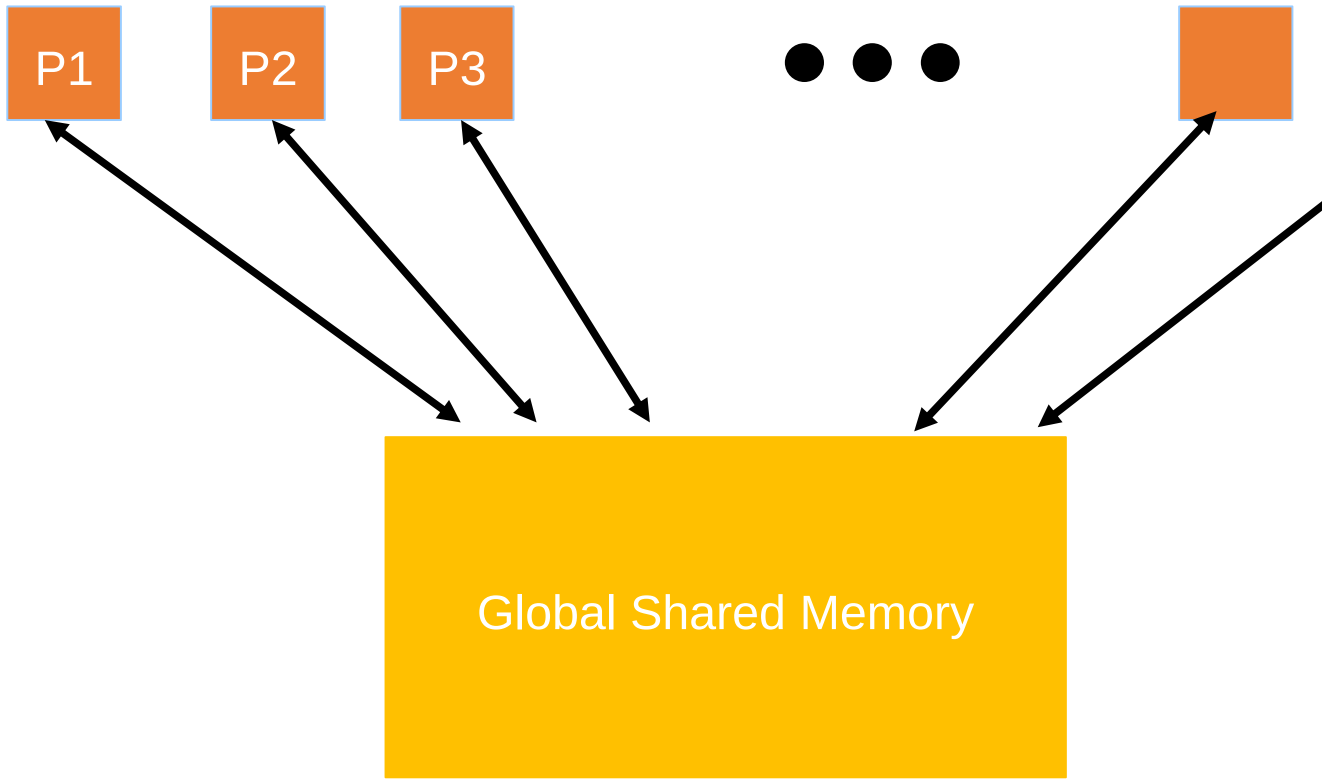
- Focus on algorithms and complexity
- Models for parallel algorithms
- Algorithm design methodologies with application
    - Semi-numerical
    - Lists
    - Trees and graphs
- Complexity, characterization, and connection to sequential complexity classes.

The PRAM Model



- The von Neumann model.

The PRAM Model



- An extension of the von Neumann model.

The PRAM Model

- A set of n identical processors
- A common access shared memory
- Synchronous time steps
- Access to the shared memory costs the same as a unit of computation.
- Different models to provide semantics for concurrent access to the shared memory
  - EREW, CREW, CRCW(Common, Aribitrary, Priority, ...)

The Semantics

- In all cases, it is the programmer to ensure that his program meets the required semantics.
- EREW : Exclusive Read, Exclusive Write
  - No scope for memory contention.
  - Usually the weakest model, and hence algorithm design is tough.
- CREW : Concurrent Read, Exclusive Write
  - Allow processors to read simultaneously from the same memory location at the same instant.
  - Can be made practically feasible with additional hardware

The Semantics

- CRCW : Concurrent Read, Concurrent Write
  - Allow processors to read/write simultaneously from/to the same memory location at the same instant.
  - Requires further specification of semantics for concurrent write. Popular variants include
    - COMMON : Concurrent write is allowed so long as the all the values being attempted are equal. Example: Consider finding the Boolean OR of n bits.
    - ARBITRARY : In case of a concurrent write, it is guaranteed that some processor succeeds and its write takes effect.
    - PRIORITY : Assumes that processors have numbers that can be used to decide which write succeeds.

PRAM Model – Advantages and Drawbacks

Advantages

- A simple model for algorithm design
- Hides architectural details for the designer.
- A good starting point

Disadvantages

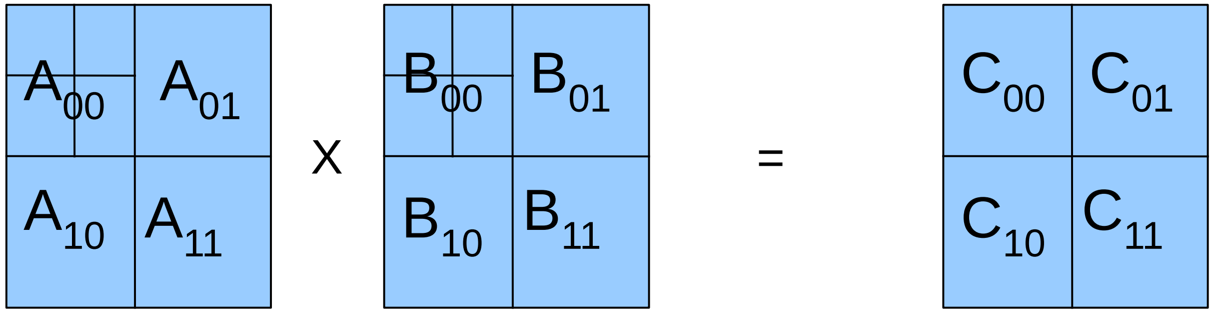
- Ignores architectural features such as:
  - memory bandwidth,
  - communication cost and latency,
  - scheduling, ...
- Hardware may be difficult to realize

Example 1 – Matrix Multiplication

- One of the fundamental parallel processing tasks.
- Applications to several important problems in linear algebra, signal processing and optimization.
- Several techniques that work in parallel also.

Example I – Matrix Multiplication

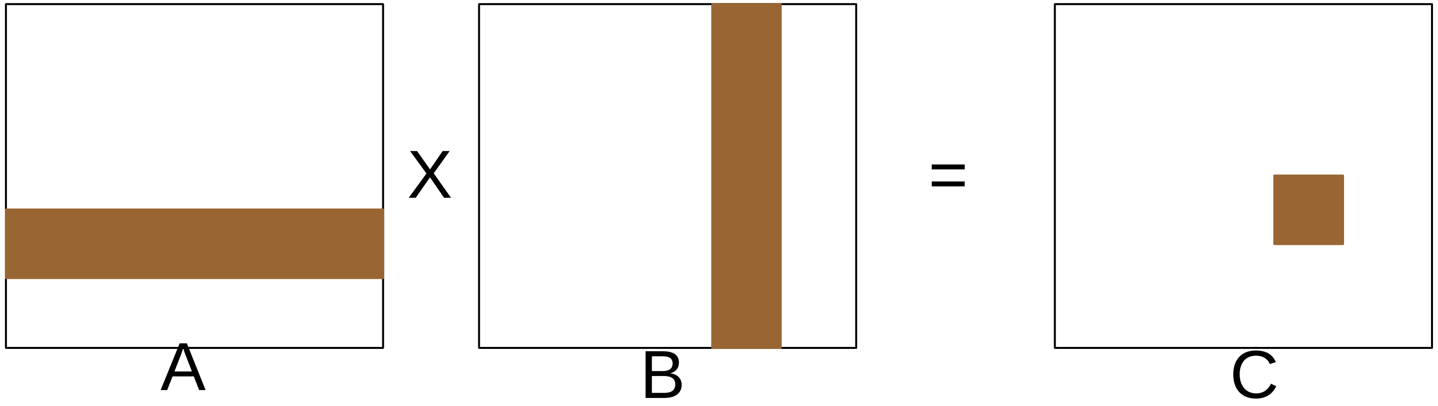
- Recall that in C = A x B, C[i,j] = Σ A[i,k].B[k,j].
- Consider the following recursive approach:
  - Works well in practice.



$$C_{00} = A_{00} \,.\, B_{00} + A_{01} \,.\, B_{10}$$
$$C_{01} = A_{00} \,.\, B_{01} + A_{01} \,.\, B_{11}$$
$$C_{10} = A_{10} \,.\, B_{00} + A_{11} \,.\, B_{10}$$
$$C_{11} = A_{10} \,.\, B_{01} + A_{11} \,.\, B_{11}$$

Example I – Matrix Multiplication

- Other approaches include Cannon's algorithm



- Can overlap computation with communication.
- Works well when the number of processors is more.

Example 2 – New Parallel Algorithm

```
Listing 1:
S(1) = A(1)
for i = 2 to n do
    S(i) = S(i-1) o A(i)
```

- Prefix Computations: Given an array A of n elements and an associative operation o, compute A(1) o A(2) o ... A(i) for each i.
- A very simple sequential algorithm exists for this problem.
- Many computations can be expressed in terms of prefix computations.

Parallel Prefix Computation

- The sequential algorithm in Listing 1 is not efficient in parallel.
  - In particular, has to wait for the output of S(i) to compute the output S(i+1).
- Need a new algorithm approach.
  - Balanced Binary Tree

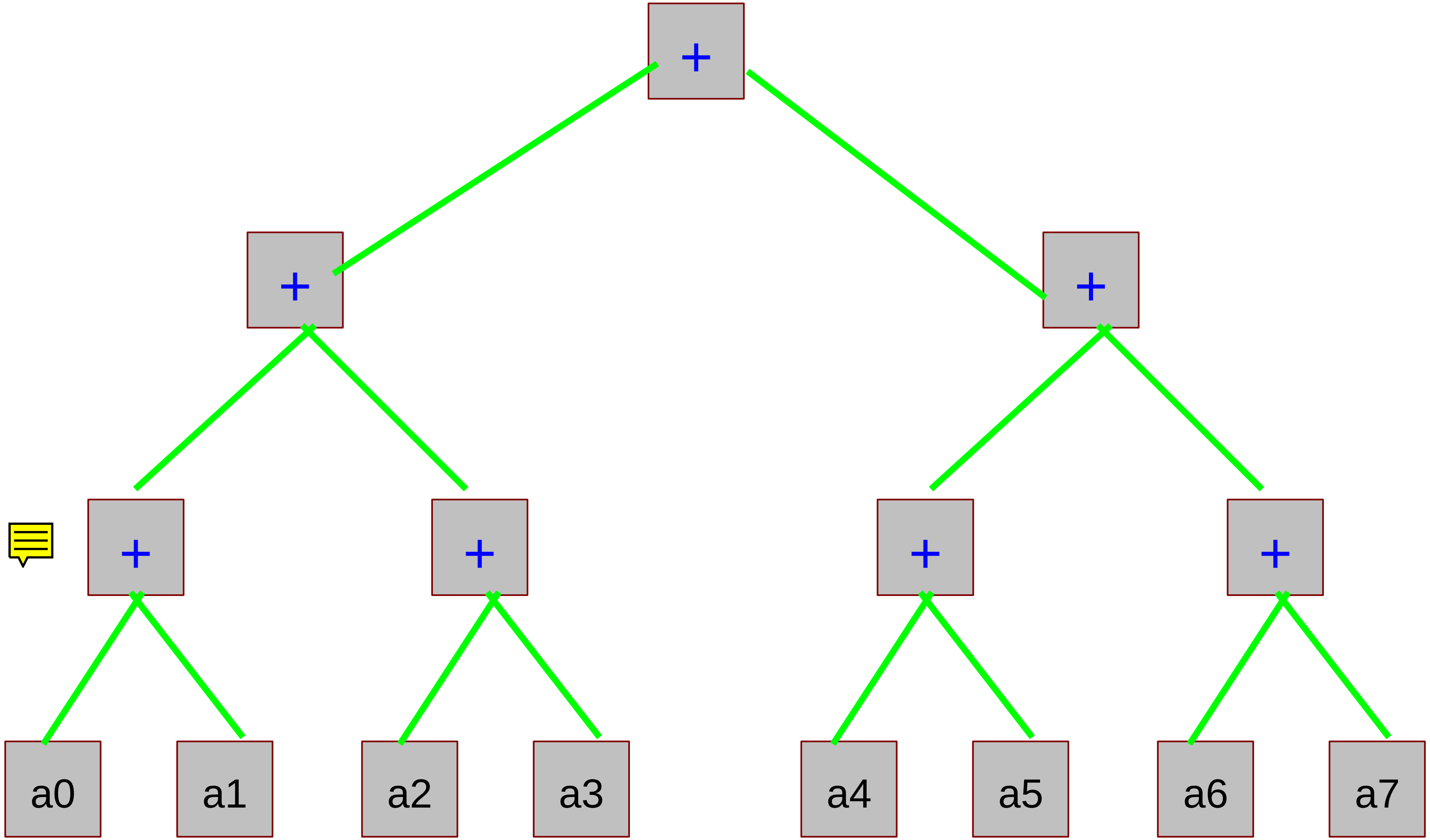
Balanced Binary Tree

- An algorithm design approach for parallel algorithms
- Many problems can be solved with this design technique.
- Easily amenable to parallelization and analysis.

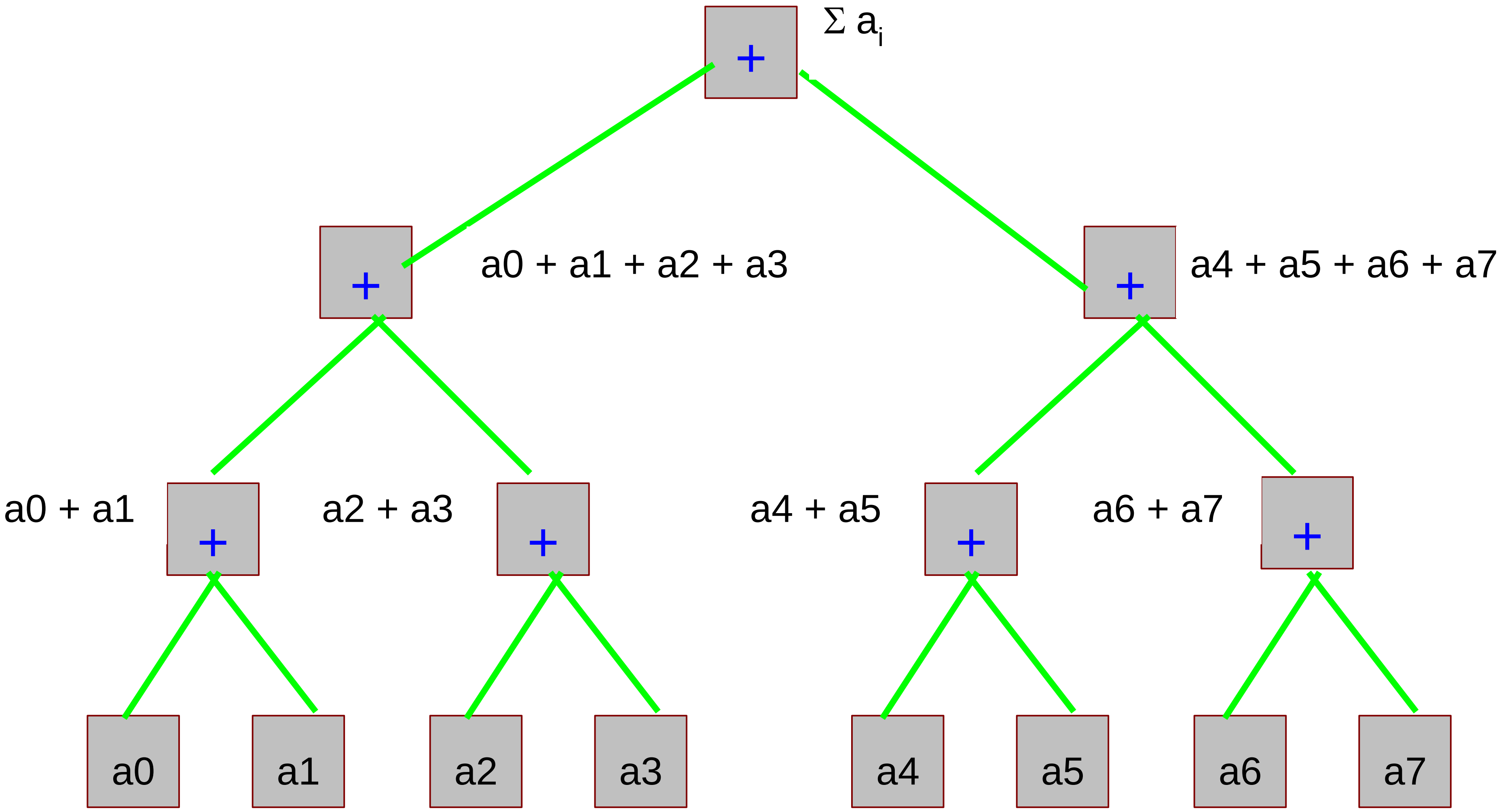
Balanced Binary Tree

- A complete binary tree with processors at each internal node.
- Input is at the leaf nodes
- Define operations to be executed at the internal nodes.
  - Inputs for this operation at a node are the values at the children of this node.
- Computation as a tree traversal from leaf to root.

Balanced Binary Tree – Prefix Sums



Balanced Binary Tree – Sum



Balanced Binary Tree – Sum

- The above approach called as an ``upward traversal''
  - Data flow from the children to the root.
  - Helpful in other situations also such as computing the max, expression evaluation.
- Analogously, can define a downward traversal
  - Data flows from root to leaf
  - Helps in settings such as element broadcast

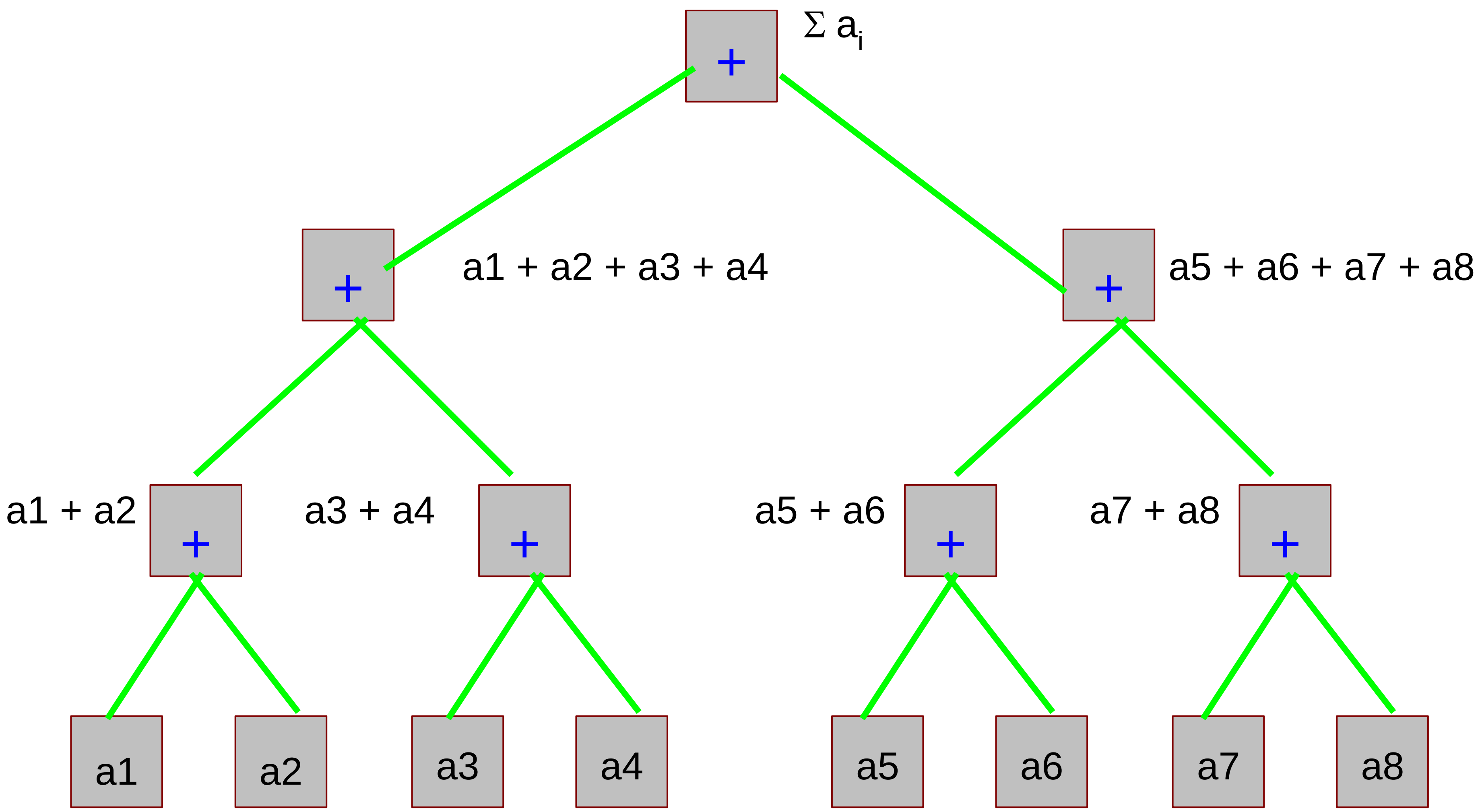
Balanced Binary Tree

- Can use a combination of both upward and downward traversal.
- Prefix computation requires that.
- Illustration in the next slide.

A : 4 2 -1 3 0 5

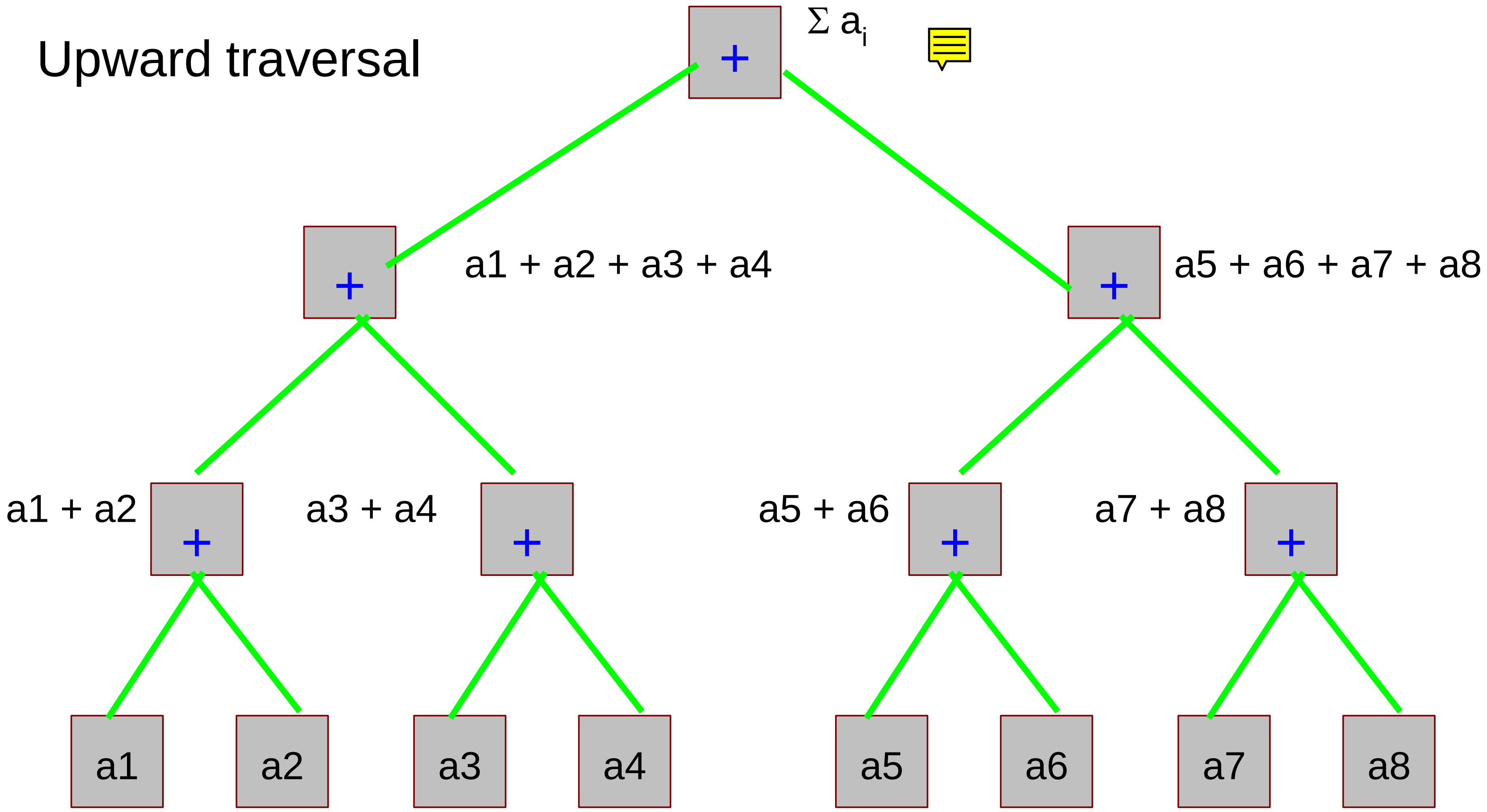
PrefixSum(A) : 4 6 5 8 8 13

Balanced Binary Tree – Sum



Balanced Binary Tree – Prefix Sum

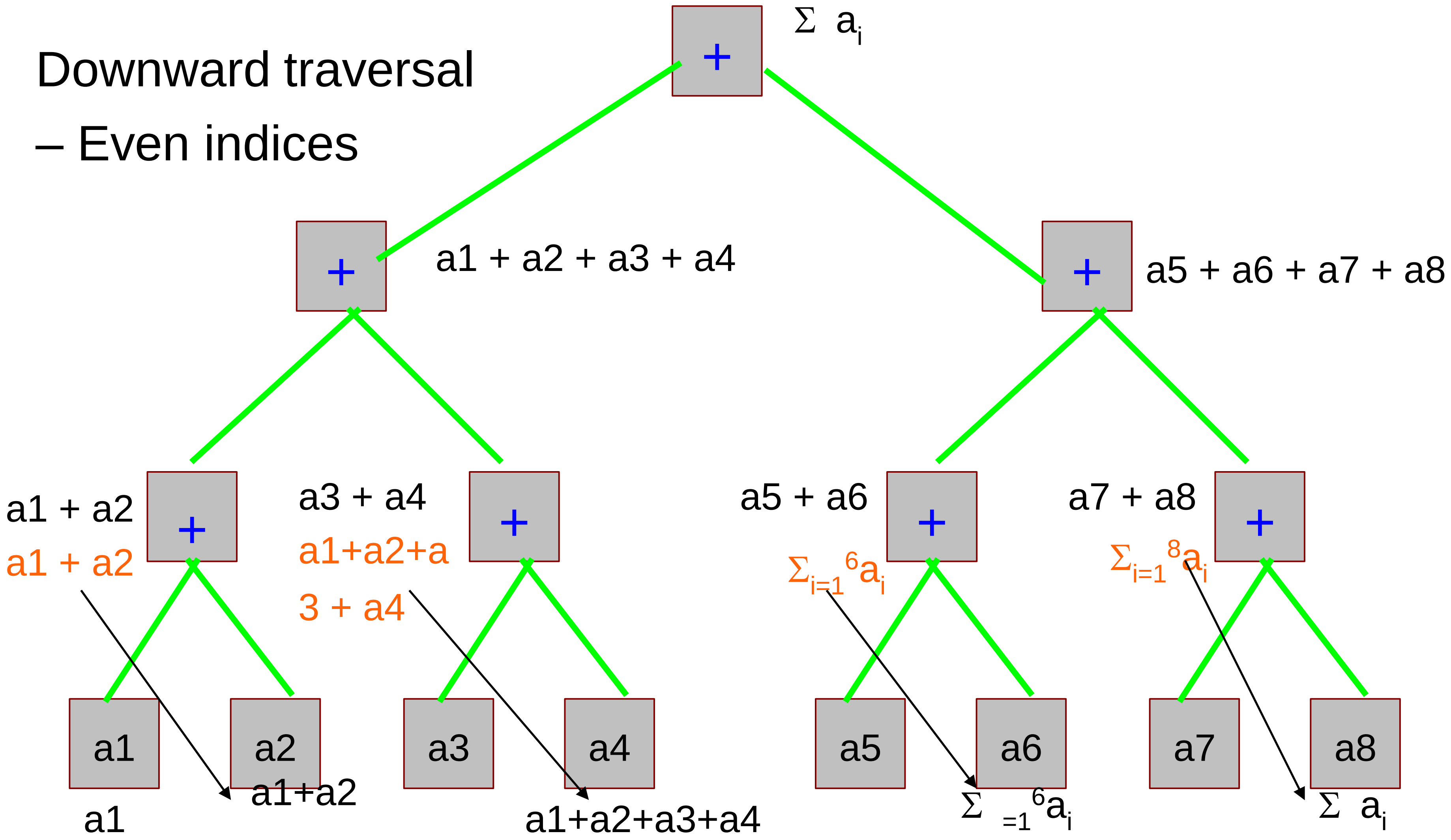
Upward traversal



Balanced Binary Tree – Prefix Sum

Downward traversal

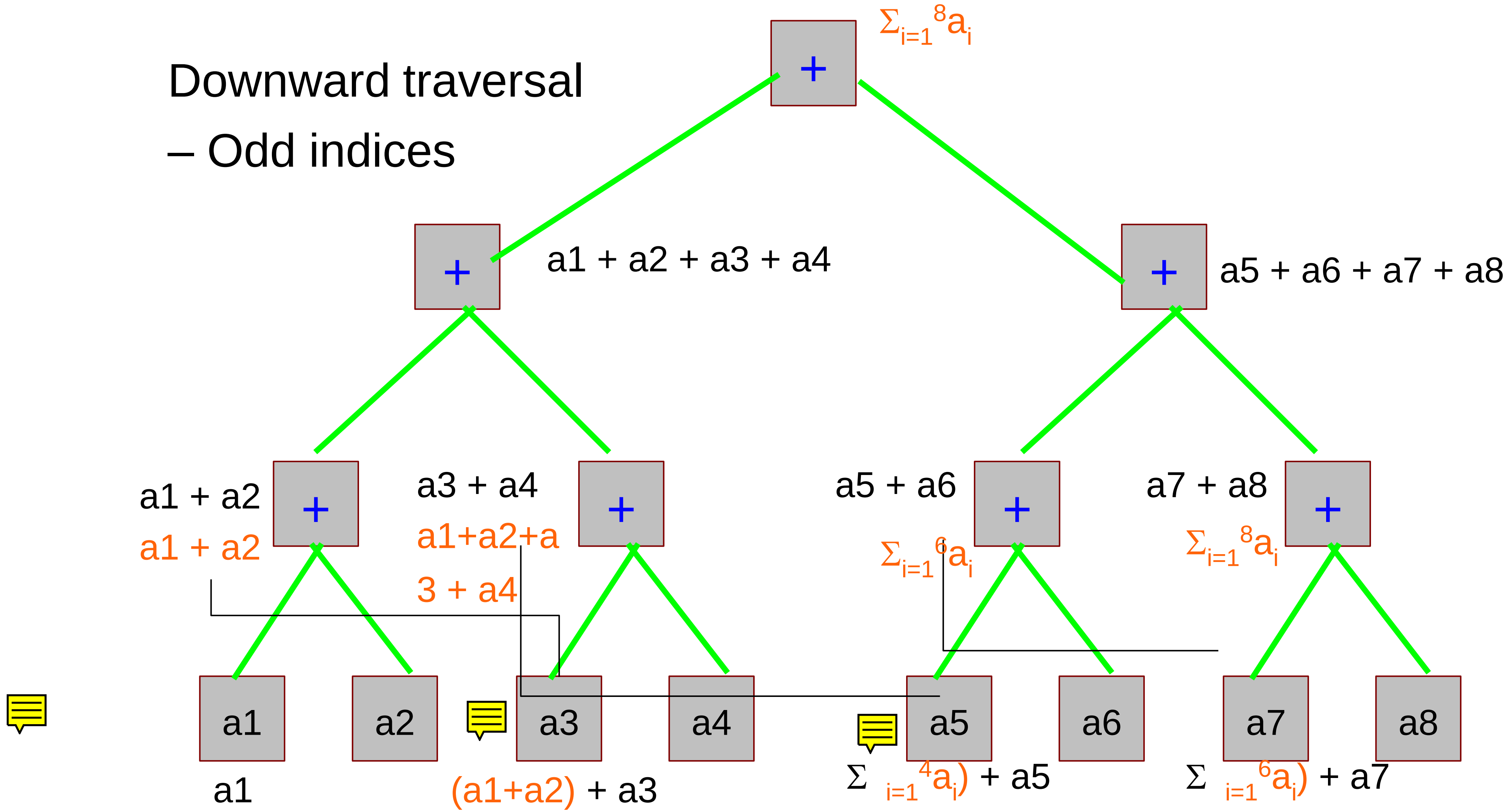
– Even indices



Balanced Binary Tree – Prefix Sum

Downward traversal

– Odd indices



Balanced Binary Tree – Prefix Sums

- Two traversals of a complete binary tree.
- The tree is only a visual aid.
  - Map processors to locations in the tree
  - Perform equivalent computations.
  - Algorithm designed in the PRAM model.
  - Works in logarithmic time, and optimal number of operations.

//upward traversal

1. for $i = 1$ to $n/2$ do in parallel

   $b_i = a_{2i-2} \text{ o } a_{2i}$

2. Recursively compute the prefix sums of $B= (b_1, b_2, \ldots, b_{n/2})$ and store them in $C = (c_1, c_2, \ldots, c_{n/2})$

//downward traversal

3. for $i = 1$ to n do in parallel

   i is even : $s_i = c_i$

   i= 1 : $s_1 = c_1$

   i is odd : $s_i = c_{(i-1)/2} \text{ o } a_i$

Analysis of Parallel Algorithms

- To analyze parallel algorithms, we rely on asymptotics and recurrences.
- Each operation costs 1 unit, only sequential time needs to be counted. We assume as many processors as can be used are available.
- In the prefix sum example, let T(n) be the time in parallel for an input of size n.
  - Step 1 can use n/2 processors in parallel each taking 1 unit of time.
  - Step 2 is a recursive call and takes T(n/2) time.
  - Step 3 uses n processors each taking 1 unit of time.

Analysis of Parallel Algorithms

- The recurrence relation is:
  - $T(n) = T(n/2) + O(1)$
  - Can ignore effects due to constant factors, such as the difference in the number of processors between steps 1 and 3.
- The solution to the above recurrence is $T(n) = O(\log n)$.

- Another parameter of interest in parallel algorithms is the work done.
- Can be stated as the sum of the works done by each of the processors.

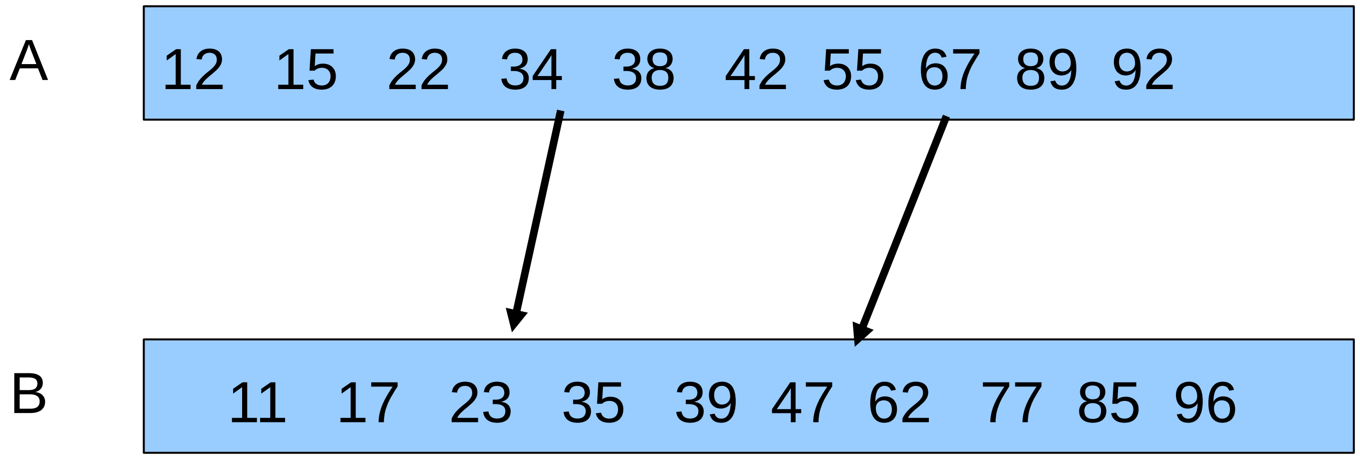
Analysis of Parallel Algorithms

- The work done by the prefix algorithm can be expressed by the recurrence
  - $W(n) = W(n/2) + O(n)$.
  - The $O(n)$ accounts for the work in the first and the third steps.
  - Solution: $W(n) = O(n)$.
- Work done can indicate if the algorithm is doing about the same amount of operations as the best known sequential algorithm.
- Such a parallel algorithm is called an optimal algorithm.

Other Design Paradigms

- Partitioning
  - Similar to divide and conquer
  - But no need to combine solutions
  - Can treat problems independently and solve in parallel.
  - Example: Parallel merging, searching.

Merging in Parallel by Partitioning



- Two sorted arrays A and B to be merged into C.
- Let A be a sorted array. Let Rank(x, A) be the number of elements smaller than x in A.
- Claim: Rank(x, C) = Rank(x, A) + Rank(x, B)
- For x in A, Rank(x,A) is immediately available. To find Rank(x, B) can use binary search in parallel.

Quick Example

A = [8 10 12 24]

B = [15 17 27 32]

Element	8	10	12	24	15	17	27	32
Rank in A	0	1	2	3	3	3	4	4
Rank in B	0	0	0	2	0	1	2	3
Rank in C	0	1	2	5	3	4	6	7

C = [8 10 12 15 17 24 27 32]

Merging in Parallel by Partitioning

- Time for each binary search is O(log n)
- Total time for merging = O(log n), the total work is O(n log n).
  - Not work optimal as compared to the best possible sequential time complexity of O(n).
- Can reduce the total work to O(n).
  - Induce equal-sized partitions in the arrays
  - Rank one element, say the first element, from each partition
  - Use these ranks to find the ranks of the other elements, sequentially.

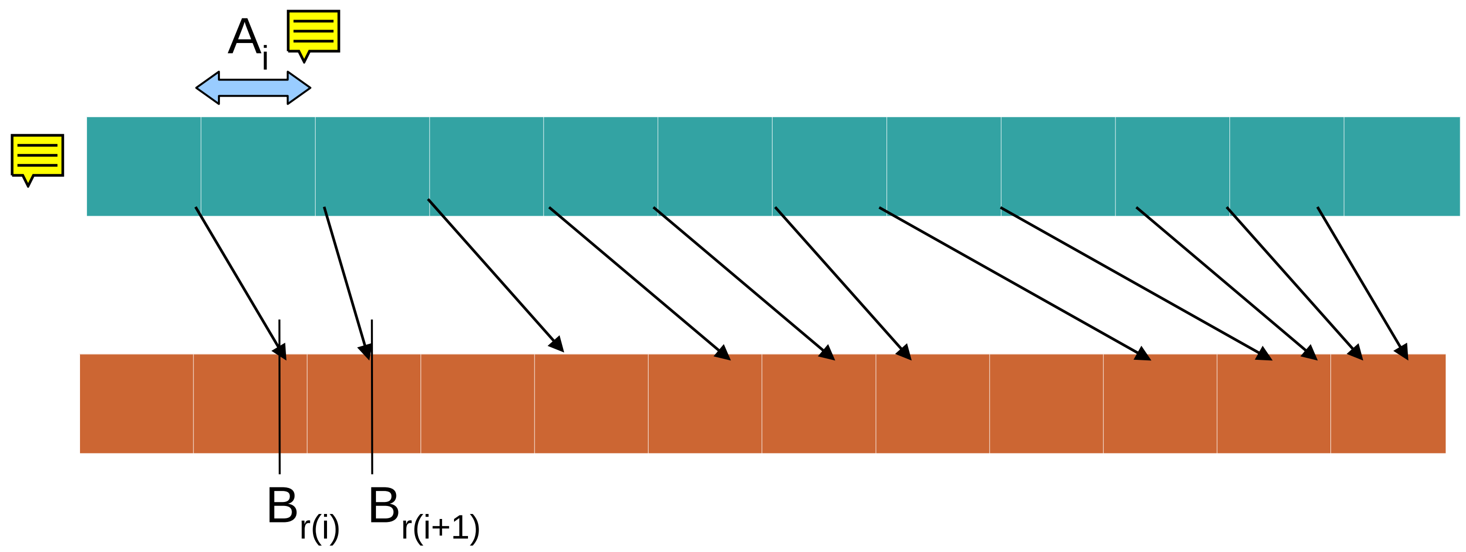
An Improved Optimal Algorithm

- General technique
  - Solve a smaller problem in parallel
  - Extend the solution to the entire problem.
- For the first step, the problem size to be solved is guided by the factor of non-optimality of an existing parallel algorithm.

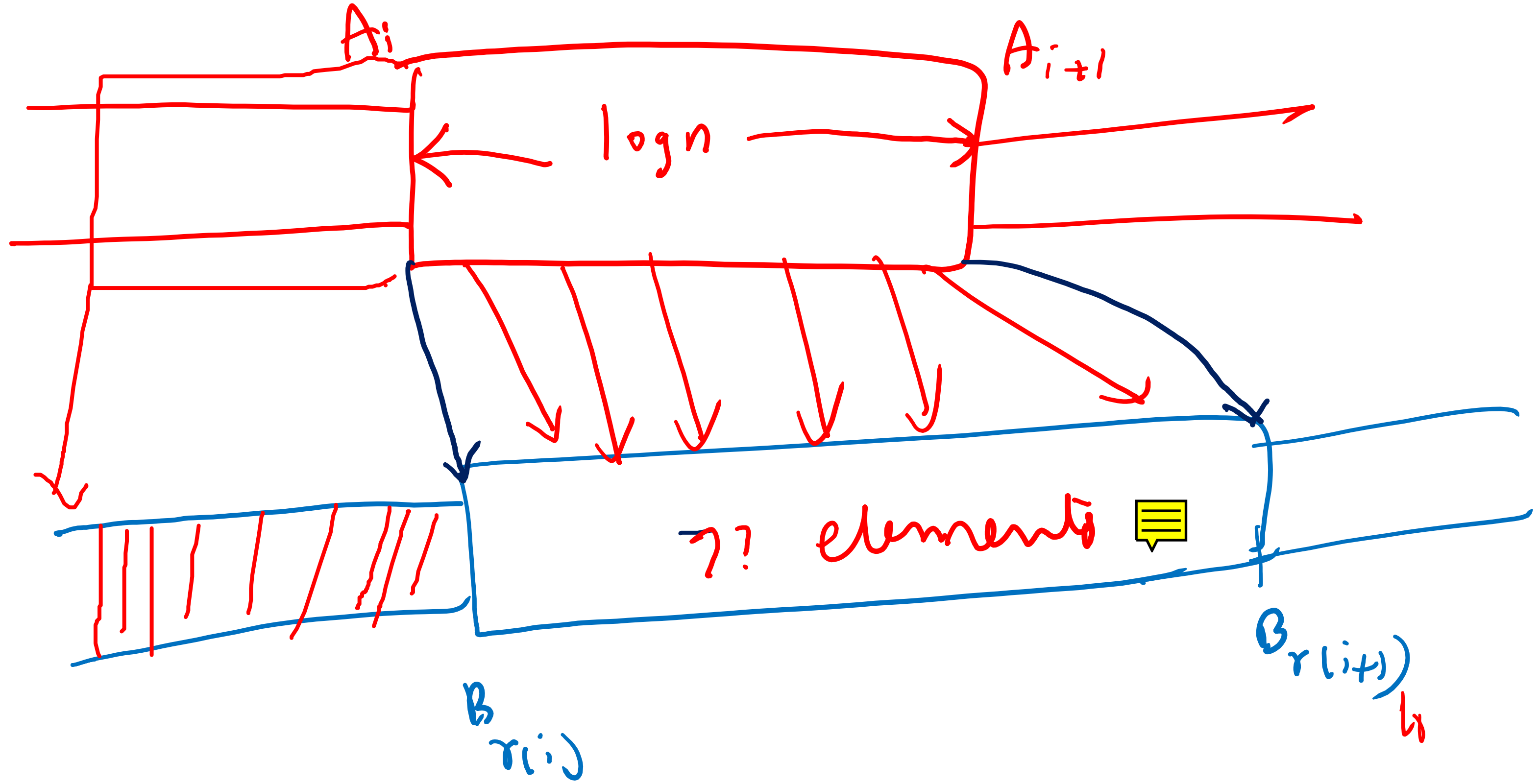
An Improved Parallel Algorithm

- Our simple parallel algorithm is away from work optimality by a factor of O(log n).
- So, we should solve a problem of size O(n/log n).
- For this purpose, we pick every log n^{th} element of A, and similarly in B.
- Use the simple parallel algorithm on these elements of A and B.
  - Binary search however in the entire A and B.

An Improved Parallel Algorithm



- Let $A_1, A_2, \ldots, A_{n/\log n}$ be the elements of A ranked in B.
- These ranks induce partitions in B.
  - Define $[B_{r(i)} \ldots B_{r(i+1)}]$ as the portion of B so that $[A(i) \ldots A(i+1)]$ have ranks in.
- Can therefore merge $[A(i) \ldots A(i+1)]$ with $[B_{r(i)} \ldots B_{r(i+1)}]$ sequentially.



A_i
A_{i+1}
log n
?? elements
$B_{r(i)}$
$B_{r(i+1)}$

An Improved Parallel Algorithm

- Such sequential merges can happen in parallel, at each index of A[i].
- Time taken for the sequential merge is O(log n + $B_{r(i+1)} - B_{r(i)}$).
- Time:
  - Binary search: O(log n), with n/log n processors.
  - Sequential merge: O(log n), subject to certain conditions. There are also n/log n such merges in parallel.
- Work:
  - There are n/log n binary searches in parallel. Work = O(n).
  - For the sequential merges too, work = O(n).