Design and Analysis of Algorithms

Lecture 11

Parallel Algorithms

Topics

- What is a parallel computer?
- How do we model a parallel computer?
- Example parallel algorithms
- How do distributed systems differ?
- Example distributed algorithms

What is a parallel computer?

- A collection of processors (processing elements – PEs) that are interconnected
  - PEs can be: Pentium, Power PC, *etc*.
  - The PEs work co-operatively
  - Interconnection Network: Mesh, Hypercube, clusters, *etc*.
- Parallel algorithms perform more than one operation per time unit
  - Need a parallel computer to run a parallel algorithm/program
  - Parallel computers: IBM Summit (US$200M to build), IBM Sierra, Sunway Taihulight are the top three most powerful computers in the world!
    - IBM Summit contains 2,397,824 cores, and requires power of 9,783 kW!
  - You can see the top 500 most powerful supercomputers in

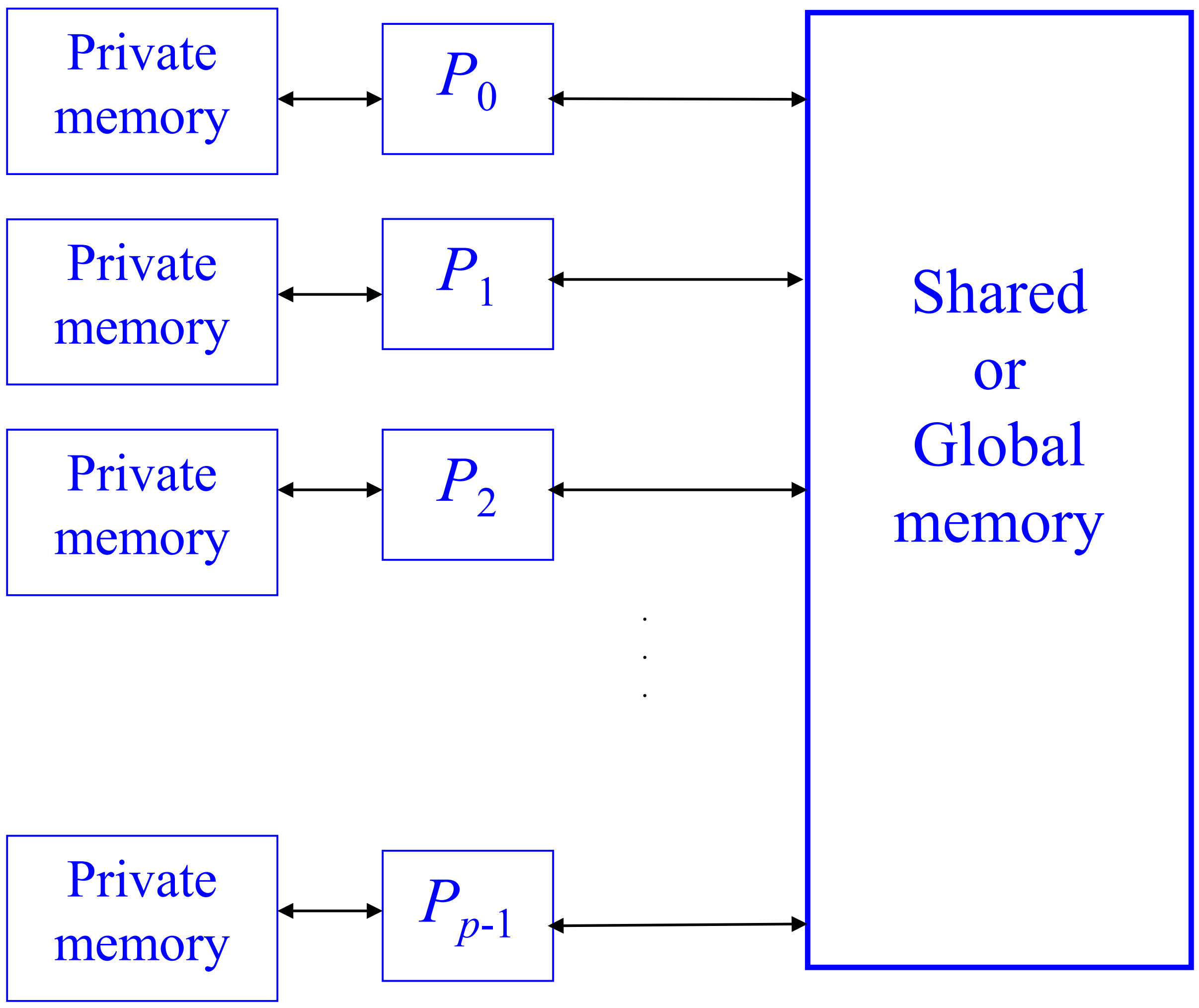
    https://www.top500.org/lists/2018/11/

  - Need tools, languages, compilers, *etc*. to help build parallel programs
  - Tools/languages: Message Passing Interface (MPI), PVM, C*, C-Linda, Fortran90, *etc*.

Parallel computer model – PRAM

- Parallel Random Access Machine (PRAM) model of computation
  - introduced by Fortune and Wyllie in 1978
  - communication cost negligible
  - synchronization overhead negligible
- PRAM structure
  - It contains *p* PEs (just like the PE in a RAM model)
  - Each PE has access to a shared global memory
  - Each PE has its own private memory
  - All PEs can read / write global memory in parallel (at same time)
- SIMD: Single Instruction Multiple Data
  - All PEs execute the same instruction at the same time on different data

PRAM – structure



Real world

- Real computers cannot perform parallel accesses to global memory in unit time
  - More processors slow down access to memory
- Real parallel computers typically have a communication network that supports the abstraction of global memory
- PRAM ignores communication costs and synchronisation costs
- So … if performance of a parallel algorithm is not good on a PRAM, it is meaningless to be implemented on a real machine!

PRAM - Shared memory model

- A memory location can be subject to
  - ER: exclusive read
  - EW: exclusive write
  - CR: concurrent read
  - CW: concurrent write, with some policy on what value to store:
    - common: all PEs write same thing
    - arbitrary: only store one value; or choose PE with the smallest index
    - reduction: apply min, max, sum, etc
- There are four classes of PRAM
  - EREW – the most restrictive and most practical
  - ERCW
  - CREW
  - CRCW – the most liberal and least practical
- EREW can simulate the other models but with additional steps
  - Thus EREW has a worse complexity

How to analyze PRAM?

- Like in RAM model, we are interested in counting steps to solve a problem of size n, i.e., time complexity.
  - But now need to also count the number of processors (PEs) being used
- Consider a problem of size n, and the following notations
  - $T(n)$ is the time complexity of the *parallel* algorithm for the problem using $P(n)$ PEs
  - $T^*(n)$ is the smallest worst-case time complexity of a *sequential* algorithm for the problem
  - $S(n)$ is the speed-up of the parallel algorithm, i.e.,

$$S(n) = T^*(n) \,/\, T(n)$$

    - Using more PEs, i.e., larger $P(n)$ can lead to faster algorithm, i.e., smaller $T(n)$ and thus larger speedup $S(n)$
  - $C(n)$ is the *work* or *cost* of the parallel algorithm, computed as

$$C(n) = P(n) \times T(n)$$

  - $E(n) \leq 1$ is efficiency in using the PEs, calculated as

$$E(n) = T^*(n) \,/\, C(n)$$

OR $E(n) = S(n) \,/\, P(n)$

How to analyze PRAM? (cont.)

- We need to compare $C(n)$ against $T^*(n)$ to measure the quality of the parallel algorithm
  - A parallel algorithm is *cost optimal* if $C(n) = T^*(n)$
  - A parallel algorithm is *cost efficient* if $C(n)$ is within a *poly-logarithmic* factor of being cost optimal, i.e., within $O(\log^k n)$ factor, for a constant k
- Using fewer PEs can improve cost $C(n)$
  - If all PEs are used all the time, we expect $E(n)$ to be close to one
    - A parallel algorithm is cost optimal only if $E(n) = 1$
- Designing a *cost optimal* with a good speedup is difficult
  - Larger speedup usually require more PEs, many of which are idle increasing its cost or reducing its efficiency
- **Design Goals:**
  - Number of PEs must be bounded by problem size
  - Parallel runtime must be significantly smaller than the execution time of the best sequential algorithm
  - The cost or work of the algorithm is efficient

How to analyze PRAM? (cont.)

Example:

- Consider a parallel algorithm that computes the sum of array $A[1\ldots n]$ of integers using n PEs in $T(n) = \Theta(\log n)$ time
  - Number of PEs $P(n) = n$
  - Cost $C(n) = n * \log n = \Theta(n \log n)$
- Sequential time to compute the sum of array $A[1\ldots n]$ of integers has $T^*(n) = \Theta(n)$
  - So the parallel algorithm is not *cost optimal* because $C(n) > T^*(n)$
  - The parallel algorithm is *cost efficient* because its $C(n)$ is within $O(\log n)$ factor from $T^*(n)$
  - The speedup of the parallel algorithm is $S(n) = T^*(n) / T(n) = n / \log n$
  - Efficiency $E(n) = T^*(n) / C(n) = n / n \log n = 1 / \log n$
    - Equivalently, $E(n) = S(n) / P(n) = (n / \log n) / n = 1 / \log n$

Parallel search algorithm

- Assume n **unsorted** integers in an array $A[1...n]$
- **Problem:** does A contain a value x?
- Serial algorithm takes $\Theta(n)$ time

Parallel_Search_CRCW (x, A[1 .. n]) // using CRCW model

    index ← -1 // initialize *index* with an invalid value -1

    forall P_i **do in parallel** // $1 \le i \le n$; NOTE: this is not a for loop!

        if $A[i] = x$ **then** // CR in reading the value of x

            index ← i // CW in writing to *index*

        endif

    endfor

Analysis

- What model is used?
  - It allows all PEs to read the value x at the same time // CR model
  - If array A contains more than one value of x, more than one PEs write into *index* for x at the same time // CW model
    - What CW model? // let PE with smallest ID to write
  - Thus, we need a CRCW model
- How many processors? $n = O(n)$
- How much time? $O(1)$
- Work = $1 * n \rightarrow O(n)$
- The best sequential algorithm can search x in $O(n)$
  - The CRCW algorithm is work-optimal

Parallel search - what if we use EREW model?

- All PEs in EREW model can not read x at the same time

Search Algorithm for EREW model

- Use a broadcast algorithm to put x in each element of an array $B[1 .. n]$

  Broadcast (x, B[1 .. n]) // assume n is a power of 2

  Step 1: P_1 reads x and stores it in $B[1]$

  Step 2: P_1 reads $B[1]$ and stores it in $B[2]$

  Step 3: P_1 reads $B[1]$ and stores it in $B[3]$ and P_2 reads $B[2]$ and stores it in $B[4]$

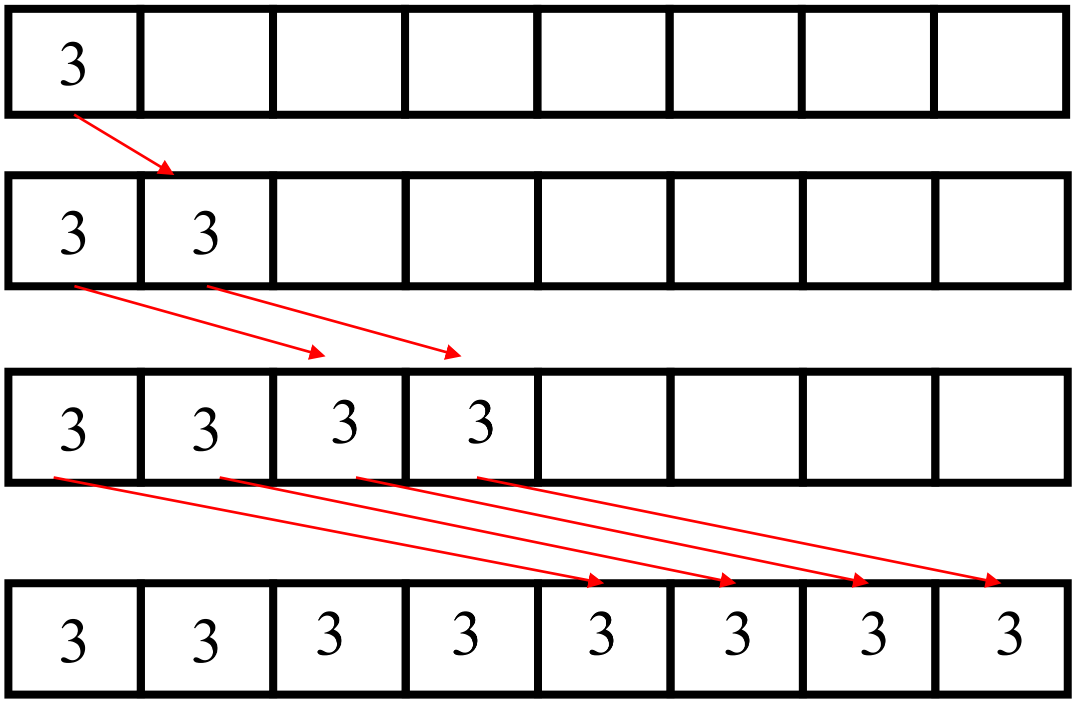
  Step 4: P_1 reads $B[1]$ and stores it in $B[5]$, P_2 reads $B[2]$ and stores it in $B[6]$, P_3 reads $B[3]$ and stores it in $B[7]$, and P_4 reads $B[4]$ and stores it in $B[8]$, etc

  - ➢ The broadcast algorithm requires $\log_2 n$ steps using $O(n)$ PEs

- Each PE i reads x from its own $B[i]$ and compares it with $A[i]$ // takes 1 step

  - ➢ **If** $B[i] = A[i]$, **then** P_i sets $B[i] = i$; Otherwise, it sets $B[i] = \infty$

Broadcast – an example for $x = 3$, $n = 8$



Algorithm Broadcast (x, B[1 .. n]) // Please Check!

B[1] $\leftarrow x$

for $i = 1$ to $\log n$ **do**

forall P_k where $2^{i-1} \le j \le 2^i - 1$ and $k = j + 1 - 2^{i-1}$ **do in parallel**

$B[j+1] \leftarrow B[j + 1 - 2^{i-1}]$

endfor

endfor

$\boldsymbol{i = 1}$**:** $2^{i-1} \le j \le 2^i-1 \rightarrow 1 \le j \le 1$; $j = 1 \rightarrow k = j + 1 - 2^{i-1} = 1$; PE_1: $B[2] \leftarrow B[1]$

$\boldsymbol{i = 2}$**:** $2^{i-1} \le j \le 2^i-1 \rightarrow 2 \le j \le 3$; $j = 2 \rightarrow k = j + 1 - 2^{i-1} = 1$; PE_1: $B[3] \leftarrow B[1]$

$j = 3 \rightarrow k = j + 1 - 2^{i-1} = 2$; PE_2: $B[4] \leftarrow B[2]$

$\boldsymbol{i = 3}$**:** $2^{i-1} \le j \le 2^i-1 \rightarrow 4 \le j \le 7$; $j = 4 \rightarrow k = j + 1 - 2^{i-1} = 1$; PE_1: $B[5] \leftarrow B[1]$

$j = 5 \rightarrow k = j + 1 - 2^{i-1} = 2$; PE_2: $B[6] \leftarrow B[2]$

$j = 6 \rightarrow k = j + 1 - 2^{i-1} = 3$; PE_3: $B[7] \leftarrow B[3]$

$j = 7 \rightarrow k = j + 1 - 2^{i-1} = 4$; PE_4: $B[8] \leftarrow B[4]$

Search, what if we use EREW model? (cont.)

- Array $B[1 .. n]$ now contains the result of comparisons
  - All n comparisons are performed in 1 step using n PEs.
- Given the comparison results in $B[1 .. n]$, how to determine if there is x in array $A[1 .. n]$? **Use the *fan-in* algorithm!**
- Algorithm *fan-in* is used to find the minimum value in $B[1 .. n]$

  fan_in ($B[1 .. n]$)   // assume n is a power of 2

  Step 1: Assign a pair of numbers in $B[1 .. n]$ to each of $n/2$ PEs

  Step 2: Each PE determines the minimum between its two numbers, and stores the minimum number into $B[1 .. n/2]$

  Step 3: Assign a pair of numbers in $B[1 .. n/2]$ to each of $n/4$ PEs

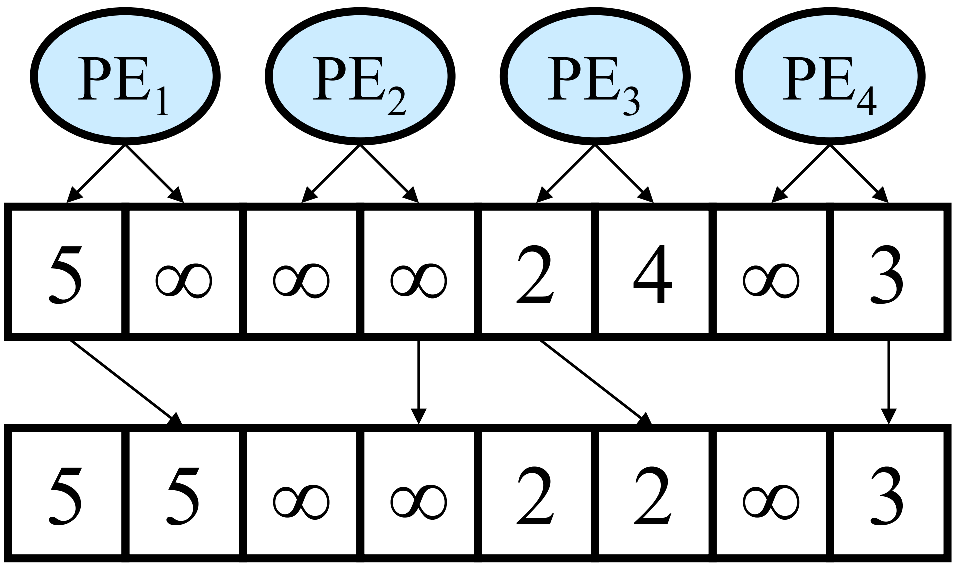
  Step 4: Each PE determines the minimum between its two numbers, and store the minimum number into $B[1 .. n/4]$

  Repeat the steps until array B contains only one number, e.g., i, denoting PE i finds $A[i] = x$

  - What is the time and work complexity of Algorithm *fan-in*?
  - **Note:** The algorithm assumes synchronization – all PEs read and write at the same time
- What is the time and work complexity of algorithm Parallel_Search_EREW?

Can you write the pseudocode for Algorithm *fan_in*?

Fan-in – an example

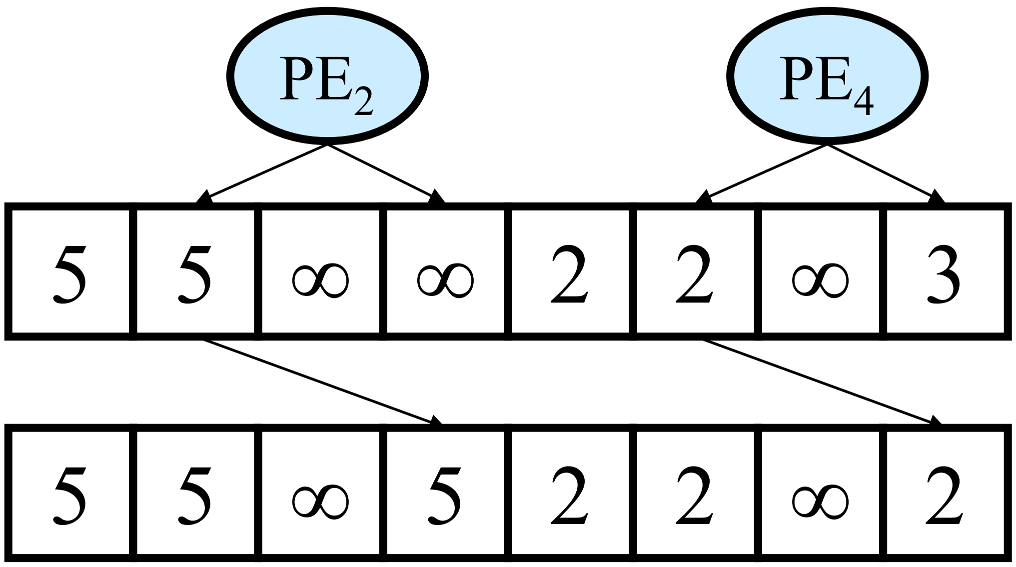


for $i = 1$ to $\log n$ **do**
  forall P_j where $1 \le j \le n/2$ **do in parallel**
    if $2j \bmod 2^i = 0$ **then**
      if $B[2j] > B[2j - 2^{i-1}]$ **then**
        $B[2j] \leftarrow B[2j - 2^{i-1}]$

$\boldsymbol{i = 1, 2^1 = 2}$

$j = 1$, $2 \% 2 = 0 \rightarrow 2j - 2^{i-1} = 1$; $B[2] > B[1] \rightarrow PE_1$: $B[2] \leftarrow B[1] = 5$
$j = 2$, $4 \% 2 = 0 \rightarrow 2j - 2^{i-1} = 3$; $B[4] = B[3] \rightarrow PE_2$: *does nothing*
$j = 3$, $6 \% 2 = 0 \rightarrow 2j - 2^{i-1} = 5$; $B[6] > B[5] \rightarrow PE_3$: $B[6] \leftarrow A[5] = 2$
$j = 4$, $8 \% 2 = 0 \rightarrow 2j - 2^{i-1} = 7$; $B[8] < B[7] \rightarrow PE_4$: *does nothing*

Fan-in – an example (cont.)

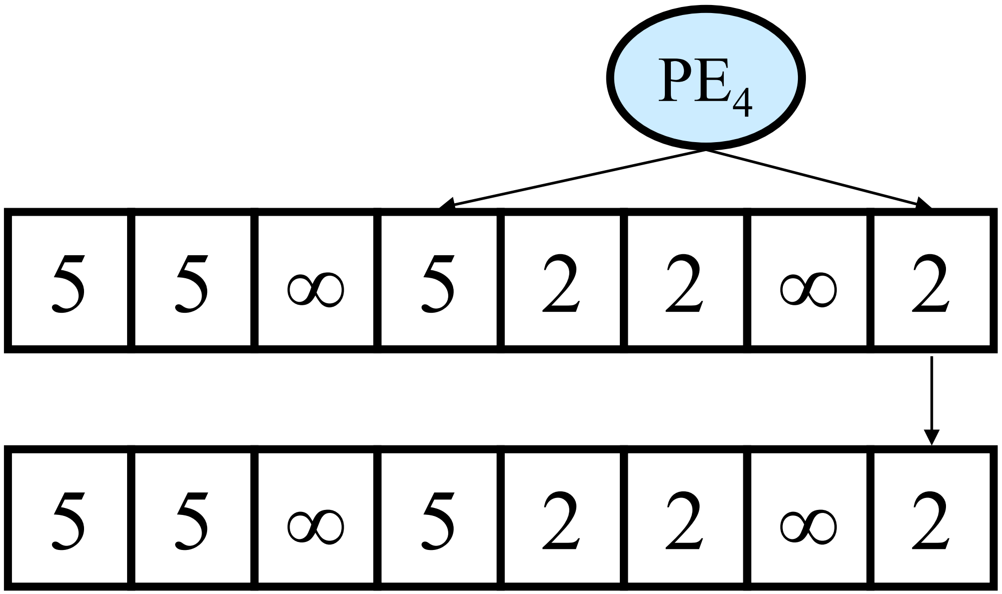


for $i = 1$ to $\log n$ **do**
  forall P_j where $1 \le j \le n/2$ **do in parallel**
    if $2j \bmod 2^i = 0$ **then**
      if $B[2j] > B[2j - 2^{i-1}]$ **then**
        $B[2j] \leftarrow B[2j - 2^{i-1}]$

***i* = 2, 2² = 4**

- $j = 1$, $2 \% 4 = 2$ → PE$_1$: *does nothing*
- $j = 2$, $4 \% 4 = 0$ → $2j - 2^{i-1} = 2$; $B[4] > B[2]$ → PE$_2$: $B[4] \leftarrow B[2] = 5$
- $j = 3$, $6 \% 4 = 2$ → PE$_3$: *does nothing*
- $j = 4$, $8 \% 4 = 0$ → $2j - 2^{i-1} = 6$; $B[8] > B[6]$ → PE$_4$: $B[8] \leftarrow B[6] = 2$

Fan-in – an example (cont.)



for i = 1 to log n **do**
  forall P_j where $1 \le j \le n/2$ **do in parallel**
    if $2j \bmod 2^i = 0$ **then**
      if $B[2j] > B[2j - 2^{i-1}]$ **then**
        $B[2j] \leftarrow B[2j - 2^{i-1}]$

$\boldsymbol{i = 3, 2^3 = 8}$

$j = 1$, $2 \% 8 = 2$ → PE_1: *does nothing*
$j = 2$, $4 \% 8 = 4$ → PE_2: *does nothing*
$j = 3$, $6 \% 8 = 6$ → PE_3: *does nothing*
$j = 4$, $8 \% 8 = 0$ → $2j - 2^{i-1} = 4$; $B[8] < B[4]$ → PE_4: *does nothing*

Search, what if we use EREW model? (cont.)

Parallel_Search_EREW (x, A[1 .. n])

Broadcast(x, B[1 .. n]) // $O(\log_2 n)$; n PEs

// $1 \le i \le n$

forall P_i **do in parallel** // $O(1)$; n PEs

if $A[i] = B[i]$ **then**

$B[i] \leftarrow i$

else

$B[i] \leftarrow \infty$

endfor

// *fan_in* returns PE with the smallest ID that finds $A[i] = x$

return i = *fan_in* (B[1 .. n]) // $O(\log_2 n)$; n PEs

$T(n) = ?$

$C(n) = ?$

Cost optimal *or* cost efficient?

Adding n numbers in parallel

- Assume n integers in an array $A[1...n]$
- Serial algorithm takes $\Theta(n)$ time
- Parallel algorithm:
  - Assume EREW PRAM
  - Assume A is in the global memory
  - Sum will end up in $A[n]$
  - Assume n is a power of 2
- Similar to algorithm *fan_in* ($A[1...n]$)
  - Instead of choosing the **minimum**, each active PE computes the **sum** of its two numbers
  - Algorithm *fan_in* ($A[1...n]$) can also be used to find the **maximum**, or compute the **products** of the numbers in $A[1...n]$

Adding n numbers in parallel (cont.)

Sum_EREW (A[1 .. n])

for $i = 1$ to $\log n$ **do**

  forall P_j where $1 \le j \le n/2$ **do in parallel**

    if $2j \bmod 2^i = 0$ **then**

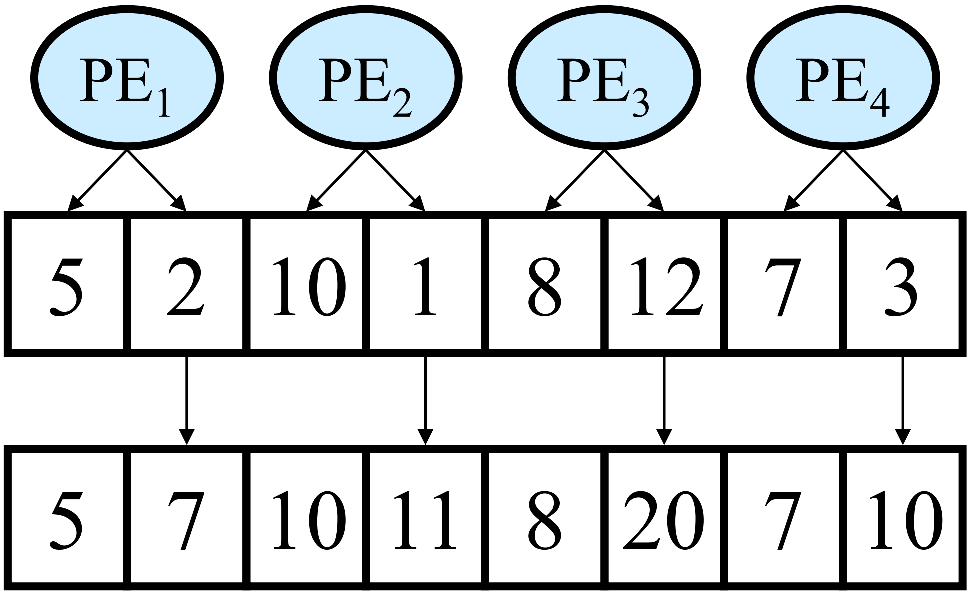
      $A[2j] \leftarrow A[2j] + A[2j - 2^{i-1}]$

    endif

  endfor

endfor

Adding n numbers in parallel – example

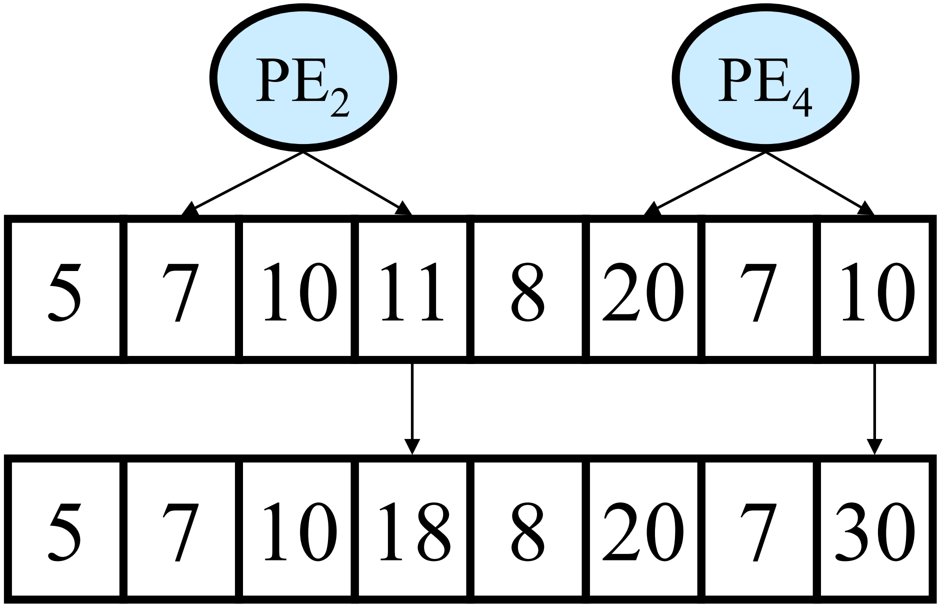


for i = 1 to log n **do**
  forall P_j where $1 \le j \le n/2$ **do in parallel**
    if $2j \bmod 2^i = 0$ **then**
      $A[2j] \leftarrow A[2j] + A[2j - 2^{i-1}]$

$\boldsymbol{i = 1, 2^1 = 2}$

$j = 1$, $2 \% 2 = 0 \rightarrow$	PE_1: $A[2] \leftarrow A[2] + A[1] = 2 + 5 = 7$
$j = 2$, $4 \% 2 = 0 \rightarrow$	PE_2: $A[4] \leftarrow A[4] + A[3] = 1 + 10 = 11$
$j = 3$, $6 \% 2 = 0 \rightarrow$	PE_3: $A[6] \leftarrow A[6] + A[5] = 12 + 8 = 20$
$j = 4$, $8 \% 2 = 0 \rightarrow$	PE_4: $A[8] \leftarrow A[8] + A[7] = 3 + 7 = 10$

Adding n numbers in parallel – example (cont.)

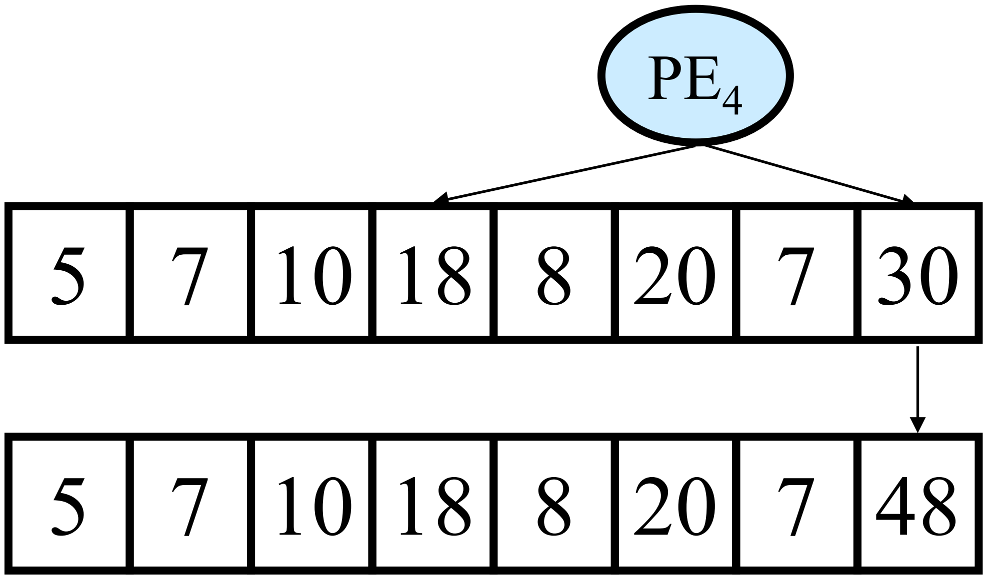


for $i = 1$ to $\log n$ **do**
  forall P_j where $1 \le j \le n/2$ **do in parallel**
    if $2j \bmod 2^i = 0$ **then**
      $A[2j] \leftarrow A[2j] + A[2j - 2^{i-1}]$

***i* = 2, 2² = 4**

$j = 1$, $2 \% 4 \neq 0$
$j = 2$, $4 \% 4 = 0 \rightarrow$ PE_2: $A[4] \leftarrow A[4] + A[2] = 11 + 7 = 18$
$j = 3$, $6 \% 4 \neq 0$
$j = 4$, $8 \% 4 = 0 \rightarrow$ PE_4: $A[8] \leftarrow A[8] + A[6] = 10 + 20 = 30$

Adding n numbers in parallel – example (cont.)



for $i = 1$ to $\log n$ **do**
  forall P_j where $1 \leq j \leq n/2$ **do in parallel**
    if $2j \bmod 2^i = 0$ **then**
      $A[2j] \leftarrow A[2j] + A[2j - 2^{i-1}]$

$\boldsymbol{i = 3, 2^3 = 8}$

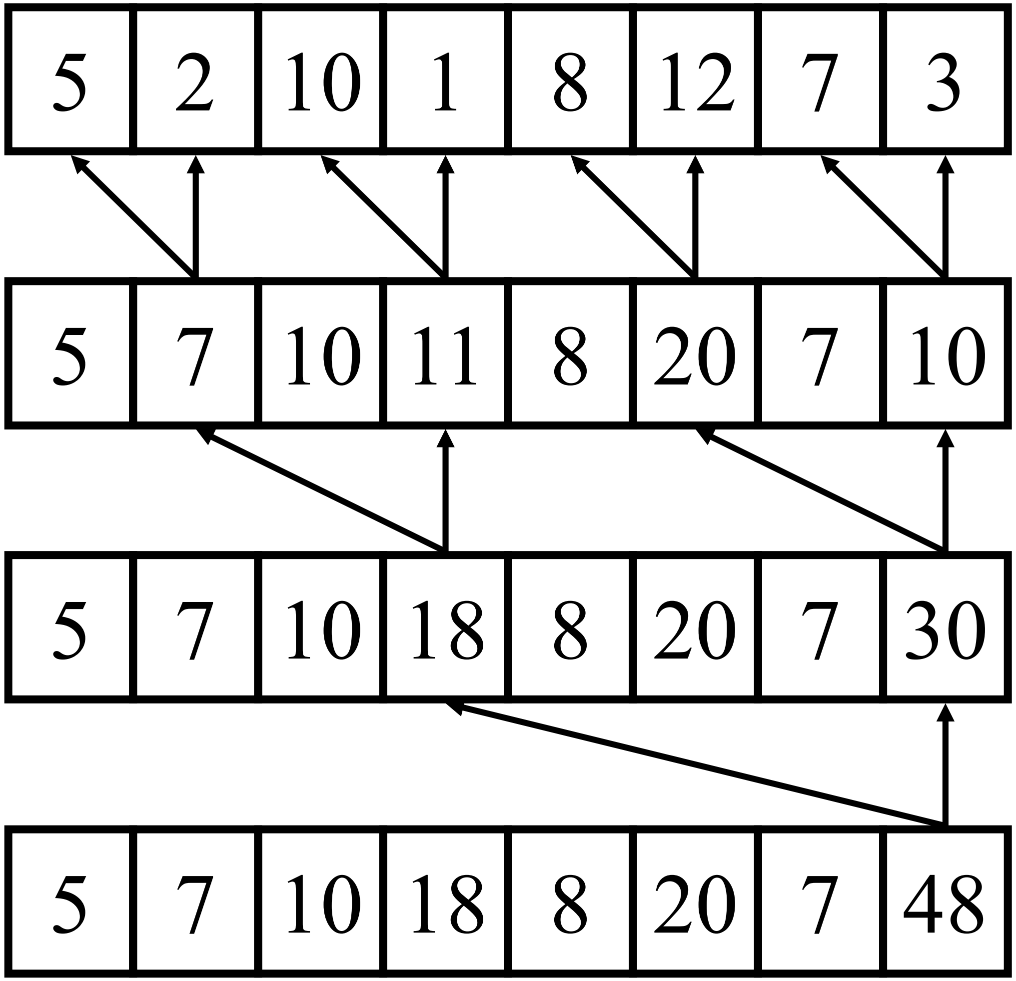
$j = 1, 2 \% 8 \neq 0$

$j = 2, 4 \% 8 \neq 0$

$j = 3, 6 \% 8 \neq 0$

$j = 4, 8 \% 8 = 0 \rightarrow$ PE_4: $A[8] \leftarrow A[8] + A[4] = 30 + 18 = 48$

Adding n numbers in parallel – example (cont.)



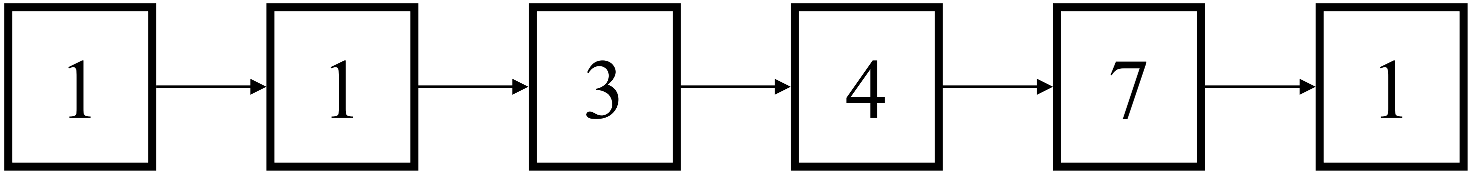
Analysis

- How many processors? $n/2 = O(n)$
- How much time on PRAM? $O(\log n)$
- Work = $O(n \log n)$
- Not work-optimal, but work-efficient Why?

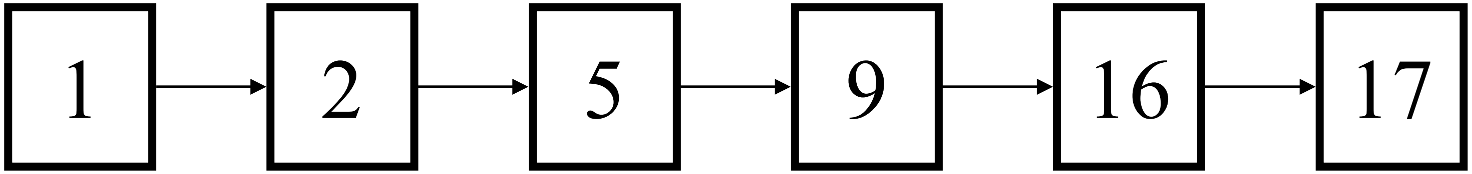
Note: The algorithm assumes synchronization – all PEs read and write at the same time

Pointer jumping

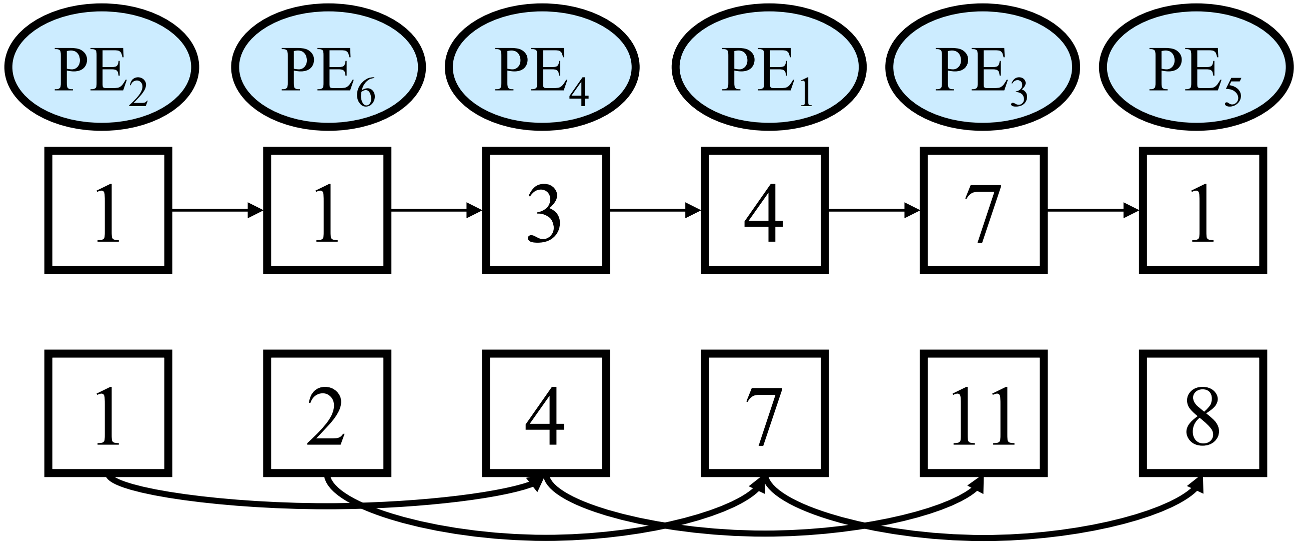
- Suppose we want to process a linked list with n items.
  - *next* is the name of the next pointer
  - Sequential time: $O(n)$
- PRAM: Assume that there is one PE for each object in the list.
- Say we want to prefix sum a linked list, *e.g.*,



results in



Pointer jumping – example

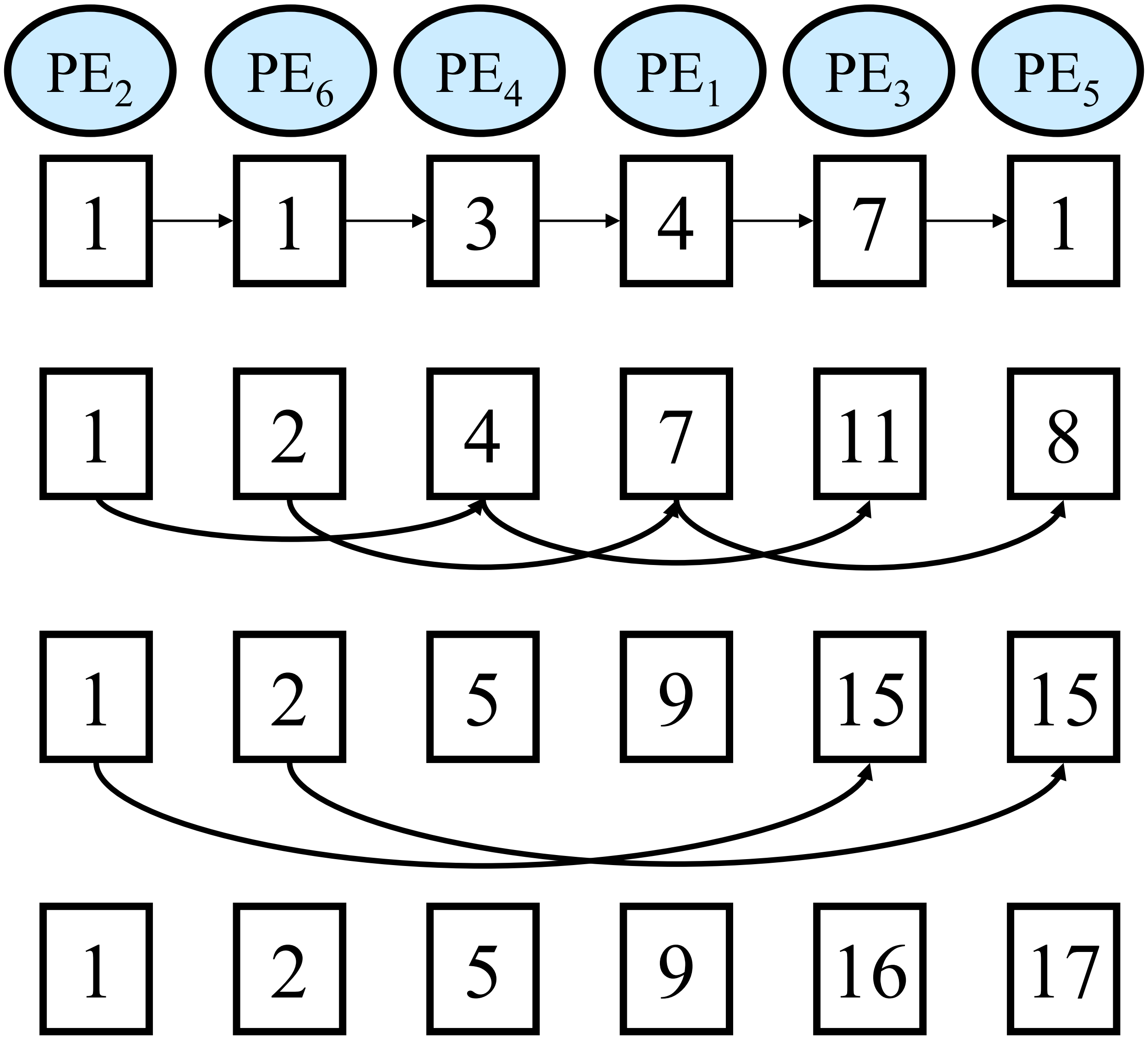


forall PE$_i$ with *next* != null **do in parallel**

value(*i.next*) = value(*i*) + value(*i.next*)

i.next = *i.next.next*

Pointer jumping – example (cont.)



Analysis

- $T(n) = O(\log n)$ time
- $P(n) = O(n)$ processors
- $C(n) = O(n \log n)$ work
- Not work-optimal (sequential $O(n)$) Is it work efficient?
- Structure of the list is destroyed,
  - Make a copy in $O(1)$ time if want to keep the list
- What other computation can use the pointer jumping? Finding the minimum? etc?

Parallel Sorting

- Assume n integers in an array $A[1 \ldots n]$
- **Problem:** sort the array in parallel

- Design of parallel sorting highly dependent on the chosen computational model
  - CRCW model with CW (**reduces to sum**) :
    - We can sort n elements using $(n^2 - n)/2$ PEs in $O(1)$ time complexity!
    - Is it work efficient?
    - It is not practical
  - EREW model:
    - We can sort n elements using n PEs in $\Omega(\log n)$ time complexity
    - Batcher's EvenOddMergeSort can sort an array of n elements using n PEs, with time complexity of $O(\log^2 n)$ → cost = $O(n\log^2 n)$
      - So, it is cost efficient, but not cost optimal!

Parallel Sorting – CRCW model

SortCRCW ($A[1 .. n]$)

// Phase 1: $O(1)$

forall PE_i where $1 \le i \le n$ **do in parallel**

$L[i] \leftarrow 1$ // initialization

endfor

// Phase 2: $O(1)$; e.g., for $n = 4$, ij = 12, 13, 14, 23, 24, 34

forall PE_{ij} where $1 \le i, j \le n$ and $i < j$ **do in parallel** // there are $n\,(n\text{-}1)/2$ PEs

if $A[i] > A[j]$ **then** // e.g., for $n = 4$, ij = 12, 13, 14, 23, 24, 34

In parallel, all PE_{ij} write $L[i] \leftarrow 1$ // CW – reduction to sum

else

In parallel, all PE_{ij} write $L[j] \leftarrow 1$ // CW – reduction to sum

endfor

// Phase 3: $O(1)$

forall PE_i where $1 \le i \le n$ **do in parallel**

$A[L[i]] \leftarrow A[i]$

endfor

Parallel Sorting – CRCW model (Example)

Array L
Phase 1

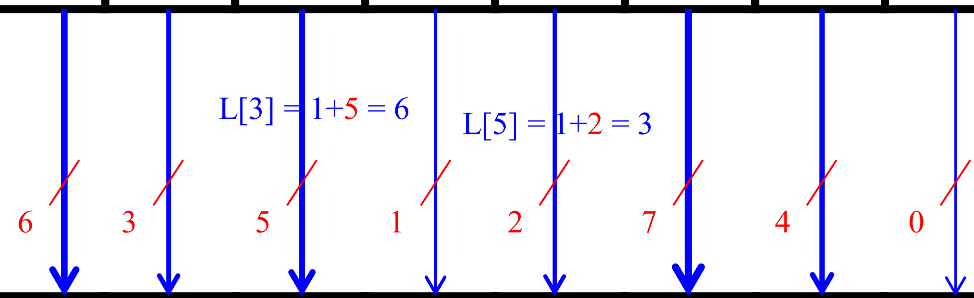
1	1	1	1	1	1	1	1

Each element is compared with 7 other elements "2" → need 8 * 7 / 2 = 28 PEs, i.e., ij =12, 13, … 18 23, 24, … 28, … 67, 68, 78

Array A

9	5	7	3	4	10	6	2

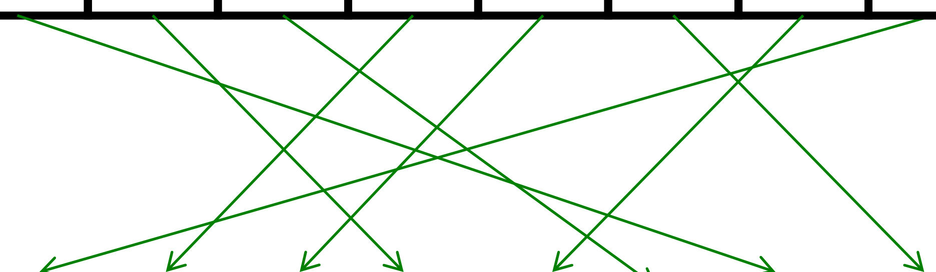
- 7 PE compares "9" with one other element, e.g., "2"
- "9" is larger 6 times
- 6 PE_{ij} write "1" to L[1]
- L[1] =1+1+1+1+1+1+1=7



- 7 PE compares "2" with one other element, e.g. "4"
- "2" is larger 0 time
- No PE_{ij} writes "1" to L[8]
- L[8] = 1

Array L
Phase 2

7	4	6	2	3	8	5	1



1 2 3 4 5 6 7 8

Array A
Phase 3

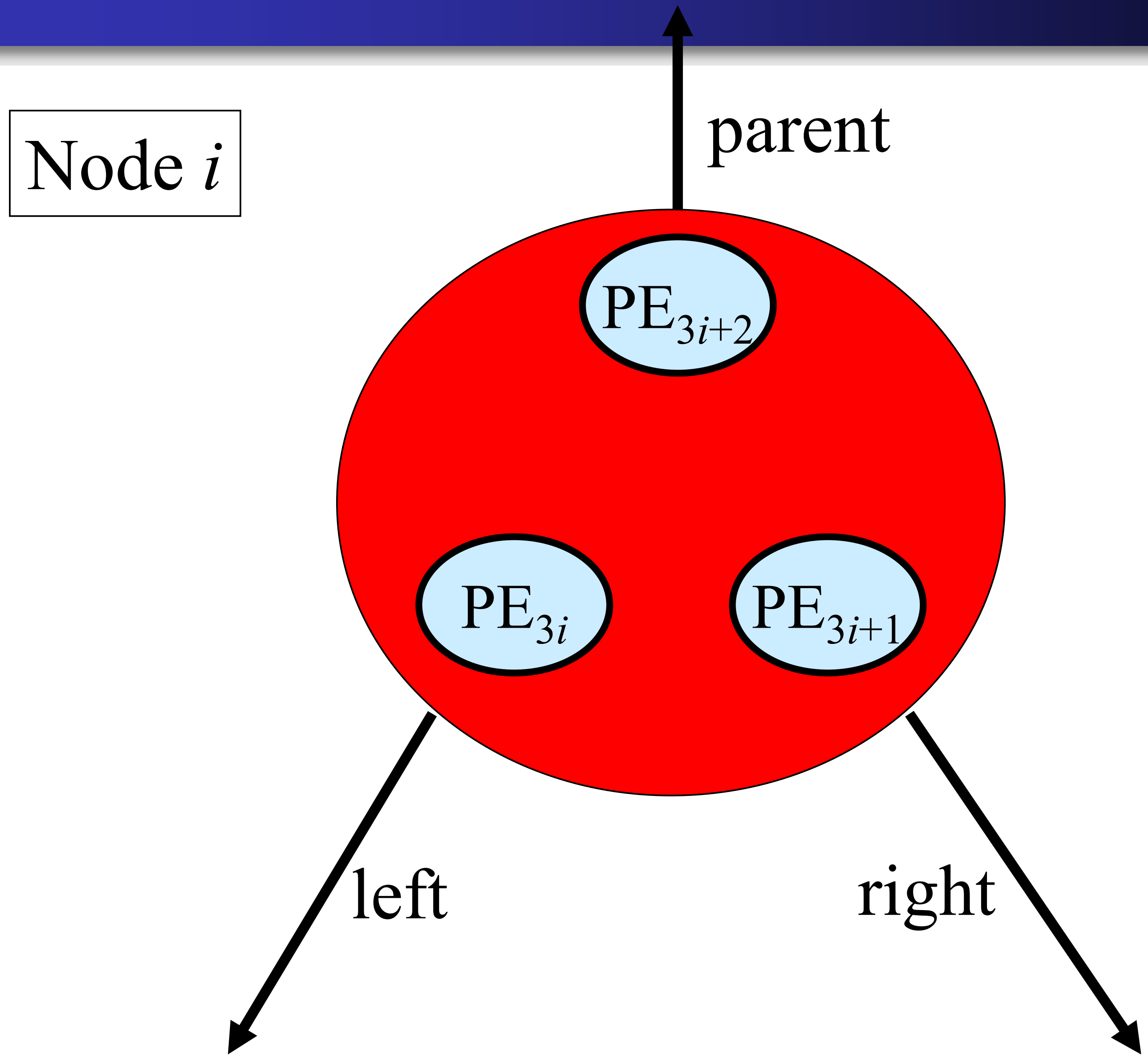
2	3	4	5	6	7	9	10

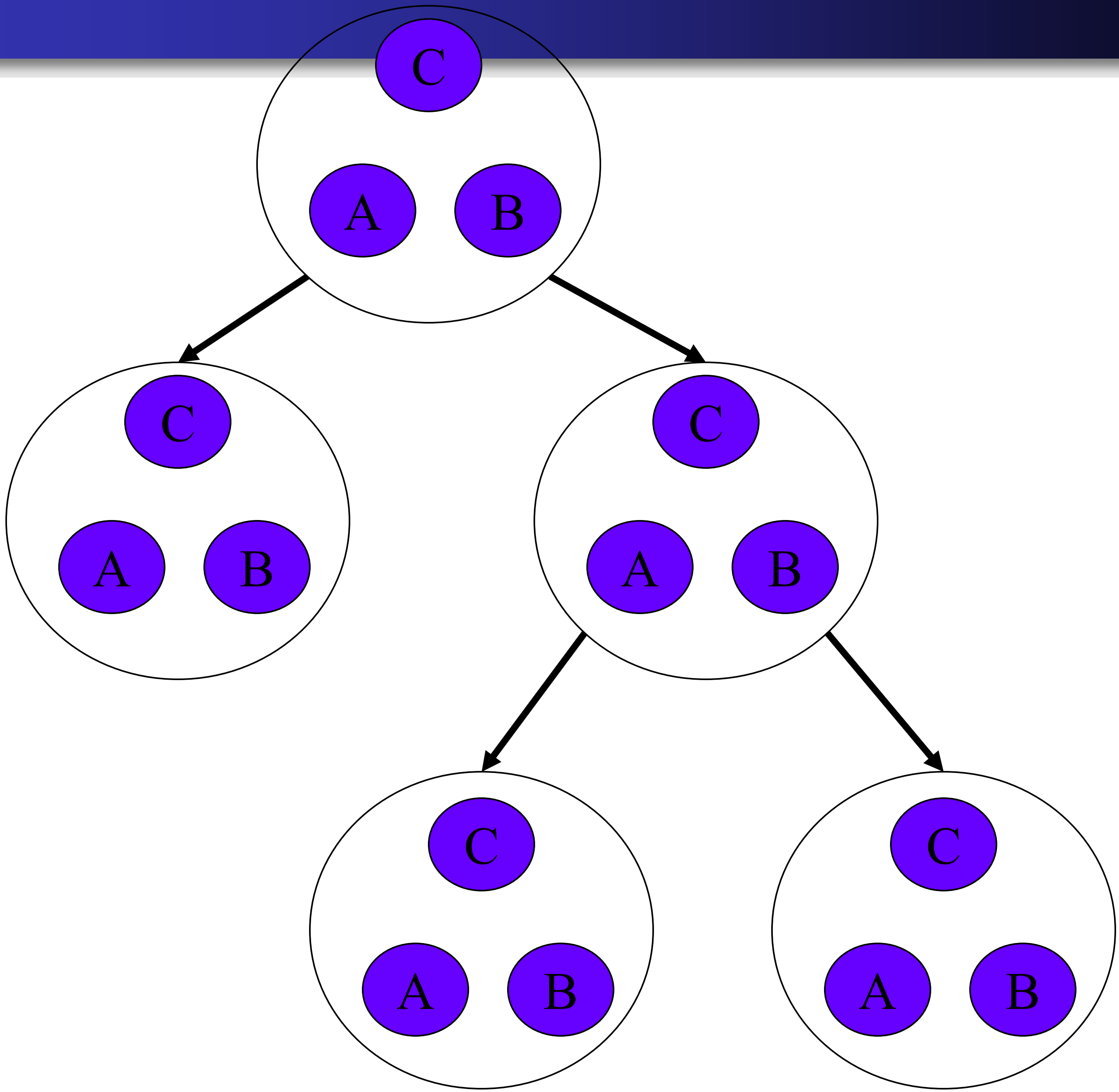
Euler tour technique

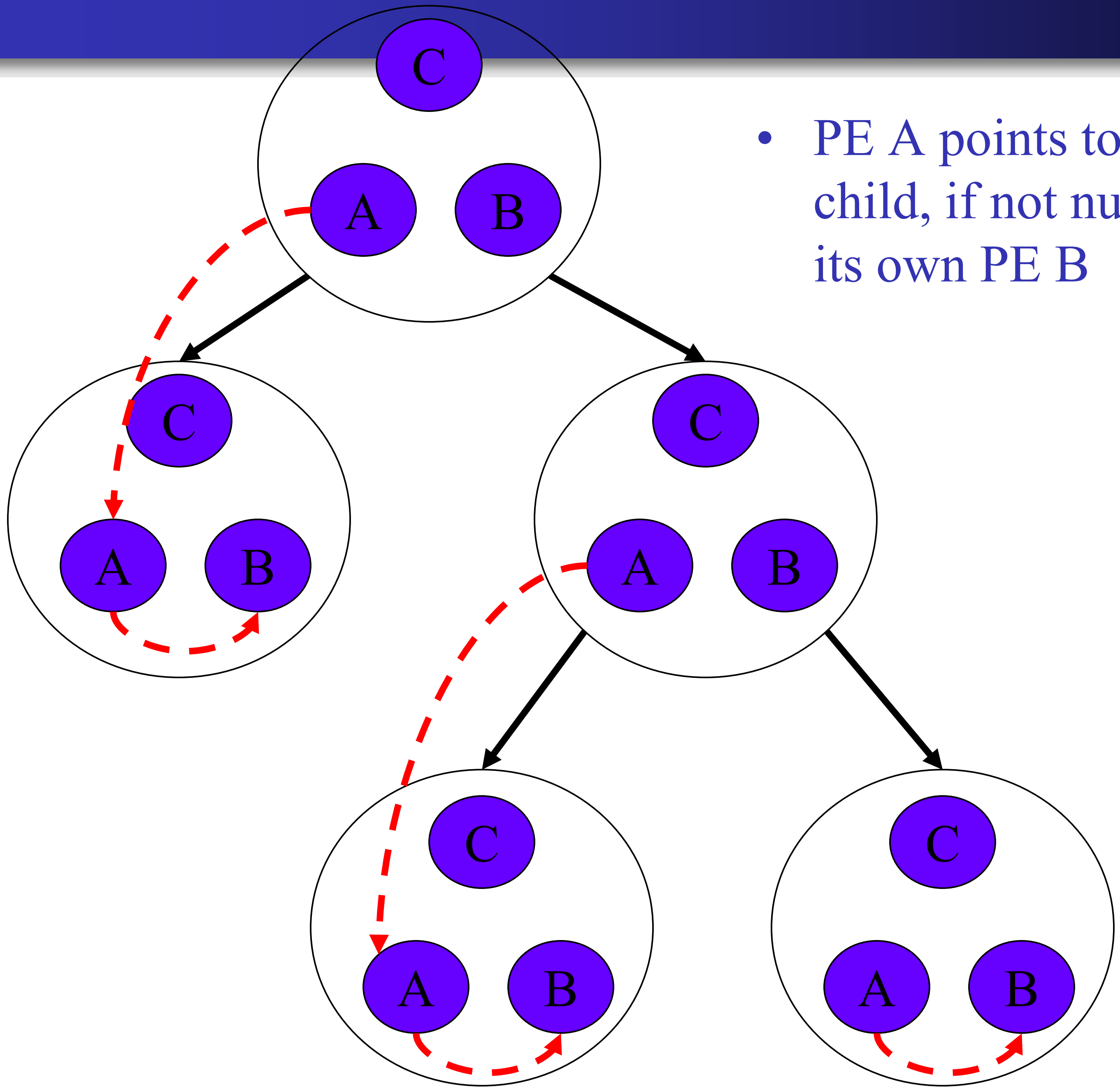
- **Problem:** Compute the depth of every node in a given binary tree of n nodes
- Sequential RAM: $O(n)$ time
- PRAM: one processor per node
- Obvious algorithm:
  - Start at root with depth 0
  - Propagate depths down one level at a time
- Time is $O(\text{depth}) = O(n)$; worst case
- $O(n^2)$ work; further, time no better than sequential
  - Can we do better?

Solution

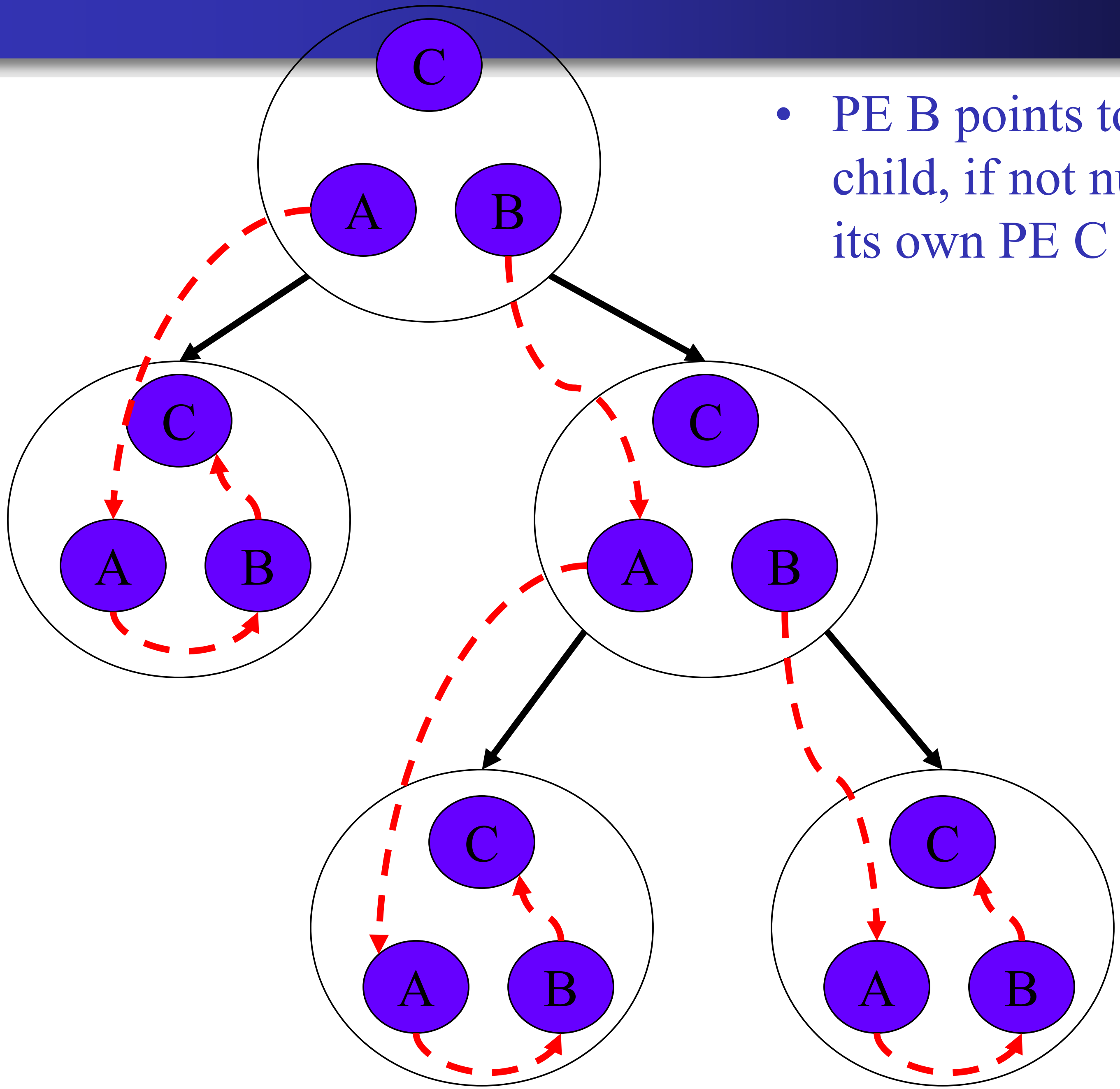
- Let each node have a *left*, *right* and *parent* pointer
- Assign a PE per pointer (*i.e.*, 3 PEs per node)
- Need to be able to map between processors and nodes easily, so say node i has processor $3i$, $3i+1$ and $3i+2$


Node i
parent
PE_{3i+2}
PE_{3i}
PE_{3i+1}
left
right

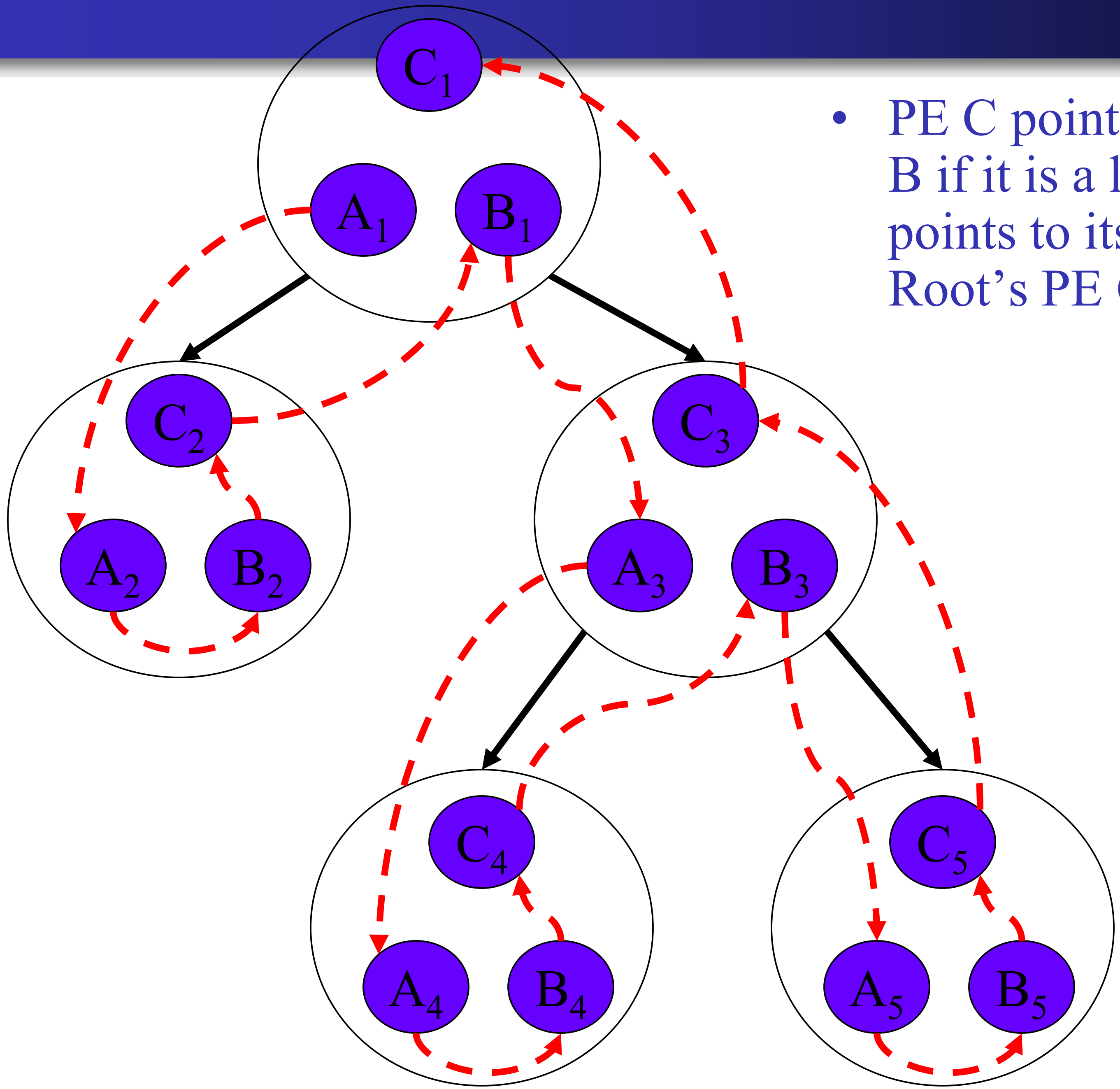

C
A
B
C
A
B
C
A
B
C
A
B
C
A
B



- PE A points to PE A of its left child, if not null, else points to its own PE B



- PE B points to PE A of its right child, if not null, else points to its own PE C

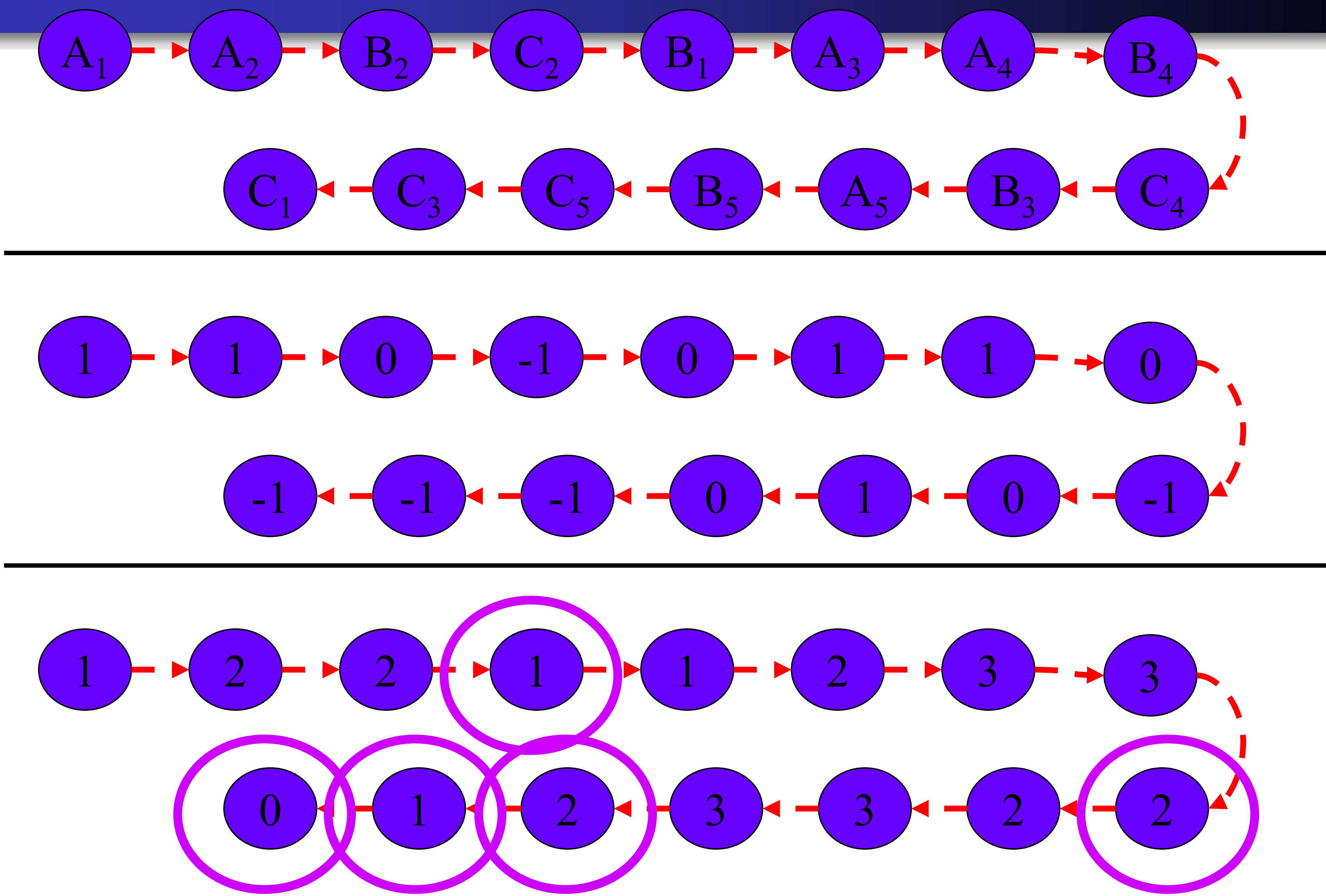


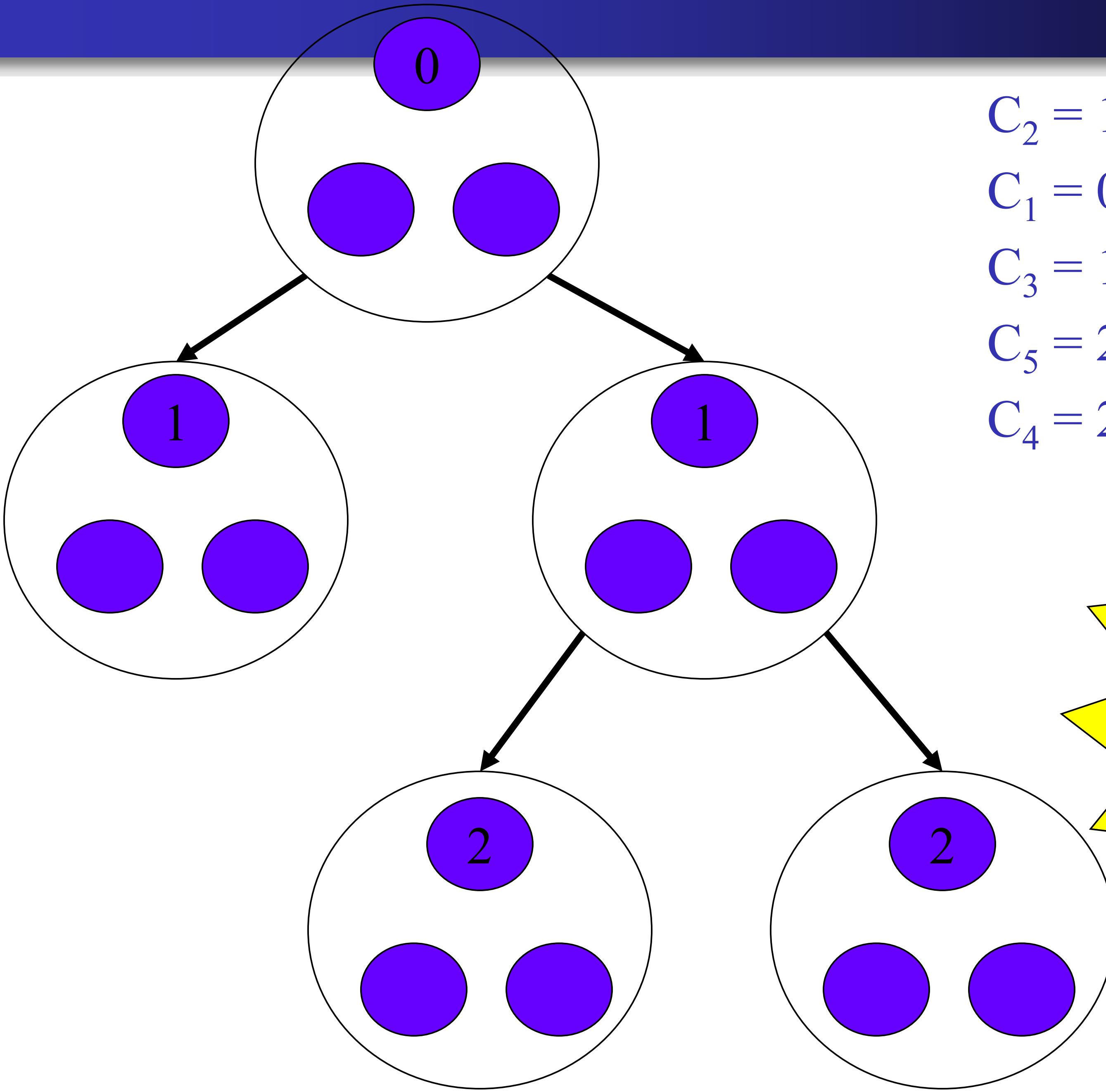
- PE C points to its parent's PE B if it is a left child, else points to its parent's PE C. Root's PE C points to null.

Euler tour (cont.)

- Give each PE A a value of 1
- Give each PE B a value of 0
- Give each PE C a value of –1
- Then do parallel prefix sum on the linked-list
- PE C now holds the depth of each node

- $O(\log n)$ time regardless of tree height
- Requires $O(n)$ processors.


A1 A2 B2 C2 B1 A3 A4 B4
C1 C3 C5 B5 A5 B3 C4
1 1 0 -1 0 1 1 0
-1 -1 -1 0 1 0 -1
1 2 2 1 1 2 3 3
0 1 2 3 3 2 2



$C_2 = 1$

$C_1 = 0$

$C_3 = 1$

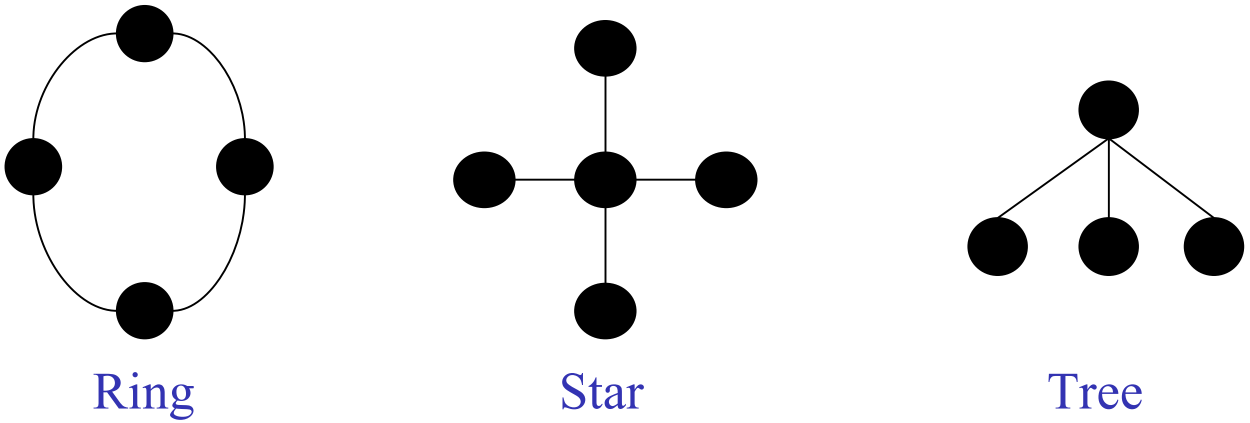
$C_5 = 2$

$C_4 = 2$

WOW!

Distributed Systems

- In a distributed system, multiple CPUs must communicate via explicit message passing,
  - rather than through a common memory as in PRAM
  - Examples: LANs, like the student labs, and the Internet
  - Programming languages need constructs to handle message passing (*e.g.*, sockets)
  - There is no global state over the whole system.
    - Each machine has its own state which it can send to the others.
  - Node failures are probable (unlike RAM and PRAM)
- System topology can be in a fixed structure, e.g., ring, star, tree, or unstructured



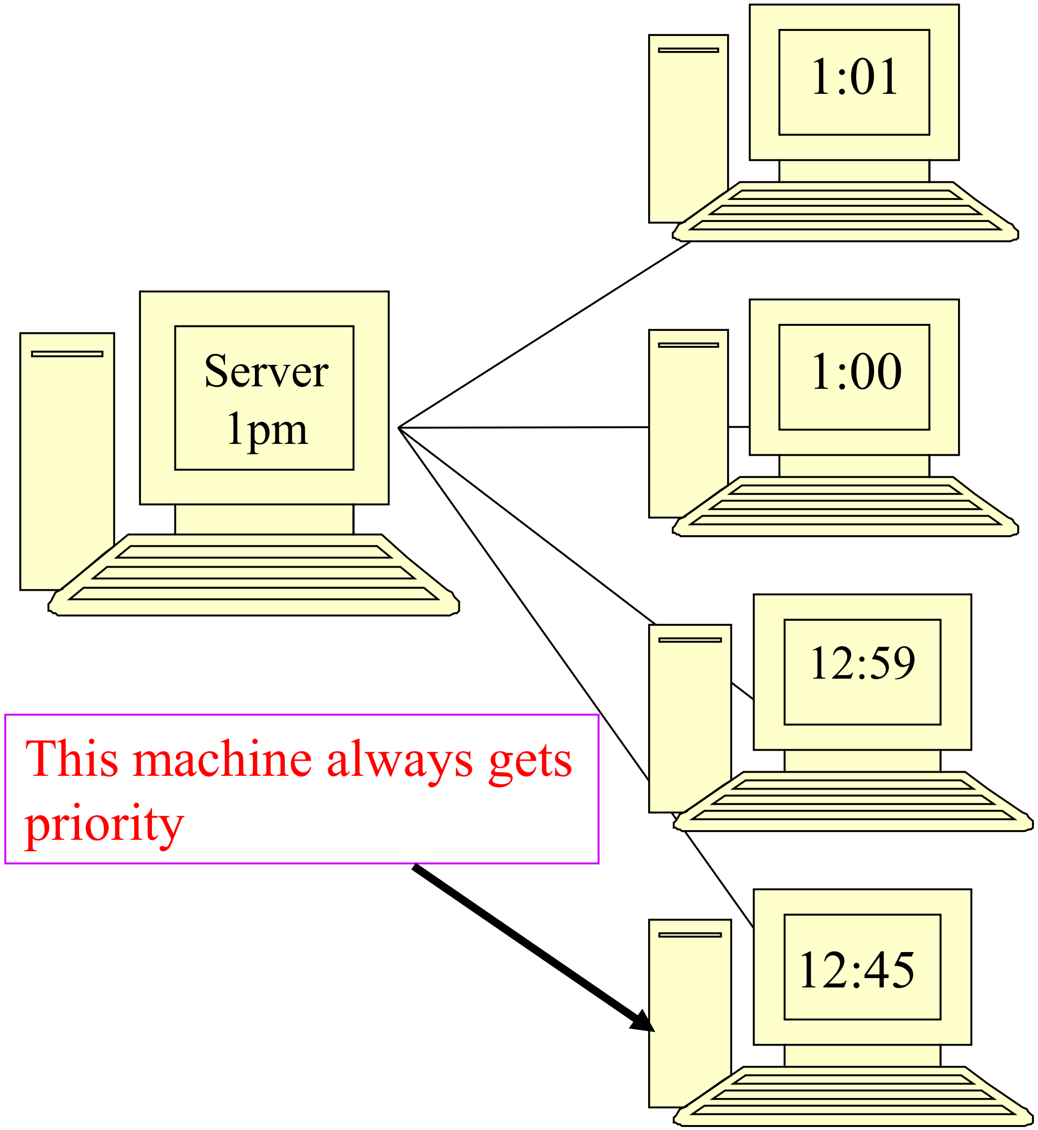
Distributed data and control

Issues:

- Do all nodes have a copy of the data?
- How are copies of the data kept consistent?
- Does each node have a small section of the data?
- What happens to data integrity if one or more nodes fail?

- What happens if one machine in charge of an algorithm fails?
  - Note, these concepts are true of any system based on messaging (*e.g.*, OO databases) and not just computer networks.

Example – File Server



- Say a file server gives jobs priorities based on the time they were sent from the client
  - If a request has an earlier time, it is processed first
- When a client sends a file request, they include their *local clock time* in the message
- **Is this a fair system?**

Example – File Server (a Solution – Time Stamping)

- Each node i maintains a logical clock, h_i
  - h_i is an integer beginning at zero
- Each message sent from node i is stamped with h_i+1
- When node i receives a message (m, h_j), it updates its clock to $h_i = \max(h_i, h_j) + 1$

Example - Web Cache

- Web cache or proxy server is a part of the network that serves HTTP requests on behalf of an actual Web server
  - The cache contains disk storage to keep the recently requested files by the clients

How does web cache work?

- A client browser is configured such that each HTTP request is directed to the cache
  - The browser creates a TCP connection to the cache, and sends HTTP request for the object to the cache
  - If the cache has a copy of the requested object, it sends the object as an HTTP response to the browser
  - If the cache does not have the requested object, it creates a TCP connection to the actual web server, and sends an HTTP request for the object to the server
    - The actual server will send the object to the cache using its HTTP response
    - Receiving the object, the web cache stores a copy of the object in its storage, and sends the object to the client browser as its HTTP response
    - Thus, a web cache acts as a server and also a client

Web Cache (cont.)

- Benefits of using Web cache
  - Each client request can be served faster, i.e., it reduces response time
    - Note that the bandwidth/speed between the client and web cache is expected to be better than between the client and the actual server
  - The organization's traffic to the Internet can be significantly reduced
    - Thus, it reduces its operational cost as well as upgrading cost for higher bandwidth facility
  - It reduces web traffic in the Internet, which is good for the Internet community in general
- Who maintains the web caches?
  - A web cache can be installed by an ISP, e.g., in a university
  - All client browsers within the university are configured such that each HTTP request is directed to the cache

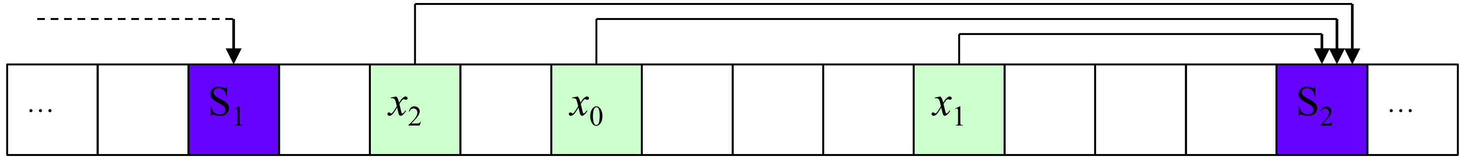
Web Cache (cont.)

- Where to store the shared cache?
  - In a special computer → expensive
  - Spread the cache over multiple computers, but …
- How to locate the cache among the multiple computers?
  - **Goal:** given an object / url, find the computer that stores the cache for the object.
- A hash table maps an object x onto a node n, i.e.,
  - n = hash (x), meaning object x is located at node n
  - A user that needs an object x finds its location n by computing hash (x)
  - Can use a table with fixed / limited sized of N, i.e., n = hash (x) MOD N; However …
    - When there are more than N nodes that can be used to put objects, not all nodes can be used
    - When N is larger than the largest possible number of nodes, hash (x) MOD N may refer to a non-existing node
    - Standard hashing scheme requires a fixed sized table
      - Not practical for applications in which nodes can *join* and *leave* frequently
      - Adding a new server needs to move almost all objects; on average only $1/n$ fraction of objects don't move

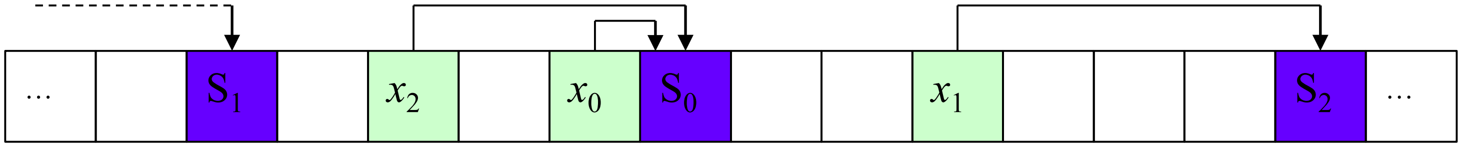
Consistent Hashing

How to map an **object** x_i to a **node / server** s_j using consistent hashing?

- It maps a set of objects x_i and the *names* / *IP addresses* s_j uniformly onto a large *ID space*
  - Given the IP address of a node, assign the node with m-bit identifier, i.e., **node_ID = hash (IP_address s_j)**
  - Given an object / key, assign each object / key with m-bit identifier, i.e., **object_ID = hash (object_name x_i)**
- Object x is stored in its *successor* server s that minimizes hash(s) $\geq$ hash (x)
- For n servers, there are n segments; each server is responsible for all objects in its segment, e.g., S_2 is responsible for objects $\{x_0, x_1, x_2\}$
  - With a good hash function, each server is responsible for exactly $1/n$ fraction of the total number of objects.
  - When a server s joins, move only the references to the objects that are mapped to server s
  - When a server s leaves, moves only the references in s to its *successor* server



A new server S_0 joins:



52

Consistent Hashing (cont.)

- Use a hash function, e.g., SHA-1 to generate each identifier for objects and servers
- The value of m must be large enough to avoid *hash collision*
  - For SHA-1, $m = 160$ bits
    - The ID space is extremely large, i.e., 2^{160}
    - It is unlikely that an object_ID collides with a node_ID
      - Nevertheless, in consistent hashing, **an object is placed in a node that has the *closest* node_ID to the object_ID** on the ID space
  - Advantage:
    - Objects are distributed almost evenly onto the nodes
    - Only a **small** number of objects have to move when there is a node joins and leaves
    - The searching time is $O(\log n)$ for n number of servers

Consistent Hashing (cont.)

- How to route a query for an object to the node responsible for the object?
  - i.e., how to find a successor node that is the **closest** to the ID of the object?

Solution-1: Use a hash table
  - Not good because hash values do not contain any order information for successors

Solution-2: Use a heap
  - Not good because heap maintains only partial ordering.

Solution-3: Use a balanced binary search tree
  - Route the query from node to node to find the necessary route / path to the destination node

- Consistent hashing has been used in:
  - Akamai – Content Delivery Network (CDN) provider, co-founded by Tom Leighton in 1998– one of the inventors of Consistent Hashing
  - Distributed storage, e.g., Amazon's Dynamo
  - Distributed Hash Table (DHT) used in P2P network
    - The hash table is distributed across all nodes in the networks

The End

- We have completed all lectures for this semester!
- NO lecture next week, but have unit review!