Complexity and Advanced Algorithms

Module 2

Parallel Computing

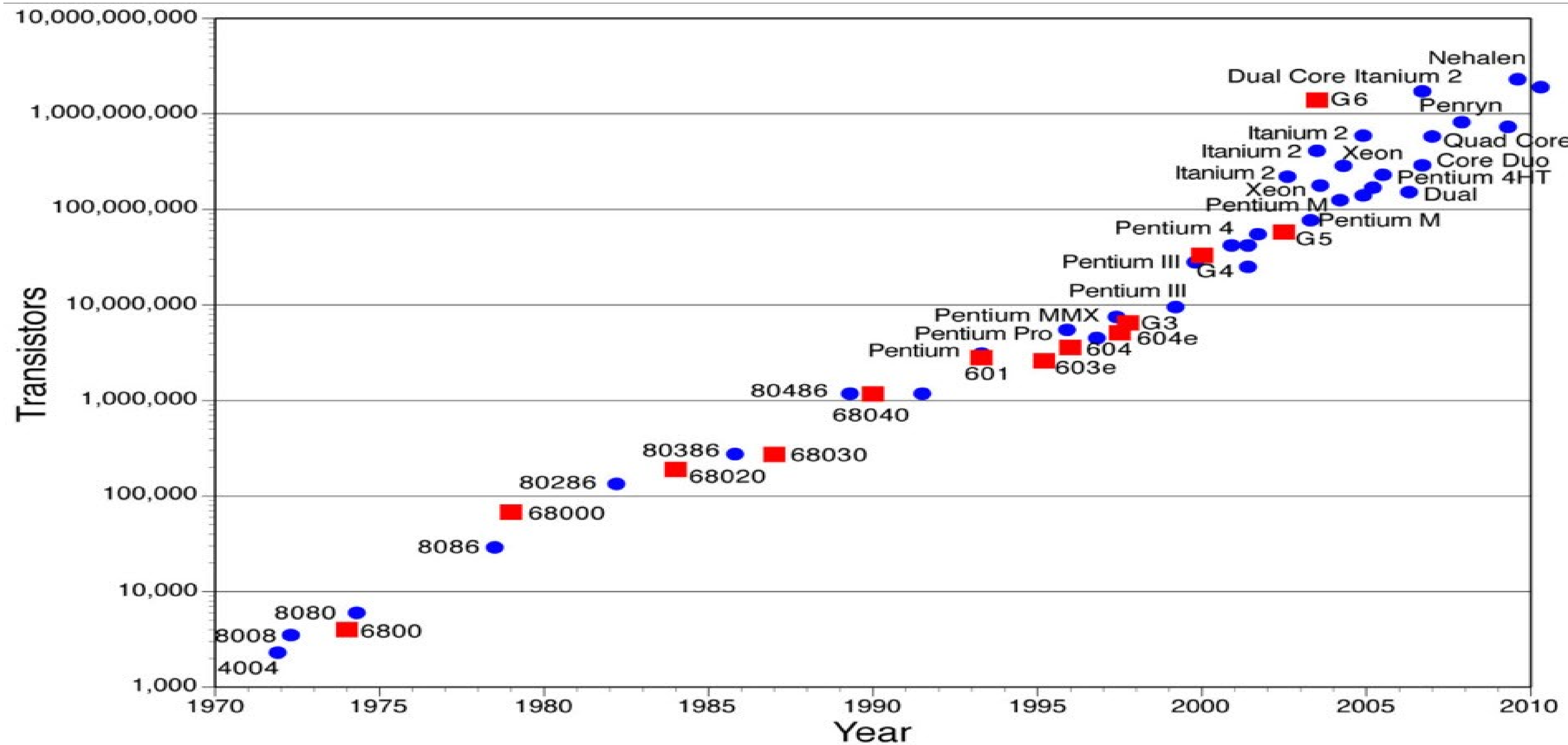
Why Parallel Computing?

- 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.

1. Application Requirements

- Several applications are pushing the limits with huge compute requirements:
  - Deep learning
  - Image/Video/Text search, retrieval, and indexing
  - Digital effects/computer graphics/animation
  - Materials/Life Sciences/Drug design/…
  - Social computing/web/...

2. Architectural Advances



- Moore's Law: The number of transistors that can be inexpensively placed on an integrated circuit is increasing exponentially, doubling approximately every two years.

On the Other Hand...

- Present Difficulties
  - Memory Wall
  - Power Wall
  - ILP Wall

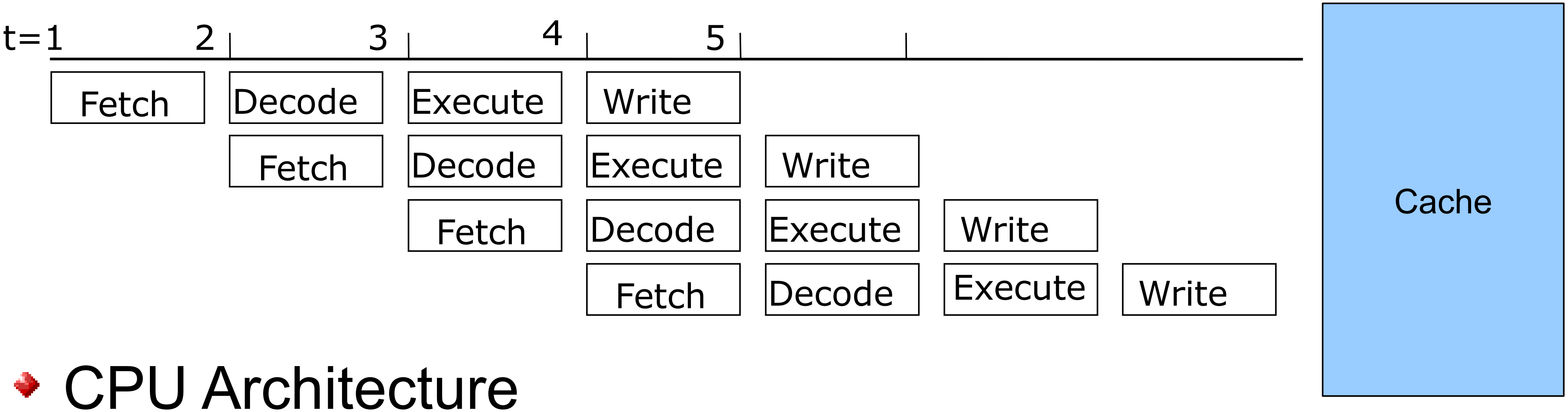
The Brick Wall - 1

- Memory Wall
  - Memory latency up to 200 cycles per load/store.
  - Floating point operations take no more than 4 cycles.
  - Earlier, it was thought that “multiply is slow but load and store is fast”.

The Brick Wall - 2

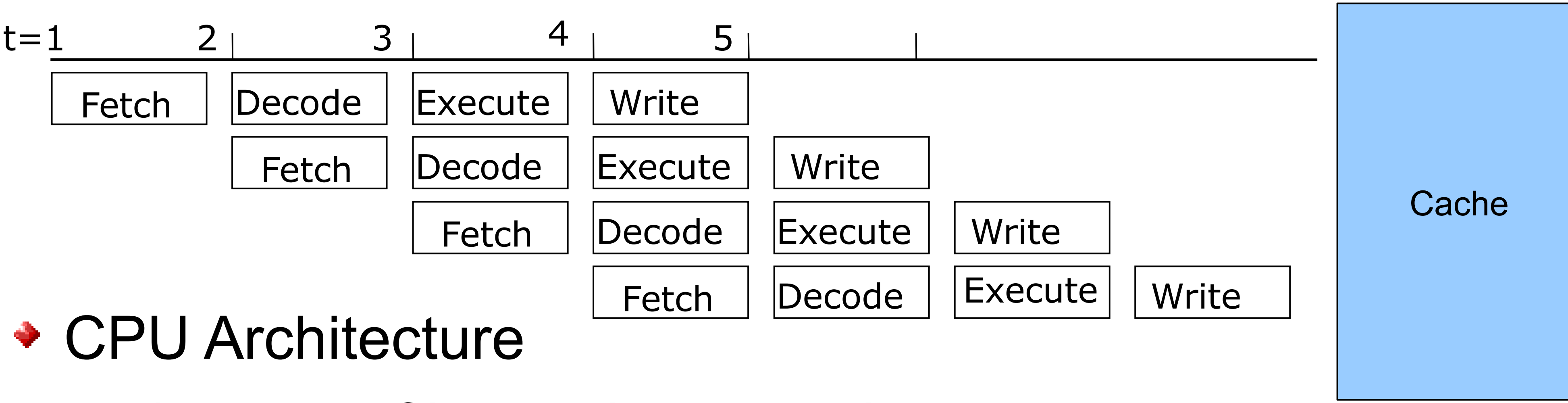
- Power Wall
  - Enormous increase in power consumption.
  - Power leakage.
  - However, presently “Power is expensive but transistors are free”.

Basic Architecture Concepts



- CPU Architecture
  - 4 stages of instruction execution
    - Too many cycles per instruction (CPI)
  - To reduce the CPI, introduce pipelined execution
    - Needs buffers to store results across stages.
    - A cache to handle slow memory access times

Basic Architecture Concepts



- CPU Architecture
  - 4 stages of instruction execution
    - Too many cycles per instruction (CPI)
  - To reduce the CPI, introduce pipelined execution
    - Needs buffers to store results across stages.
    - A cache to handle slow memory access times
    - Caches, out-of-order execution, branch prediction, …

The Brick Wall - 3

- ILP Wall
  - ILP via branch prediction, out-of-order and speculative execution
  - Diminishing returns from instruction level parallelism.

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?

Multicores to the Rescue

- Predicted that 100+ core computers would be a reality soon.
- Increased number of cores without significant improvement in clock rates.
  - —Due to silicon technology improvements
- Big questions
  - —How to exploit these cores in parallel?
  - —What are the killer applications that can democratize these new models?
    - Search, web, ???

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?

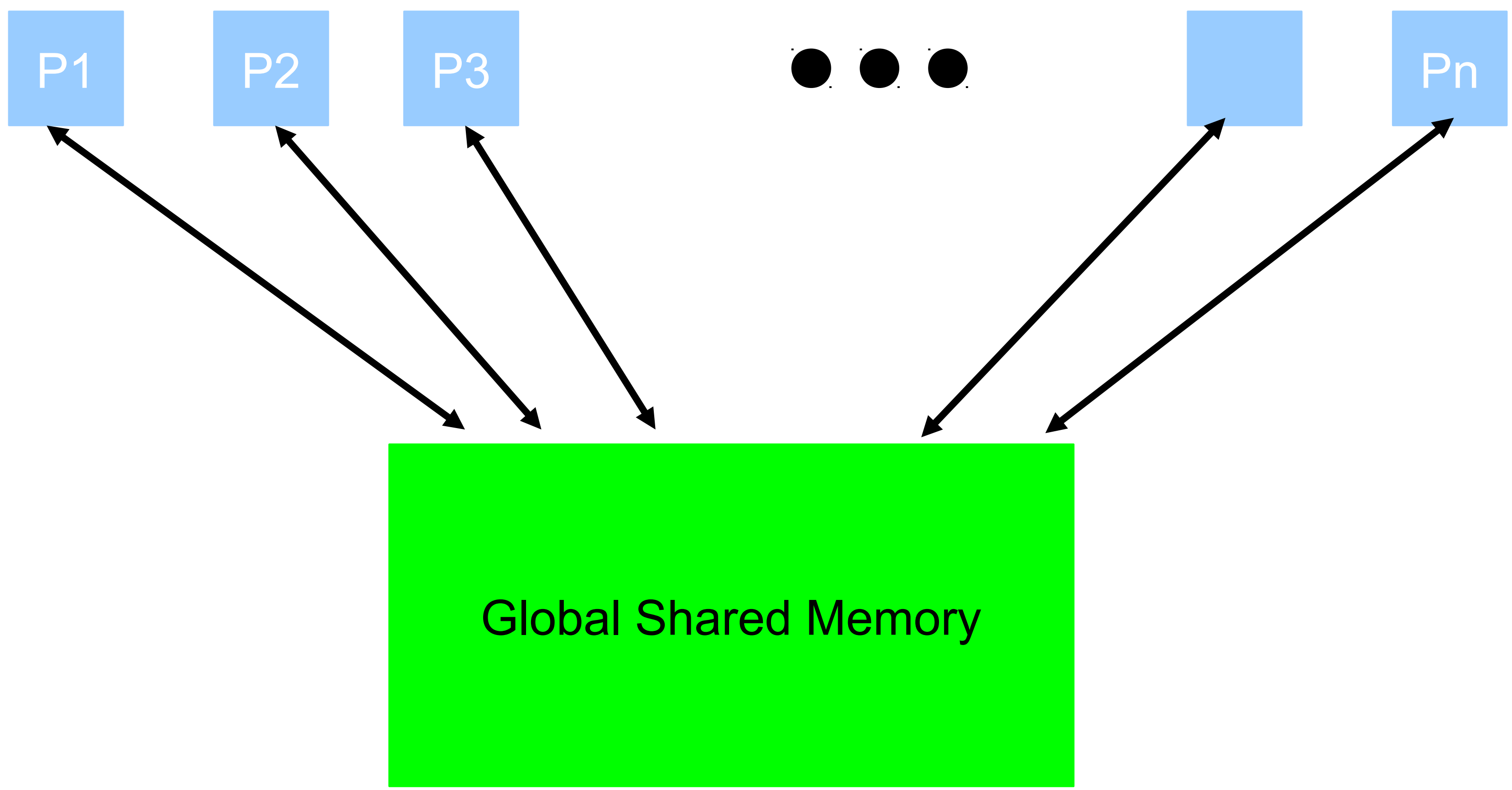
The Academic Interest

- 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
- Some parallel programming practice
- Complexity, characterization, and connection to sequential complexity classes.

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, $C[i,j] = \Sigma A[i,k].B[k,j]$.
- Consider the following recursive approach:
  - Works well in practice.

$$\begin{bmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{bmatrix} \times \begin{bmatrix} B_{00} & B_{01} \\ B_{10} & B_{11} \end{bmatrix} = \begin{bmatrix} C_{00} & C_{01} \\ C_{10} & C_{11} \end{bmatrix}$$

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

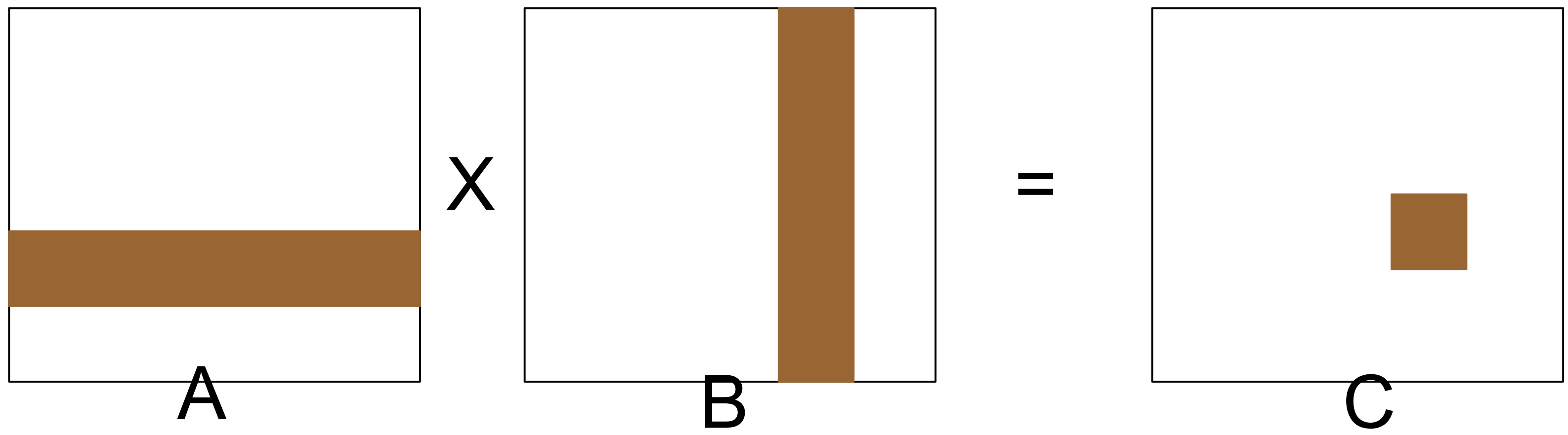
$$C_{01} = A_{00} . B_{01} + A_{01} . B_{11}$$

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

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

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.

Parallel Prefix Computation

- The sequential algorithm in Listing 1 is not efficient in parallel.
- Need a new algorithm approach.
  - Balanced Binary Tree

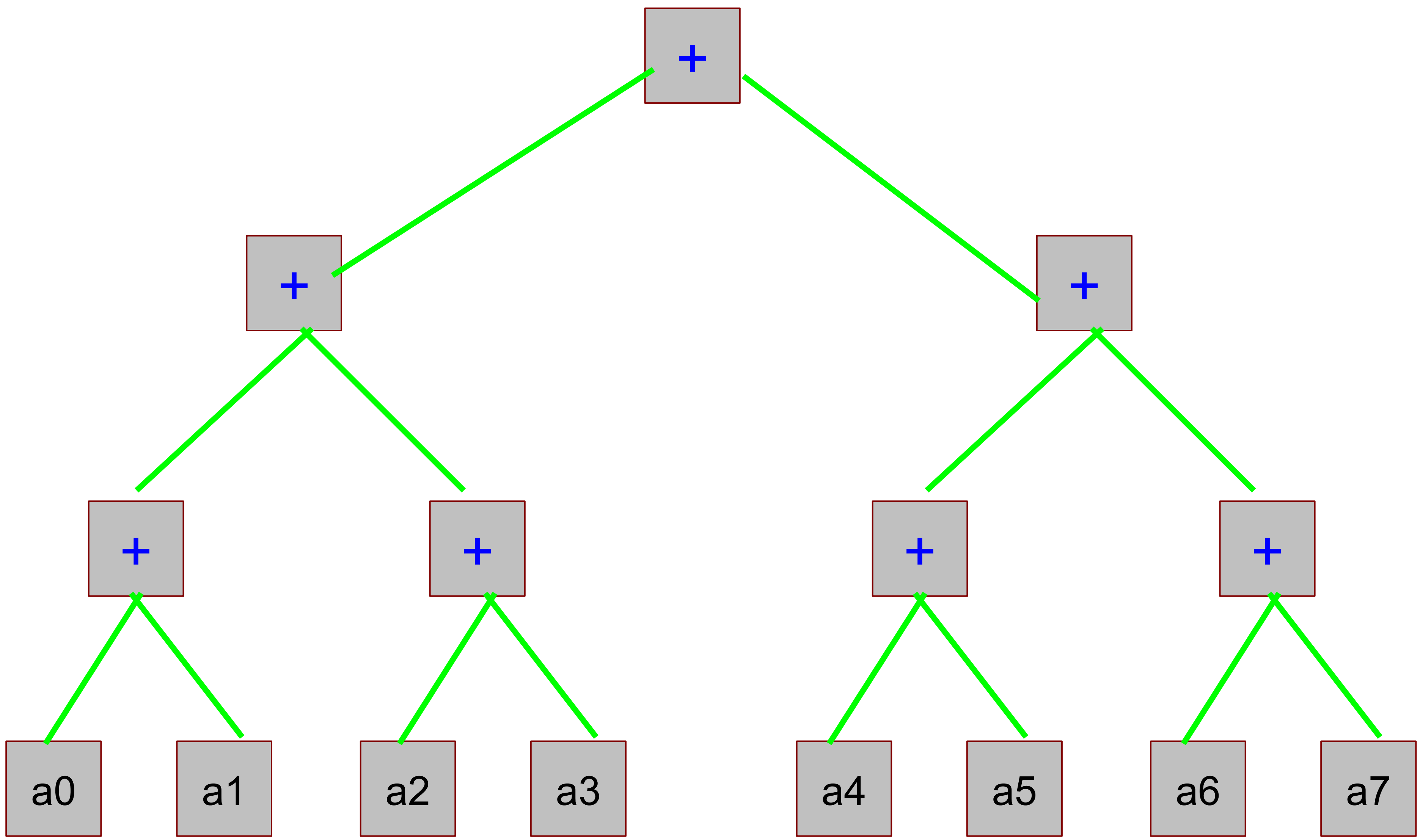
Balanced Binary Tree

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

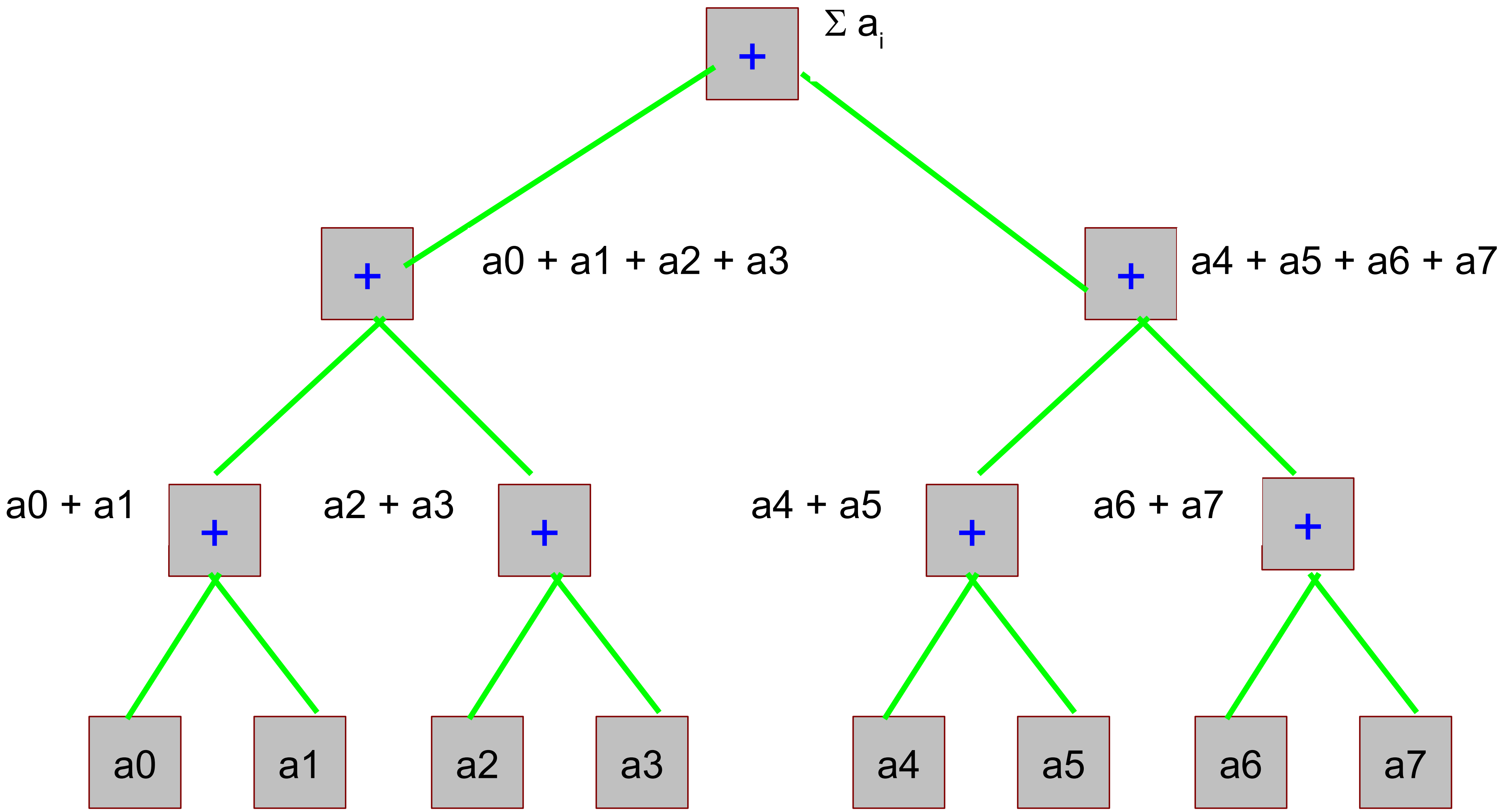
Balanced Binary Tree

- 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



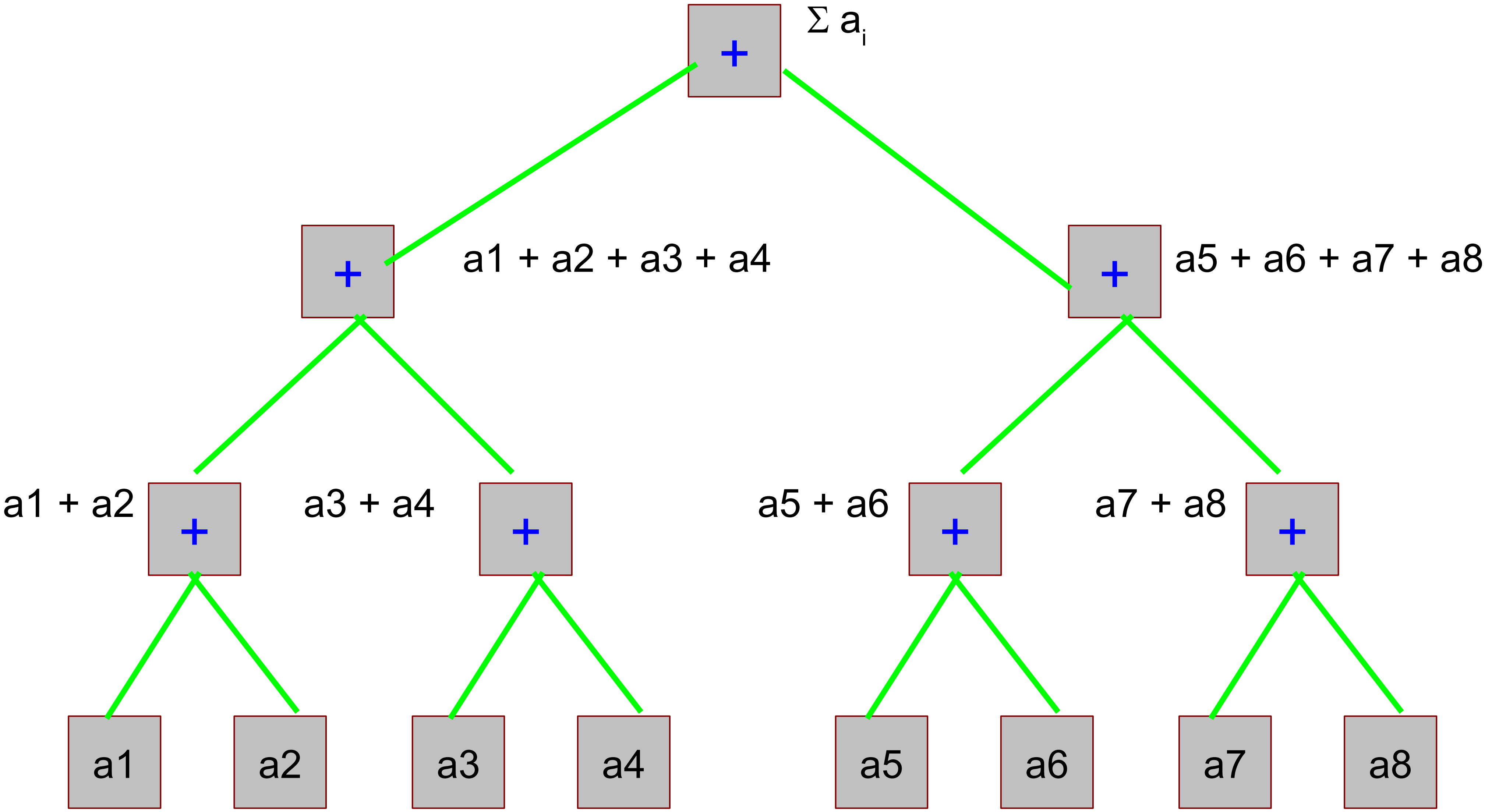
Balanced Binary Tree – Sum

- 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

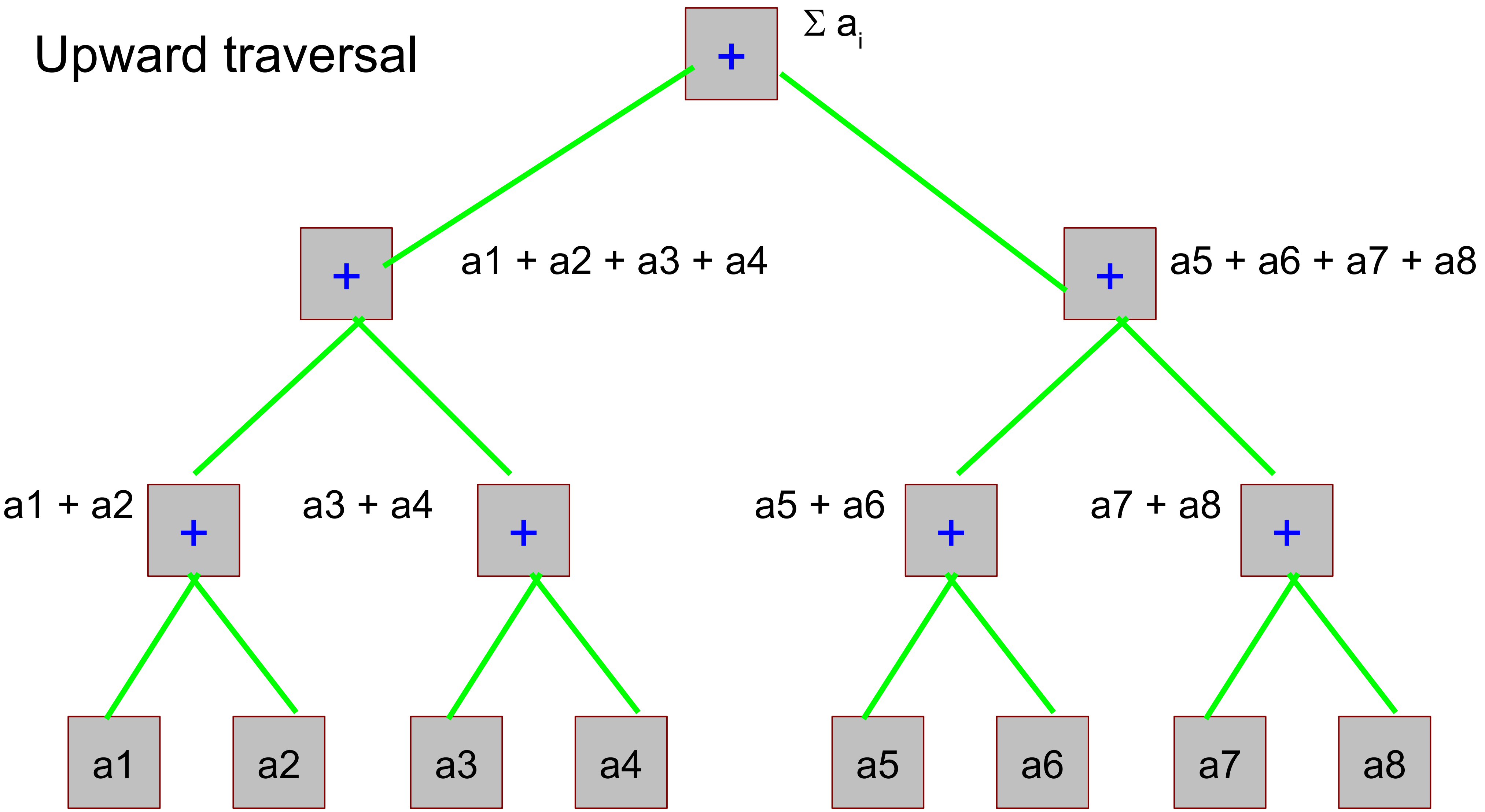
- Can use a combination of both upward and downward traversal.
- Prefix computation requires that.
- Illustration in the next slide.

Balanced Binary Tree – Sum



Balanced Binary Tree – Prefix Sum

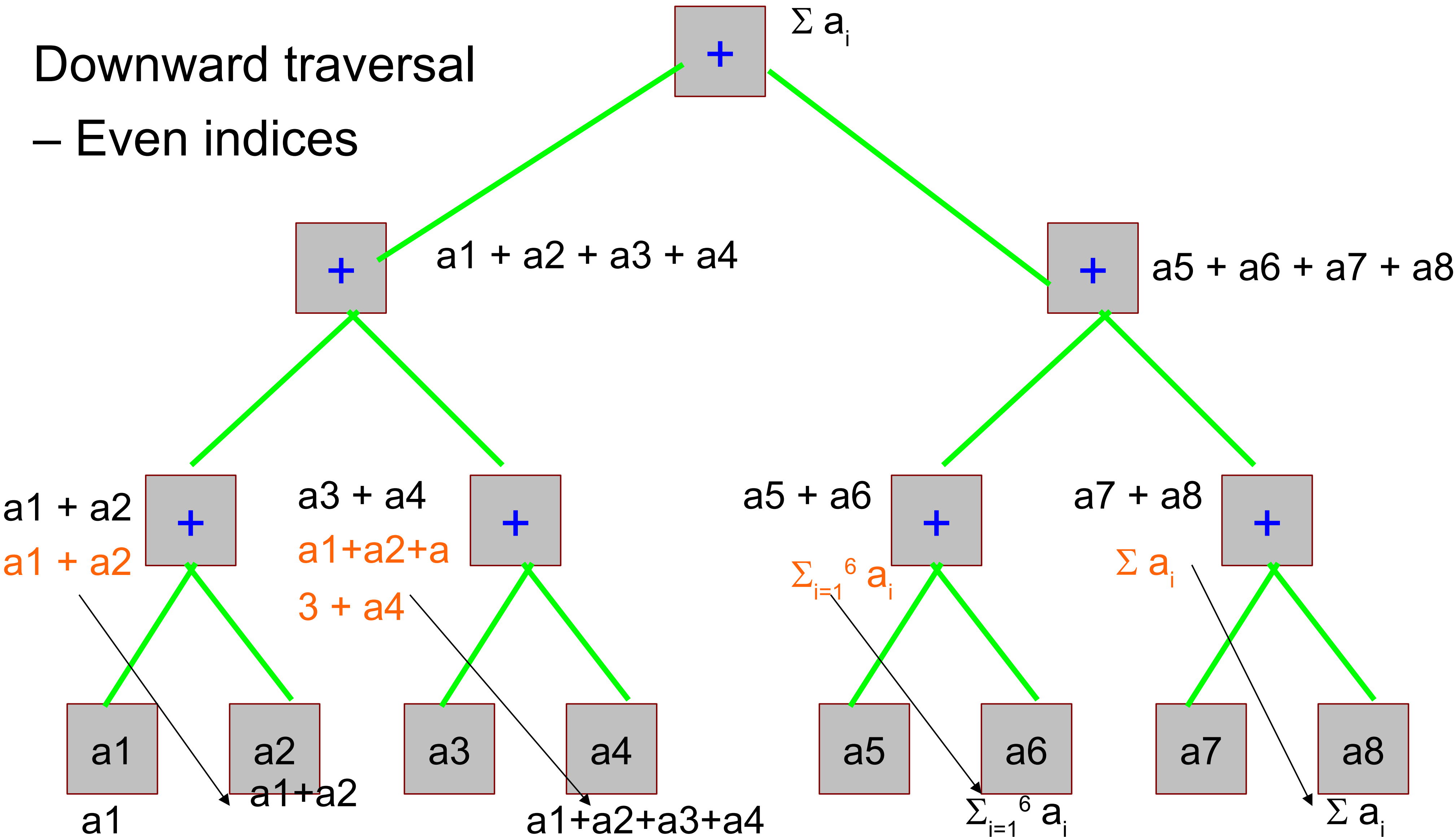
Upward traversal



Balanced Binary Tree – Prefix Sum

Downward traversal

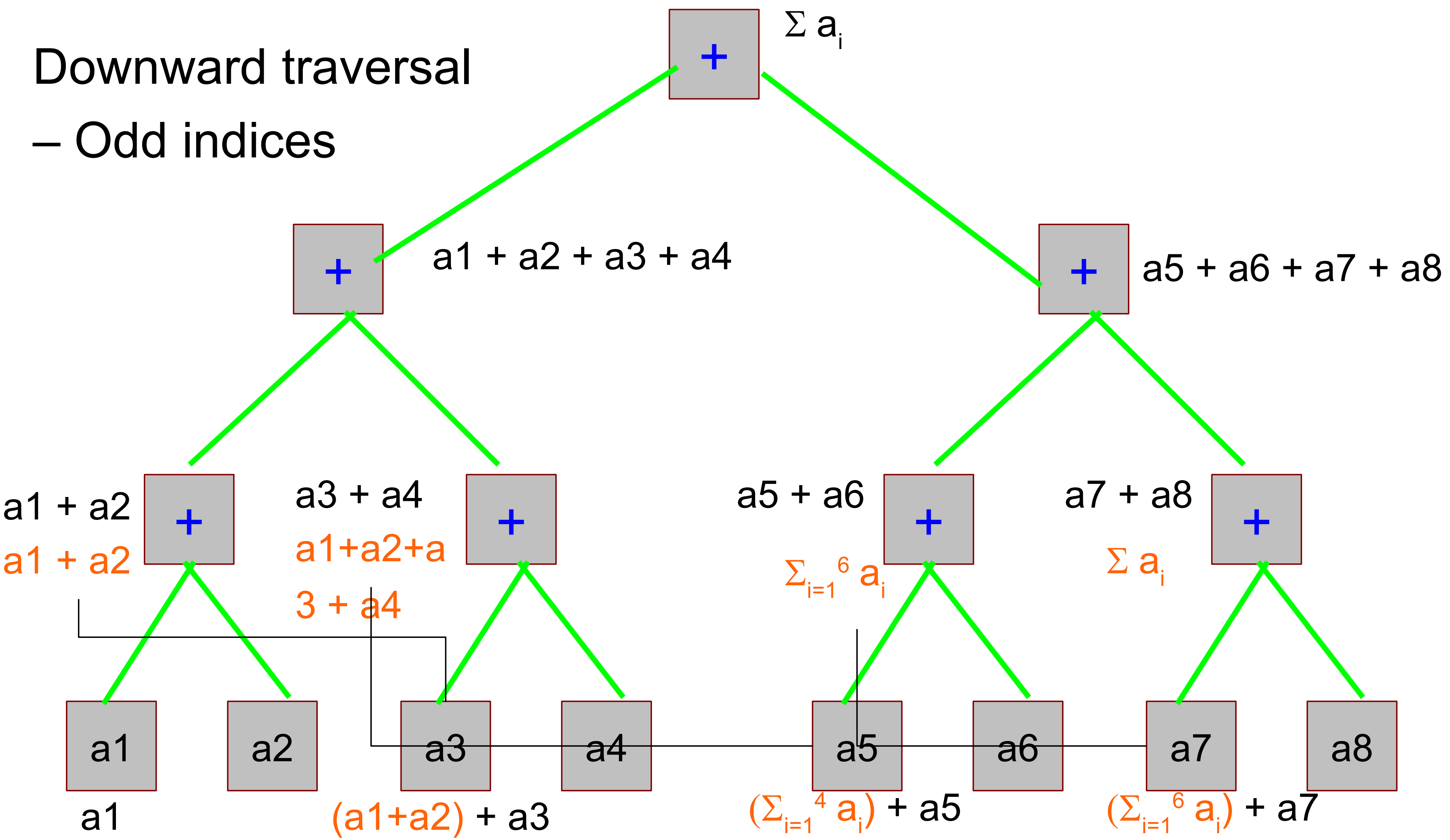
– Even indices



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.

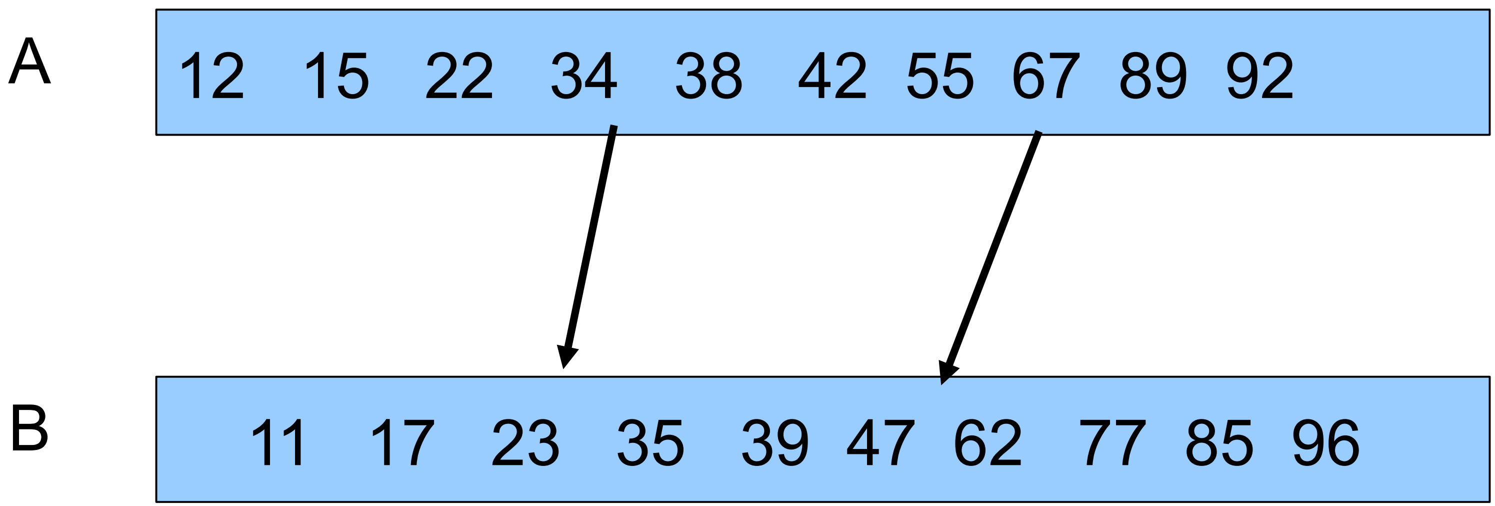
Analysis of Parallel Algorithms

- 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.
- 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).
  - Non optimal as compared to sequential time complexity of O(n).
- Can reduce the total work to O(n).
  - Induce partitions in the arrays of equal size
  - Rank one element from each partition
  - Use these ranks to find the ranks of the other elements, sequentially.

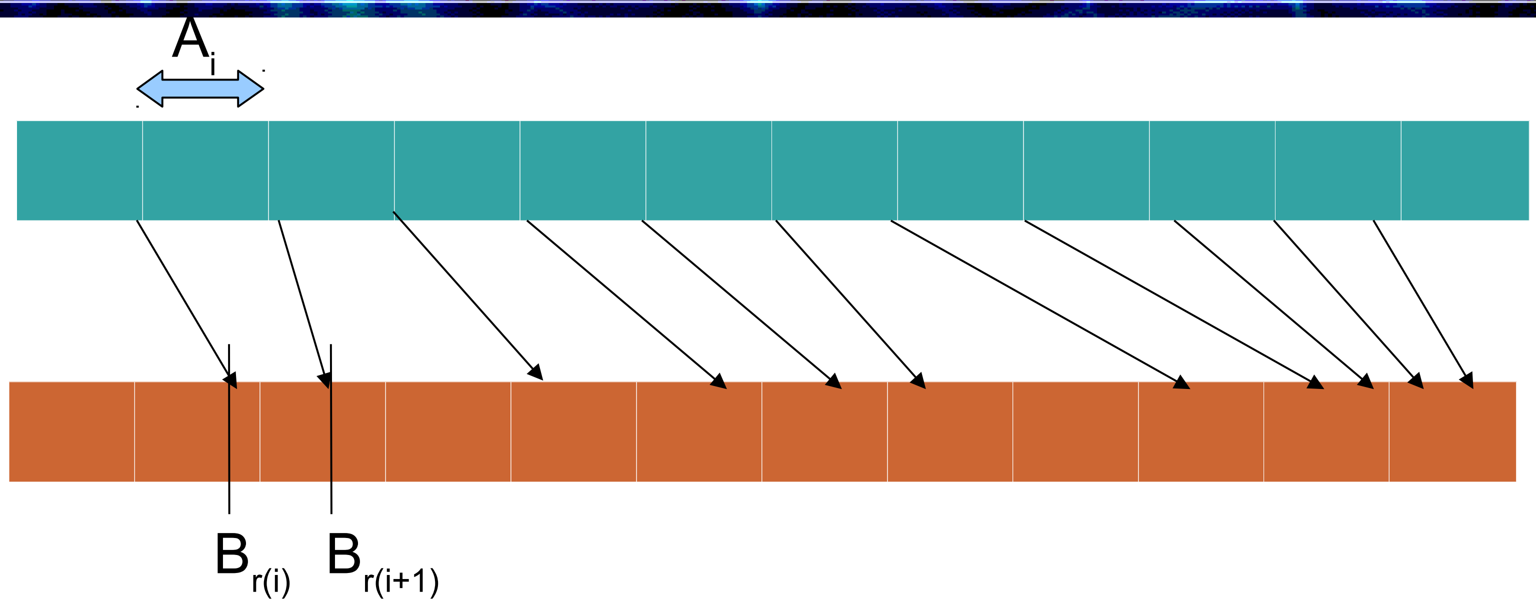
An Improved Optimal Algorithm

- 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 factor of an existing parallel algorithm.

An Improved Parallel Algorithm

- Our simple parallel algorithm is away from 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

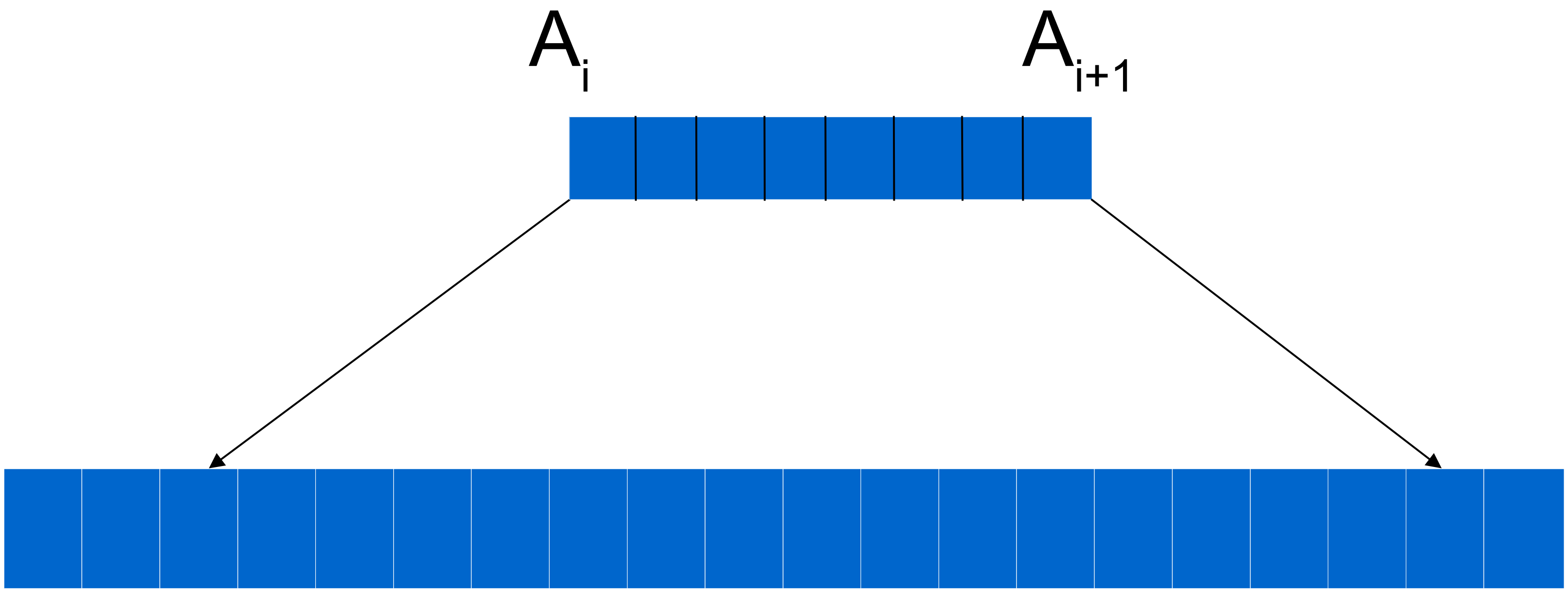


- Let A_1, A_2,…,$A_{n/\log n}$ be the elements of A ranked in B.
- These ranks induce partitions in B.
  - Define $[B_{r(i)}\ldots B_{r(i+1)}]$ as the portion of B so that $[A(i)\ldots A(i+1)]$ have ranks in.
- Can therefore merge $[A(i)\ldots A(i+1)]$ with $[B_{r(i)}\ldots B_{r(i+1)}]$ sequentially.

An Improved Parallel Algorithm

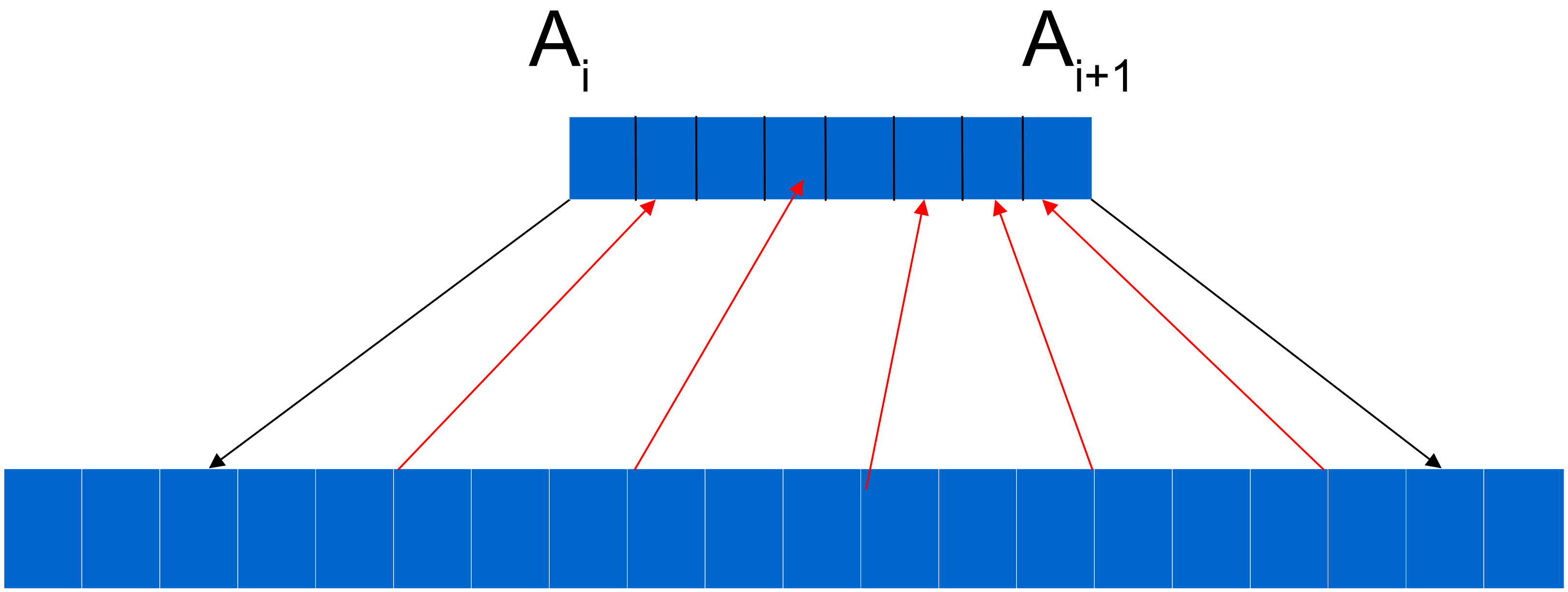
- Such sequential merges can happen in parallel, at each index of A[i].
- Time taken for the sequential merge is $O(\log n + B_{r(i+1)} - B_{r(i)})$.
- Time:
  - Binary search: O(log n), with n/log n processors.
  - Sequential merge: O(log n), subject to certain conditions. There are also n/log n such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

An Improved Parallel Algorithm



- What if $[B_{r(i)}...B_{r(i+1)}]$ has a size of more than log n?
- The situation can be addressed
  - Pick equally spaced, no more than log n, spaced items in $[B_{r(i)}...B_{r(i+1)}]$.
  - Rank these in $[A_i...A_{i+1}]$.

An Improved Parallel Algorithm



- What if $[B_{r(i)} \ldots B_{r(i+1)}]$ has a size of more than log n?
- The situation can be addressed
  - Pick equally spaced, no more than log n, spaced items in $[B_{r(i)} \ldots B_{r(i+1)}]$.
  - Rank these in $[A_i \ldots A_{i+1}]$.

Final Result

- Can merge two sorted arrays of size n in time O(log n) with work O(n).
  - Need CREW model, for binary searches.
- Can improve further, we will see later.
- The technique to achieve optimality is a general technique, with several applications. We will see more applications of this later.

A Further Improvement

- Where is the scope for improvement?
- Each binary search takes O(log n) time, and we also have O(n/log n) subproblems each of size O(log n).
- To get further improvements, we should look at both aspects.
- Can we search faster? Parallel?

A Further Improvement

- Parallel search first.
- Consider a sorted array A of n element and we want to search for an element x.
- Given p processors, we can always search at positions (indices) 1, n/p, 2n/p, ..., n.
- Record the result of each comparison as a 1 or 0 with 1 for position i indicating that A[i] < x and 0 indicating that A[i] >= x.
- The sequence of p results will have :
  - Either all 1's
  - Either all 0's
  - A shift from 1's to 0's

A Further Improvement

- Parallel search first.
- Consider a sorted array A of n element and we want to search for an element x.
- Given p processors, we can always search at positions (indices) 1, n/p, 2n/p, ..., n.
- Record the result of each comparison as a 1 or 0 with 1 for position i indicating that A[i] < x and 0 indicating that A[i] >= x.
- The sequence of p results will have :
  - Either all 1's : x is not in A
  - Either all 0's : x is not in A
  - A shift from 1's to 0's : x is likely in the n/p segment corresponding to the shift from 1 to 0.

Search in Parallel

- We can identify the next step depending on the three cases.
  - Either all 1's : x is not in A
  - Either all 0's : x is not in A
  - A shift from 1's to 0's : x is likely in the n/p segment corresponding to the shift from 1 to 0.
    - Therefore, search recursively in the corresponding segment of size n/p while still using p processors.
- The recurrence relation for the time taken is
  - $T(n) = T(n/p) + O(1)$, for a solution of $T(n) = O(\log_p n)$.
- The work done has the recurrence $W(n) = p.\ W(n/p) + O(p)$, for a solution of $W(n) = O(n/p)$.

Search in Parallel

- Consider typical values of p.
- For $p = O(1)$, no change in time taken asymptotically.
- For $p = O(\log n)$, the time taken is $O(\log n / \log\log n)$.
- For $p = O(n^{1/2})$, the time taken is $O(\log n / \log n^{1/2}) = O(1)$!
  - Of course, looks like wasteful from a work point of view.
  - Let us see what it is good for!

From Parallel Search to Merge

- Recall our idea to arrive at an optimal algorithm to merge two sorted arrays A and B.
- We rank a few elements of A in B to partition B into sub-arrays.
- Let us consider ranking $n^{1/2}$ elements of A in B.
- We have n processors, so each search can use $n^{1/2}$ processors!
- Each search now finishes in O(1) time.

From Parallel Search to Merge

- Let us consider ranking $n^{1/2}$ elements of A in B.
- We have n processors, so each search can use $n^{1/2}$ processors!
- Each search now finishes in O(1) time.
- There is a downside however.
- The partitions of A are now much longer at $n^{1/2}$ each.
- The partitions of B are like in the earlier case, unknown.

From Parallel Search to Merge

- There is a downside however.
- The partitions of A are now much longer at $n^{1/2}$ each.
- The partitions of B are like in the earlier case, unknown.
- But, can use recursion to make further progress.

From Parallel Search to Merge

- The partitions of A are now much longer at $n^{1/2}$ each.
- The partitions of B are like in the earlier case, unknown.
- But, can use recursion to make further progress.
- Recursively apply the same procedure on each partition of A into the corresponding partition of B.
- Notice that each part of A is only $n^{1/2}$ in size.
- We want to rank $n^{1/4}$ element of each part of A into the corresponding B.

From Parallel Search to Merge

- The recurrence relation guiding this process is captured by $T(n, m) = \max_i T(n^{1/2}, m_i) + O(1)$.
  - In the above, n and m refer to the lenght of A and B respectively.
  - And, m_i refers to the length of the i^{th} partition of B.
- Can show that $T(n,m) = O(\log \log n)$.
- Once recursion ends, each partition of A and partitions of B will be $O(\log\log n)$ long, and we merge them sequentially.

The Power of CRCW – Minima

- Two points of interest
  - Illustrate the power of CRCW models
  - Illustrate another optimality technique.
- Find minima of n elements.
  - Input: An array A of n elements
  - Output: The minimum element in A.
- From what we already know:
  - Standard sequential algorithm not good enough
  - Can use an upward traversal, with min as the operator at each internal node. Time = O(log n), work = O(n).

The Power of CRCW – Minima

- Our solution steps:
  - Design a $O(n^2)$ work, $O(1)$ time algorithm.
  - Gain optimality by sacrificing runtime to $O(\log \log n)$.

An O(1) Time Algorithm

	12	17	8	18	26
12	--	1	0	1	1
17	0	--	0	1	1
8	1	1	--	1	1
18	0	0	0	--	1
26	0	0	0	0	--

- Use n^2 processors.
- Compare A[i] with A[j] for each i and j.
- Now can identify the minimum.

An O(1) Time Algorithm

	12	17	8	18	26
12	--	1	0	1	1
17	0	--	0	1	1
8	1	1	--	1	1
18	0	0	0	--	1
26	0	0	0	0	--

- Use n^2 processors.
- Compare A[i] with A[j] for each i and j.
- Now can identify the minimum.
  - How?

An O(1) Time Algorithm

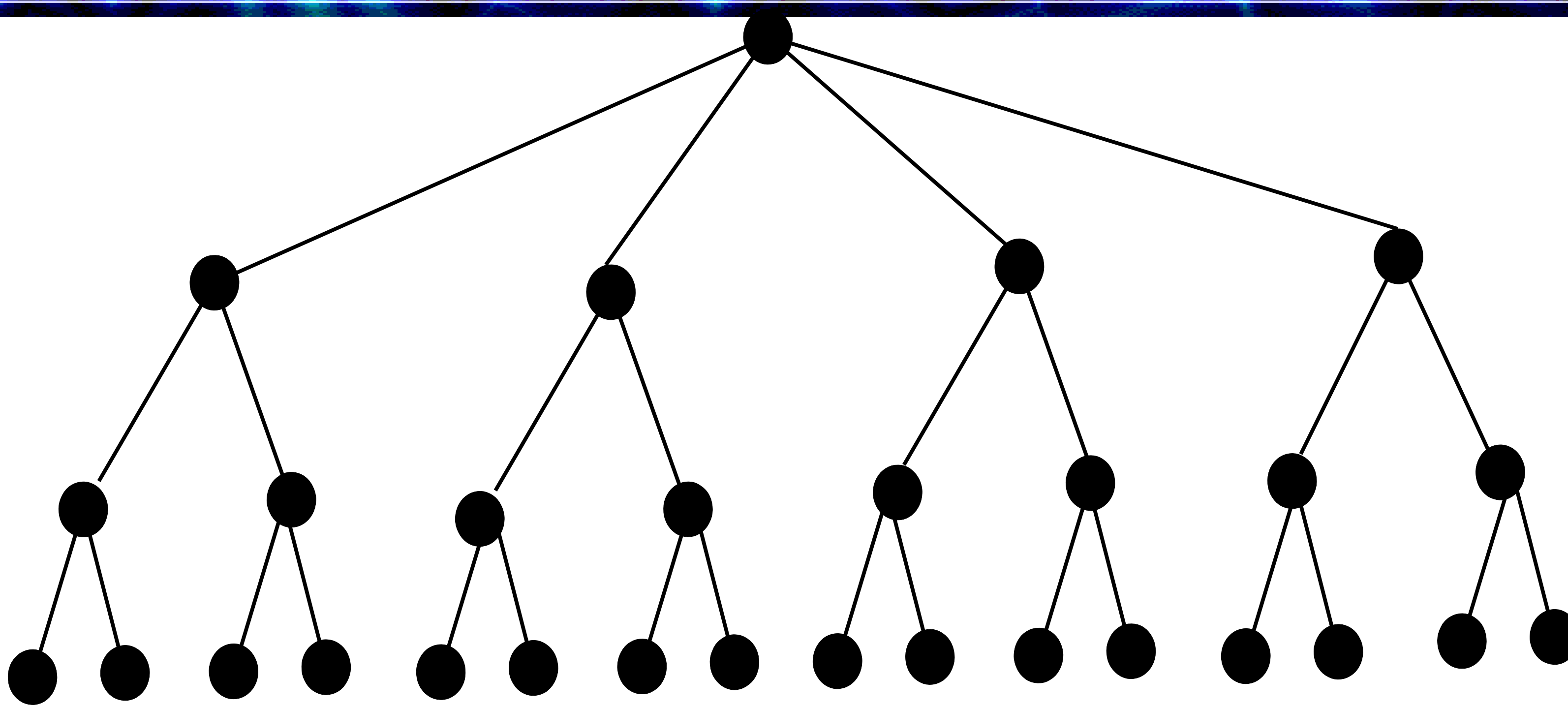
	12	17	8	18	26
12	--	1	0	1	1
17	0	--	0	1	1
8	1	1	--	1	1
18	0	0	0	--	1
26	0	0	0	0	--

- Use n^2 processors.
- Compare A[i] with A[j] for each i and j.
- Now can identify the minimum.
    - How?
- Where did we need the CRCW model?

Towards Optimality

- The earlier algorithm is heavy on work.
- To reduce the work, we proceed as follows.
- We derive an O(nlog log n) work algorithm running in O(log log n) time.
- For this, use a doubly logarithmic tree.
  - Defined in the following.

Doubly Logarithmic Tree



- Let there be $n = 2^{2^k}$ leaves, the root is level 0. The root has $\sqrt{n} = 2^{2^{k-1}}$ children.
- In general, a node at level i has $2^{2^{k-i-1}}$ children, for $0 \leq i \leq k-1$.
- Each node at level k has two leaf nodes as children.

Doubly Logarithmic Tree

- Some claims:
  - Number of nodes at level i is $2^{2^k - 2^{k-i}}$.
  - Number of nodes at the kth level is n/2.
  - Depth of a doubly logarithmic tree of n nodes is k+1 = log log n + 1.
- To compute the minimum using a doubly logarithmic tree:
  - Each internal node performs the min operation does not suffice.
  - Why?

Minima Using the Doubly Logarithmic Tree

- Intuition:
  - Should spend only O(1) time at each internal node.
  - Use the O(1) time algorithm at each internal node.
- At each internal node of level i, if there are c_i children, use c_i^2 processors.
  - Minima takes O(1) time at each level.
  - Also, No. of nodes at level i x No. of processors used = $2^{2^k - 2^{k-i}} . (2^{2^{k-i-1}})^2 = 2^{2^k} = n$.

Minima Using a Doubly Logarithmic Tree

- Second, slightly improved result:
  - With n processors, can find the minima of n numbers in O(log log n) time.
  - Total work = O(n log log n).
- Still suboptimal by a factor of O(log log n).
- We now introduce a technique to achieve optimality.

Accelerated Cascading

- Our two algorithms:
  - Algorithm 1: A slow but optimal algorithm.
    - Binary tree based: O(log n) time, O(n) work.
  - Algorithm 2: A fast but non-optimal algorithm
    - Doubly Logarithmic tree based: O(log log n) time, O(nlog log n) work.
- The accelerated cascading technique suggests combining two such algorithms to arrive at an optimal algorithm
  - Start with the slow but optimal algorithm till the problem is small enough
  - Switch over to the fast but non-optimal algorithm.

Accelerated Cascading

- The binary tree based algorithm starts with an input of size n.
- Each level up the tree reduces the size of the input by a factor of 2.
- In log log log n levels, the size of the input reduces to $n/2^{\text{logloglog } n} = n/\text{loglog } n$.

- Now switch over to the fast algorithm with n/loglog n processors, needing O(log log (n/log log n)) time.

Final Result

- Total time = $O(\log \log \log n) + O(\log \log n)$.
- Total work = $O(n)$.
- Need CRCW model.
- Where did we need the CRCW model?