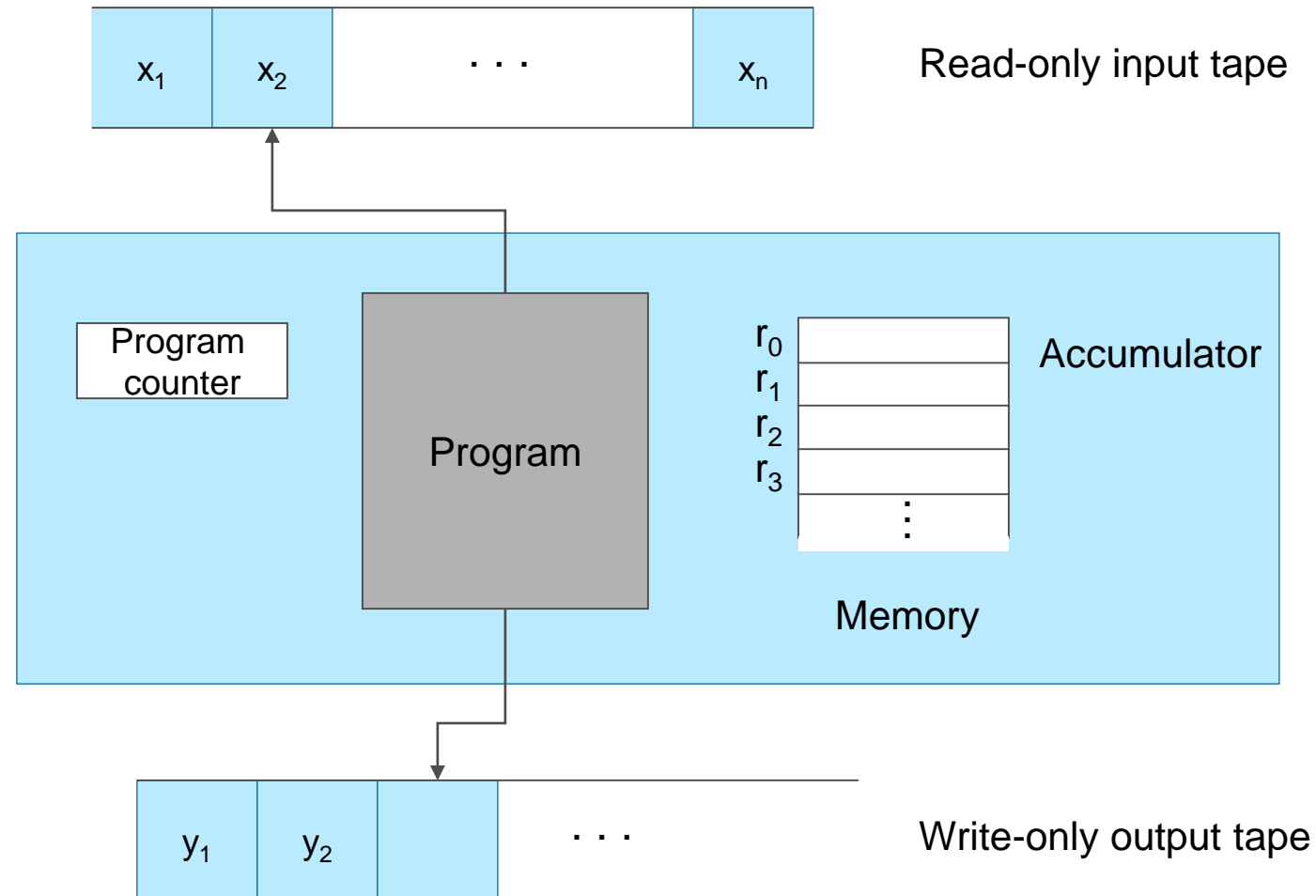


PRAM Algorithms

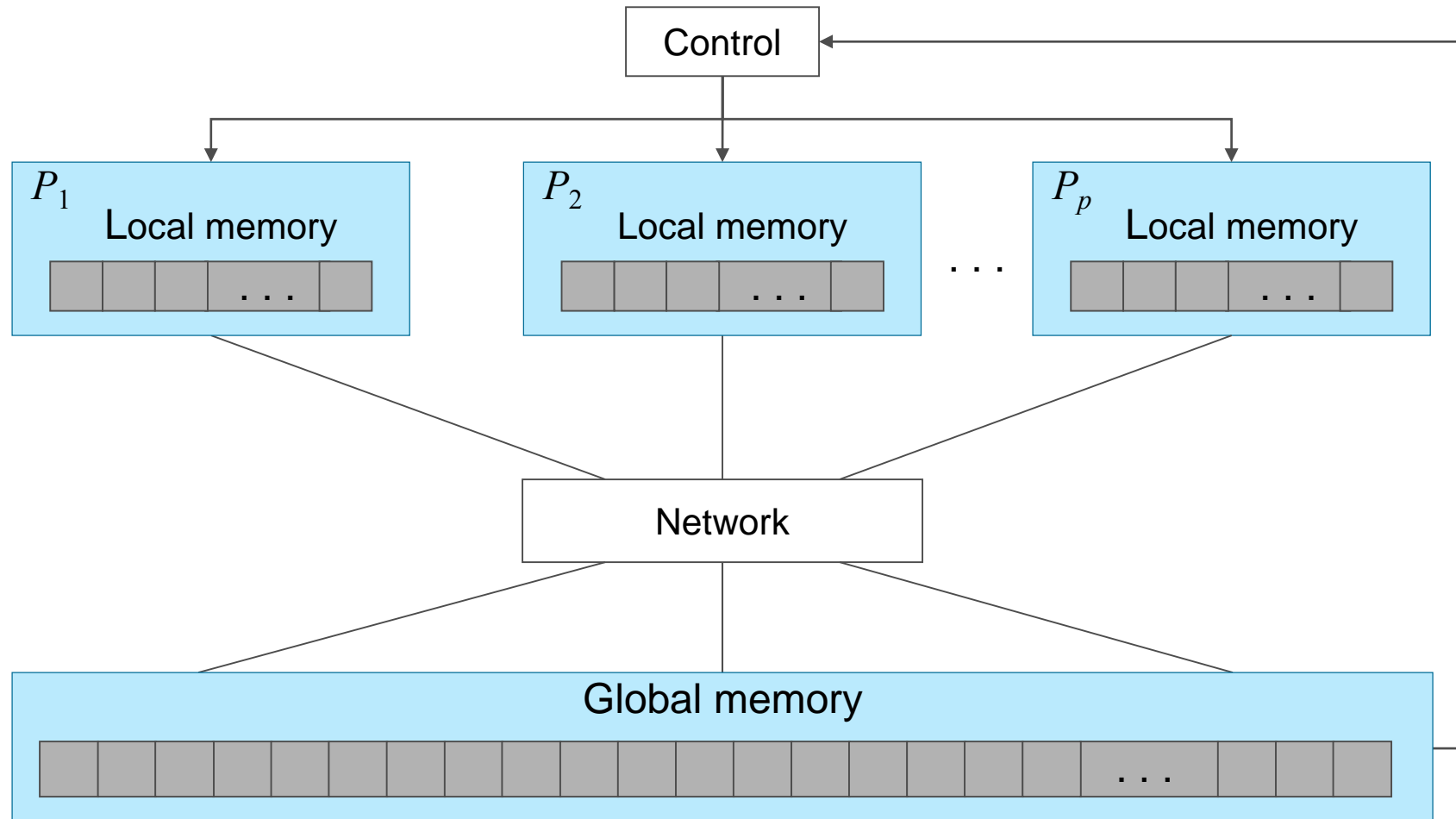


Algorithms and Data Structures 2, 340300
Lecture – 2023W
Univ.-Prof. Dr. Alois Ferscha, teaching@pervasive.jku.at

RAM Modell (Aho/Hopcroft/Ullmann 74)



PRAM Modell (Fortune/Willey 78)



PRAM Models

EREW (Exclusive Read Exclusive Write)

- read or write conflicts are not allowed

CREW (Common Read Exclusive Write)

- concurrent reading is allowed, i.e., multiple processors may read from same global memory location during the same instruction step, write conflicts are not allowed
- default PRAM model

CRCW (Common Read Common Write)

- concurrent reading and concurrent writing allowed
- variety of CRCW models exist with different policies for handling concurrent writes to same global address, e.g.

COMMON:

all processors concurrently writing into same global address must be writing same value

ARBITRARY:

if multiple processors concurrently write into same global address, one of the competing processors is arbitrarily chosen as “winner” and its value is written into the register

PRIORITY:

if multiple processors concurrently write into same global address, the processors with the lowest index succeeds in writing its value into the reg.

Relative Strength of PRAM Models

EREW is weakest model

- CREW can execute any EREW in same amount of time, concurrent read facility is simply not used
- CRCW can execute any EREW in same amount of time, concurrent read and write facilities are not used

PRIORITY CRCW PRAM is strongest model

- COMMON PRAM will execute with same complexity on ARBITRARY and PRIORITY PRAM
(choosing an arbitrary value or the value from the processor with lowest index gives the same result if all values are identical)
- ARBITRARY PRAM will execute with same time complexity as PRIORITY PRAM, because if an algorithm executes correctly when an arbitrary processor is chosen as the winner, the processor with the lowest index is as reasonable an alternative as any other

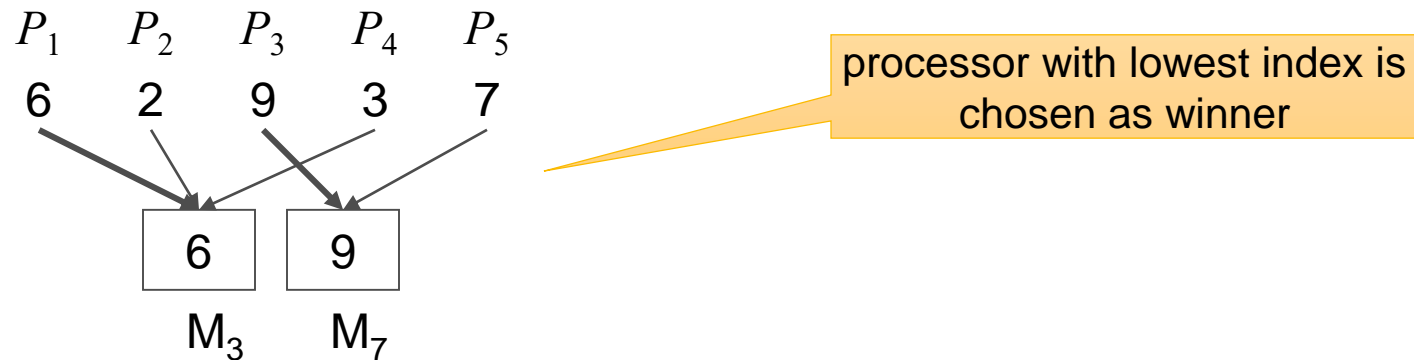
PRIORITY PRAM vs. EREW PRAM

Theorem ([Eckstein 1979], [Vishkin 1983])

- A p -processor CRCW PRIORITY PRAM can be simulated by a p -processor EREW PRAM with the time complexity increased by a factor of $\Theta(\log p)$

Proof

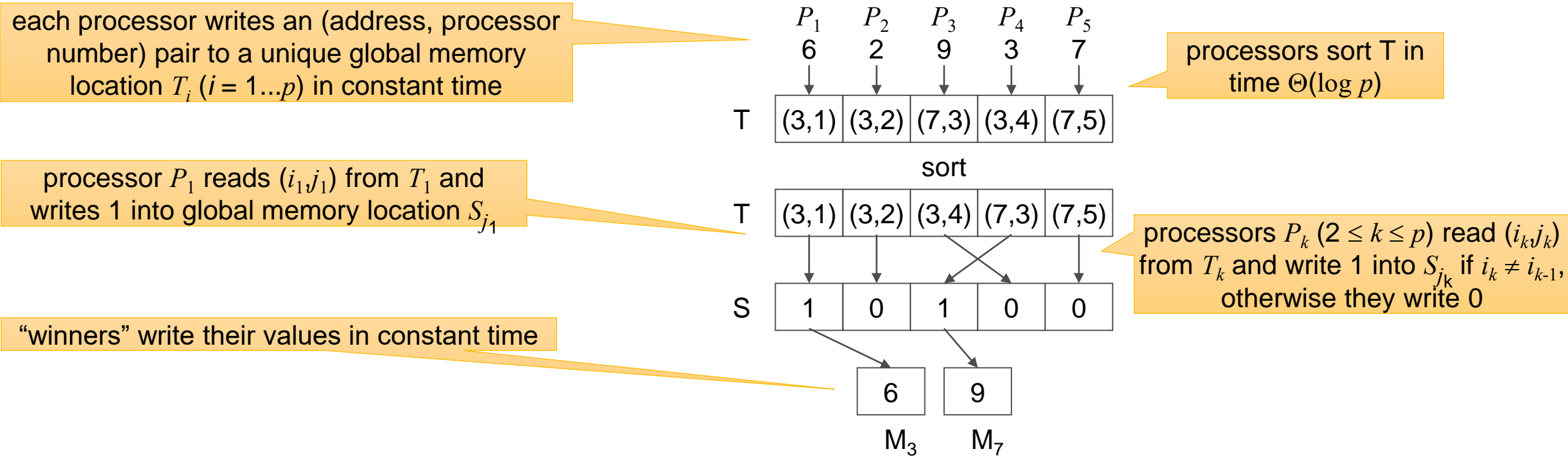
- concurrent access in PRIORITY PRAM takes constant time



PRIORITY PRAM vs. EREW PRAM

Proof (contd.)

- concurrent access can be simulated on a p -processor EREW PRAM in $\Theta(\log p)$ as follows



PRAM Algorithms

begin with one processor being active

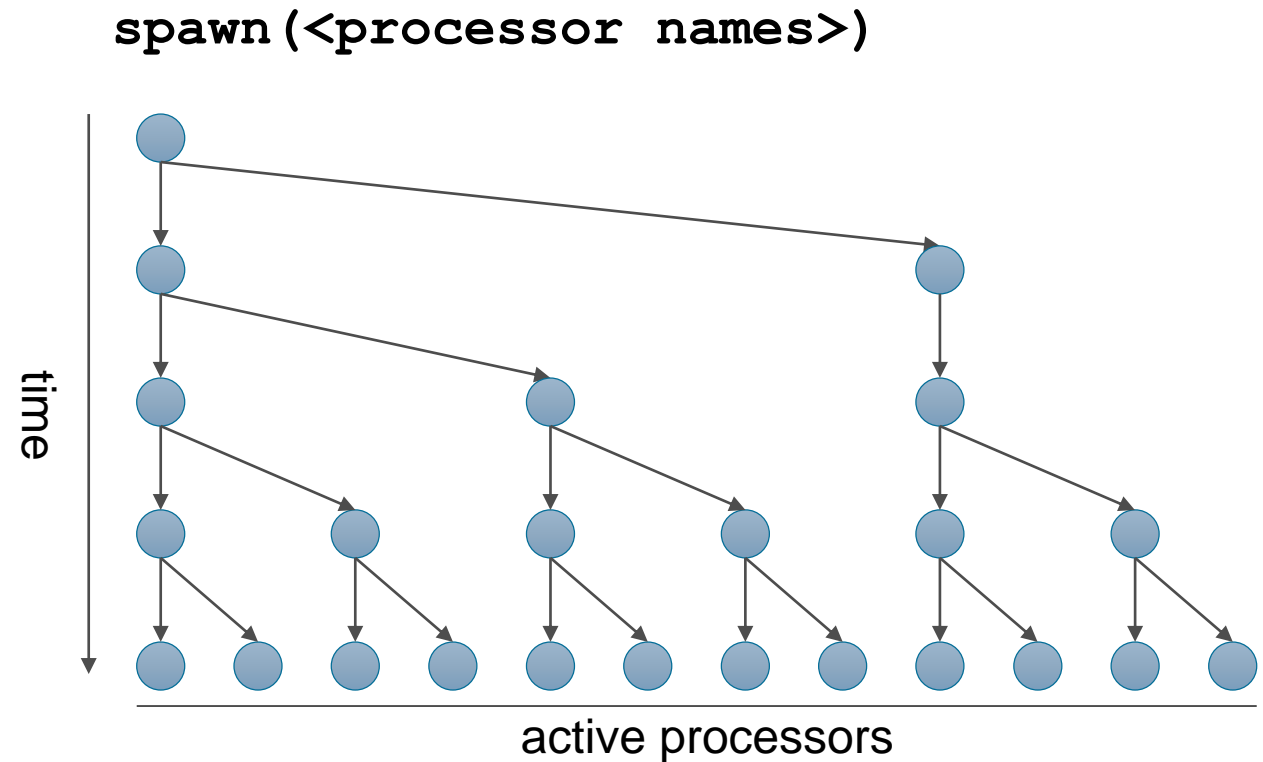
have two phases

- **activation** of processors
- performing **computation in parallel**

Activation of processors takes $\lceil \log p \rceil$ steps

concurrent execution

```
for all <processor list>  
do <statement list>  
endfor
```



Parallel Reduction

bottom-up dataflow in a binary tree from leaves to root

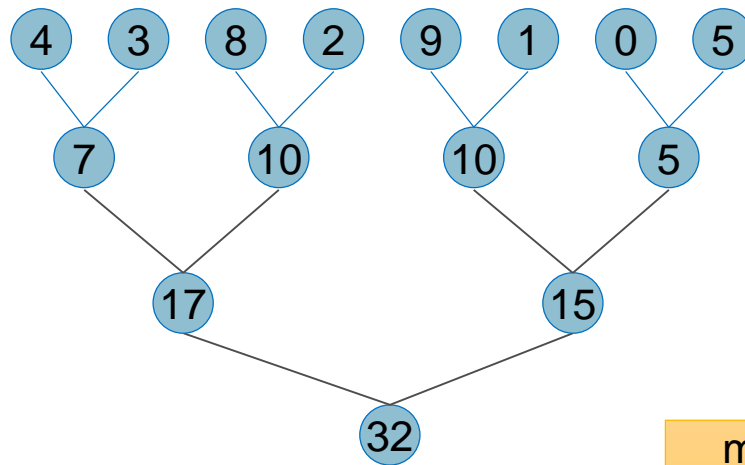
given a set of n values a_1, a_2, \dots, a_n and an **associative binary operator** \oplus ,
then reduction is the process of computing $a_1 \oplus a_2 \oplus \dots \oplus a_n$

Parallel Reduction

bottom-up dataflow in a binary tree from leaves to root

given a set of n values a_1, a_2, \dots, a_n and an **associative binary operator** \oplus ,
then reduction is the process of computing $a_1 \oplus a_2 \oplus \dots \oplus a_n$

Example: **Parallel Summation** using $\lfloor n/2 \rfloor$ processors



masking

SUM (EREW PRAM)

Global Variables: $n, A[0 \dots (n - 1)], j$

begin

spawn ($P_0, P_1, \dots, P_{\lfloor n/2 \rfloor}$)

for all P_i where $0 \leq i \leq \lfloor n/2 \rfloor - 1$ do

for $j \leftarrow 0$ to $\lceil \log n \rceil - 1$ do

if $i \bmod 2^j = 0$ and $2i + 2^j < n$ then

$A[2i] \leftarrow A[2i] + A[2i + 2^j]$

endif

endfor

endfor

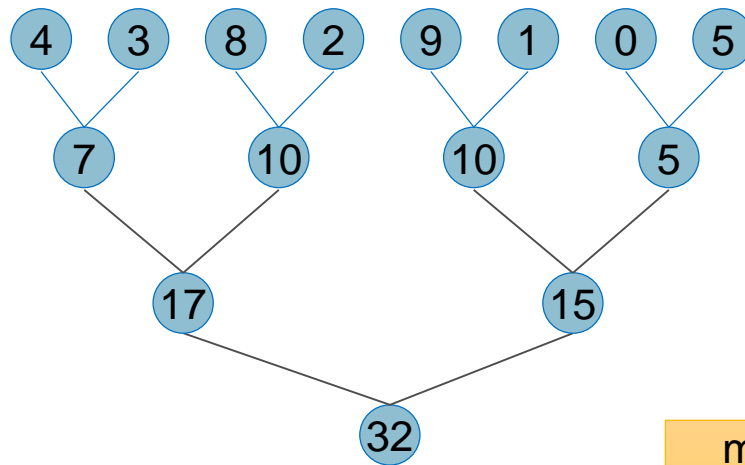
end

Parallel Reduction

bottom-up dataflow in a binary tree from leaves to root

given a set of n values a_1, a_2, \dots, a_n and an **associative binary operator** \oplus ,
then reduction is the process of computing $a_1 \oplus a_2 \oplus \dots \oplus a_n$

Example: Parallel Summation using $\lfloor n/2 \rfloor$ processors



Complexity

- spawn
- loop is executed
- total:

$$\lceil \log \lfloor n/2 \rfloor \rceil$$

$\lceil \log n \rceil$ times (constant time in each iteration)

$$\lceil \log \lfloor n/2 \rfloor \rceil + \lceil \log n \rceil = \Theta(\log n)$$

masking

SUM (EREW PRAM)

Global Variables: $n, A[0 \dots (n - 1)], j$

begin

spawn ($P_0, P_1, \dots, P_{\lfloor n/2 \rfloor}$)

for all P_i where $0 \leq i \leq \lfloor n/2 \rfloor - 1$ do

for $j \leftarrow 0$ to $\lceil \log n \rceil - 1$ do

if $i \bmod 2^j = 0$ and $2i + 2^j < n$ then

$A[2i] \leftarrow A[2i] + A[2i + 2^j]$

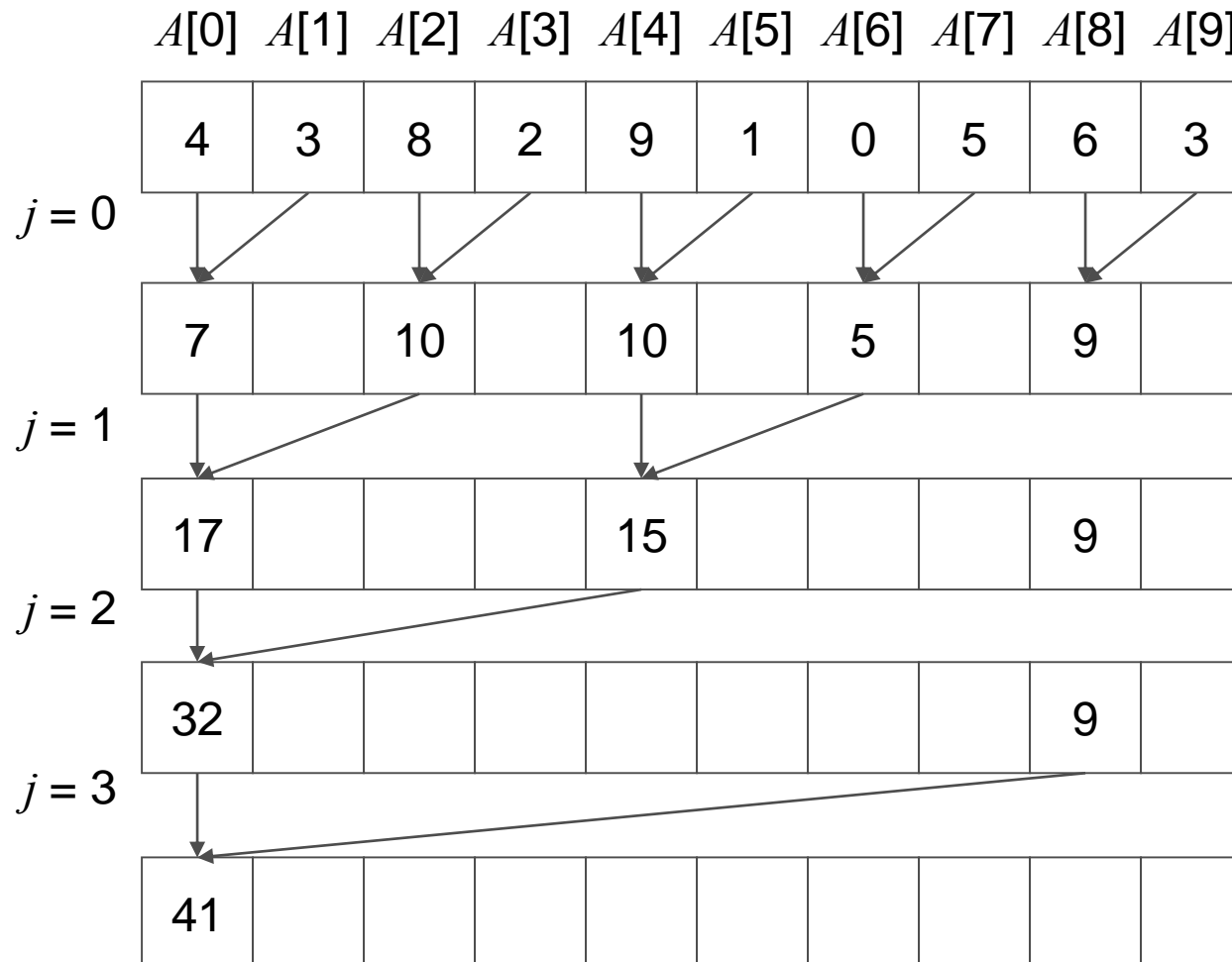
endif

endfor

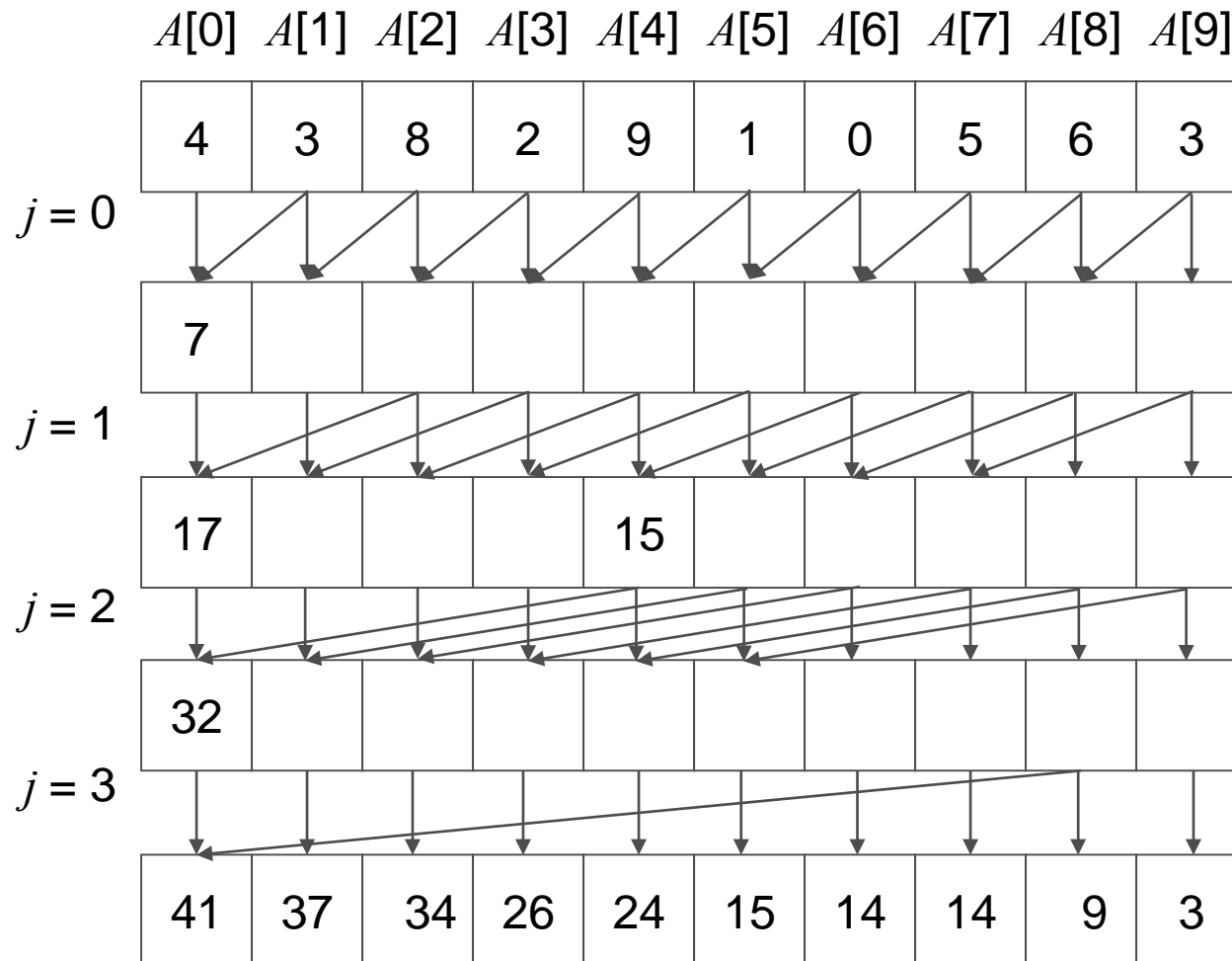
endfor

end

Parallel Summation Example



Parallel Prefix Summation Example



Prefix Sums

Packing elements is one application of prefix sums

given:

array A contains both uppercase and lowercase letters

goal:

pack uppercase letters into beginning of A

procedure

1. create array T which contains 1 for every uppercase letter and 0 for every lowercase letter
2. compute prefix sums of T
3. for every uppercase letter in A corresponding entry in T gives index in packed array

Array A

A	b	C	D	e	F	g	h	I
---	---	---	---	---	---	---	---	---

Array T

1	0	1	1	0	1	0	0	1
---	---	---	---	---	---	---	---	---

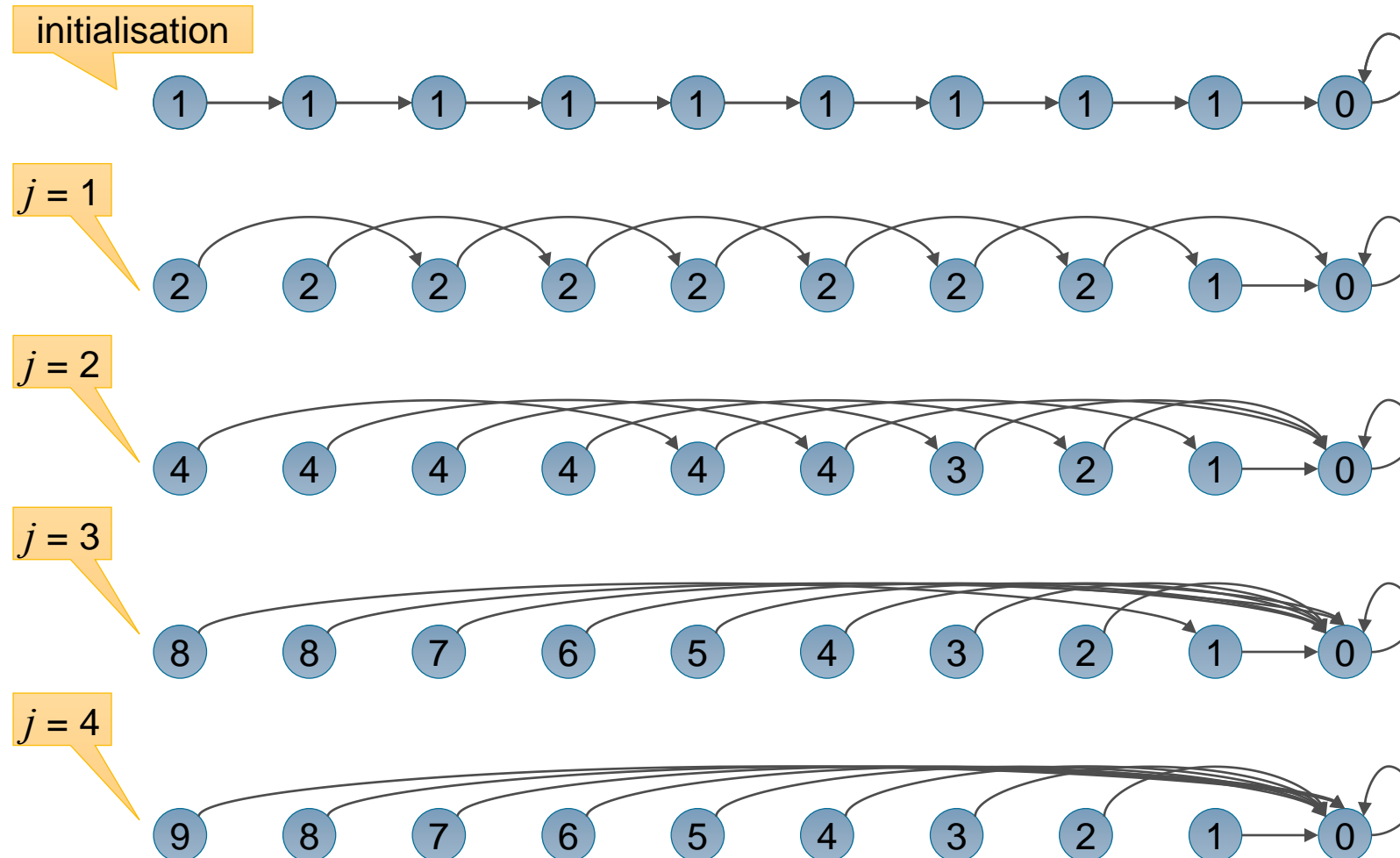
T after computing prefix sums

1	1	2	3	3	4	4	4	5
---	---	---	---	---	---	---	---	---

A packed

A	C	D	F	I				
---	---	---	---	---	--	--	--	--

List Ranking (Suffix Sums)



List Ranking (Suffix Sums)

determine position (distance from end) of each item in an n -element linked list

```
LIST.RANKING (CREW PRAM)
Global variables:  $n$ ,  $\text{position}[0 \dots (n-1)]$ ,  $\text{next}[0 \dots (n-1)]$ ,  $j$ 
begin
  spawn ( $P_0, P_1, \dots, P_{n-1}$ )
  for all  $P_i$  where  $0 \leq i \leq n-1$  do
    if  $\text{next}[i] = i$ 
      then  $\text{position}[i] \leftarrow 0$ 
      else  $\text{position}[i] \leftarrow 1$ 
    endif
    for  $j \leftarrow 1$  to  $\lceil \log n \rceil$  do
       $\text{position}[i] \leftarrow \text{position}[i] + \text{position}[\text{next}[i]]$ 
       $\text{next}[i] \leftarrow \text{next}[\text{next}[i]]$ 
    endfor
  endfor
end
```

initialisation

iteration

computational complexity: requires $\lceil \log n \rceil$ steps

Preorder Tree Traversal

labels are assigned to nodes

```
PREORDER:TRAVERSAL (nodeptr)
begin
  if nodeptr ≠ null then
    nodecount ← nodecount + 1
    nodeptr.label ← nodecount
    PREORDER.TRAVERSAL(nodeptr.left)
    PREORDER.TRAVERSAL(nodeptr.right)
  endif
end
```

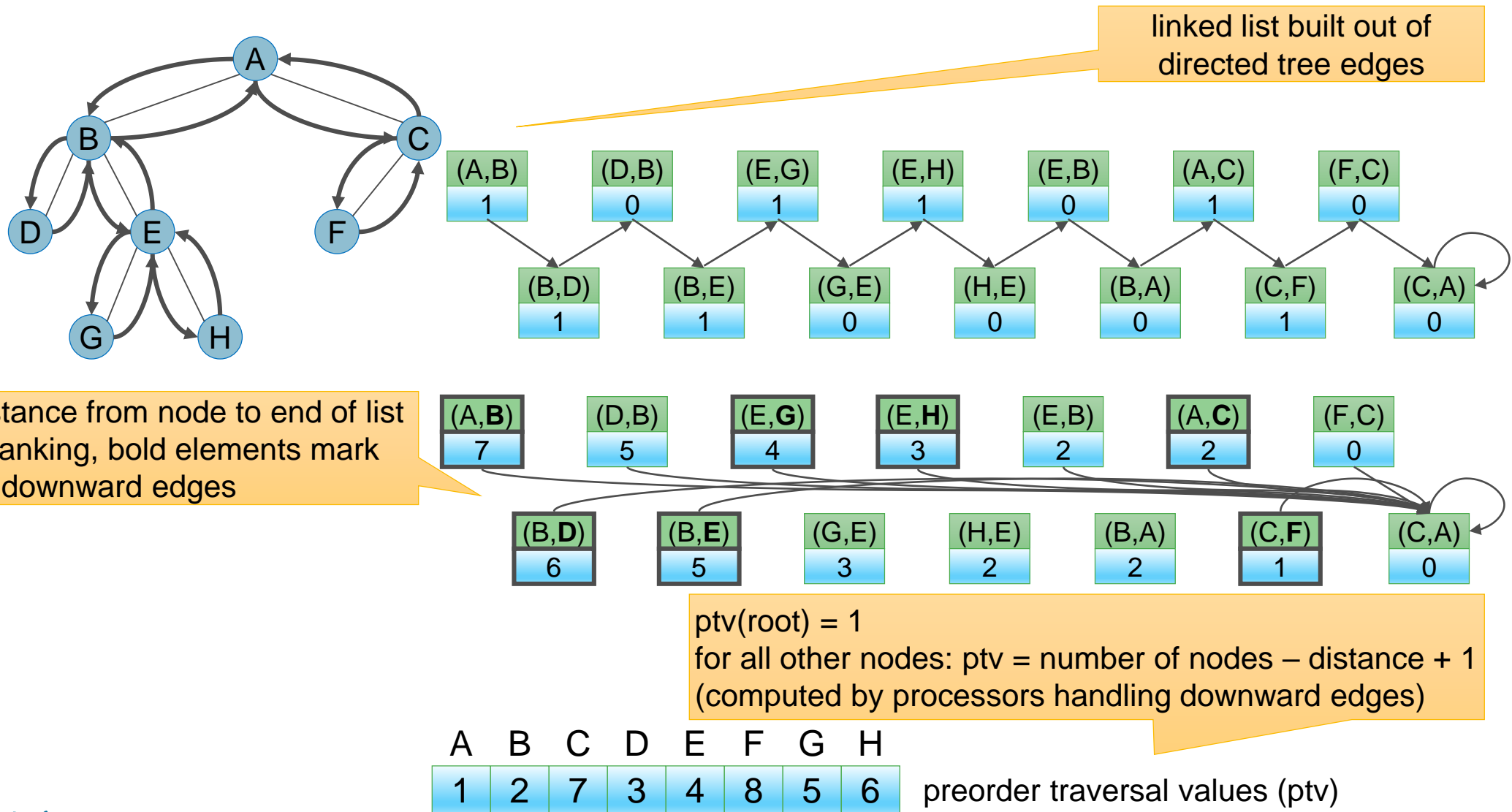
seems to be **inherently sequential**,

because labels in right subtree cannot be assigned until number of nodes in left subtree is known

but consider edges: **each edge is visited twice**, one heading **down** from parent to child and one going **back up**

- if edges are divided into two directed edges, tree traversal corresponds to traversal of a single linked list (which is parallel, see prev. example)

Preorder Tree Traversal



Preorder Tree Traversal

Tree has n nodes and $(n - 1)$ edges and is stored in a data structure that stores for every node (i) its **parent**, (ii) immediate **right sibling**, (iii) and **left child**

	A	B	C	D	E	F	G	H
parent	null	A	A	B	B	C	E	E
sibling	null	C	null	E	null	null	H	null
child	B	D	F	null	G	null	null	null

algorithm needs $2(n - 1)$ processors to manipulate $2(n - 1)$ elements of linked list

Computational steps:

- construct linked list of edges (i, j) : each processor holding an edge (i, j) computes successor edge as follows
 - if $parent[i] = j$ mark edge as upward edge and set position value to 0
 - if $sibling[i] \neq null$ next edge is $(j, sibling[i])$
 - else if $parent[j] \neq null$ next edge is $(j, parent[j])$
 - else edge is last in preorder traversal, set preorder value of j to 1 (j is root)
 - if $parent[i] \neq j$ mark edge as downward edge and set position value to 1
 - if $child[j] = null$ j is leave and next edge is (j, i)
 - if $child[j] \neq null$ next edge is $(j, child[j])$
- do the pointer jumping loop (see list ranking)
- every processor holding a downward edge computes preorder traversal value by subtracting its position from number of nodes +1

Preorder Tree Traversal

PREORDER.TREE.TRAVERSAL (CREW PRAM):

```
begin
  spawn (set of all P(i, j) where (i, j) is an edge)
  for all P(i, j) where (i, j) is an edge do
    // Put the edges into a linked list
    if parent[i] = j then
      if sibling[i] ≠ null then
        succ[(i, j)] ← (j, sibling[i])
      else if parent[j] ≠ null then
        succ[(i, j)] ← (j, parent[j])
      else
        succ [(i, j)] ← (i, j)
        preorder[j] ← 1 {j is root of tree}
      endif
    else
      if child[j] ≠ null then succ[(i, j)] ← (j, child[j])
      else succ[(i, j)] ← (j, i)
      endif
    endif
  endif
  ...
```

Preorder Tree Traversal (contd.)

```
...
// Number of edges of the successor list
if parent[i] = j then position[(i, j)] ← 0
else position[(i, j)] ← 1
endif

// Perform suffix sum on successor list
for k ← 1 to  $\lceil \log(2(n - 1)) \rceil$  do
    position[(i, j)] ← position[(i, j)] + position[succ[(i, j)]]
    succ[(i, j)] ← succ[succ[(i, j)]]
endfor

// Assign preorder values
if i = parent[j] then preorder[j] ← n + 1 - position[(i, j)]
endif
endfor
end
```

Parallel Merge Sort

many **PRAM** algorithms achieve **low time complexity** by performing **more operations** than the original (optimal) RAM algorithm (parallel merge sort is an example)

optimal RAM merge sort requires at most $(n - 1)$ comparisons to merge two sorted lists of $n / 2$ elements, complexity is $\Theta(n)$

PRAM algorithm can perform the task in $\Theta(\log n)$ using **n processors**

- each processor holds one element of the **two lists** to be merged
- each processor determines its **search range**
- each processor performs **binary search** (complexity is $\Theta(\log n)$) to determine correct **position** in **merged list**

total number of operations in PRAM algorithm

- **n processors** performing **binary search**, therefore $\Theta(n \log n)$

Parallel Merge Sort

MERGE.LISTS (CREW PRAM):

```
begin
  spawn (P1, P2, ..., Pn)
  for all Pi where 1 ≤ i ≤ n do

    // Each processor sets bounds for binary search
    if i ≤ n / 2 then
      low ← (n / 2) + 1
      high ← n
    else
      low ← 1
      high ← n / 2
    endif
  ...
```

Parallel Merge Sort (contd.)

```
...  
// Each processor performs binary search  
x ← A[i]  
repeat  
    index ← [(low + high) / 2]  
    if x < A[index] then  
        high ← index - 1  
    else  
        low ← index + 1  
    endif  
until low > high  
  
// Put value in correct position on merged list  
A[high + i - n / 2] ← x  
endfor  
end
```


Graph Colouring



1852

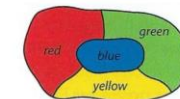
Guthrie's map-color problem

Can every map be colored with four colors so that neighboring countries are colored differently?

We certainly need four for *some* maps



Francis Guthrie



four neighboring countries



... but not here

1878 Arthur Cayley

presented to *London Mathematical Society*

1879 Alfred Kempe

first (claimed) Proof

1890 Percy Heawood

disproved first Proof

1960 Heinrich Heesch

„Computer-Proof“

1976 Kenneth Appel / Wolfgang Haken

University of Illinois



Graph Colouring

Create a **processor** for every **possible colouring** and **check** on each processor, **if** assigned colouring is **valid**

given: graph of n vertices in an $n \times n$ **adjacency matrix** and a positive **constant c** (denoting the number of **colours**)

goal: determine validity of each colouring by assigning **1 (valid)** or **0 (invalid)** to each possible colouring stored in n -dimensional candidate array

needs $\Theta(c^n)$ processors where processor $P(i_0, i_1, \dots, i_{n-1})$ holds a colouring of vertex 0 with colour i_0 , vertex 1 with colour i_1 , ..., vertex $n - 1$ with colour i_{n-1} .

Procedure:

- check, whether a given colouring is valid (i.e. no two adjacent vertices are coloured in the same colour)
 - each processor **initially** sets its value in the n -dimensional **candidate** array **to 1**
 - if $A[j, k] = 1$ and $i_j = i_k$ then colouring is **not valid** and value in candidate array is **set to 0**
 - after at most **n^2 comparisons** the validity for one colouring is checked
- summing up values in candidate array to determine total number of valid colourings requires $\Theta(\log c^n)$
- total time complexity is $\Theta(n \cdot \log c + n^2) = \Theta(n^2)$ (because $c < n$)

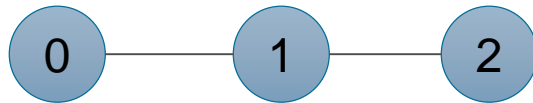
Graph Colouring

GRAPH.COLOURING (CREW PRAM):

```
begin
  spawn (P(i0, i1, ..., in-1)) where 0 ≤ iv < c for 0 ≤ v ≤ n
  for all P(i0, i1, ..., in-1) where 0 ≤ iv < c for 0 ≤ v < n do
    candidate[i0, i1, ..., in-1] ← 1
    for j ← 0 to n - 1 do
      for k ← 0 to n - 1 do
        if A[j][k] and ij = ik then
          candidate[i0, i1, ..., in] ← 0
        endif
      endfor
    endfor
    valid ← Σ candidate {Sum of all elements of candidate}
  endfor
  if valid > 0 then print „Valid colouring exists“
  else print „Valid colouring does not exist“
  endif
end
```

Graph Colouring

Example: graph with 3 vertices using 2 colours



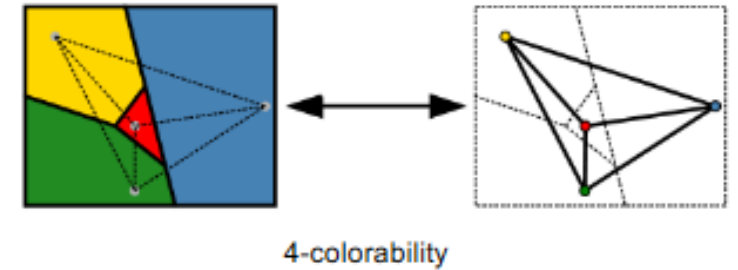
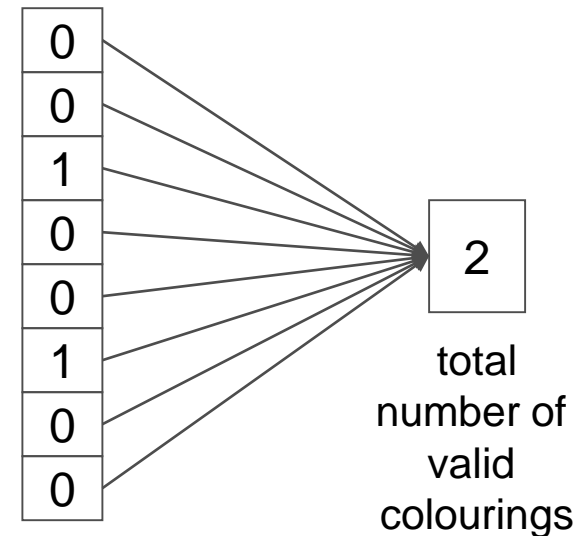
possible
colourings

0,0,0
0,0,1
0,1,0
0,1,1
1,0,0
1,0,1
1,1,0
1,1,1

candidate array
initial values

1
1
1
1
1
1
1
1

candidate array
after checking



Cost Optimal Parallel Algorithms

Definition

- **cost** = parallel time complexity \times the number of processors
- Example:
the parallel reduction algorithm has time complexity $O(\log n)$ and uses $O(n)$ processors, **cost** is therefore $O(n \log n)$

Definition

- A cost optimal parallel algorithm is an algorithm on which the cost is in the same complexity class as an optimal sequential algorithm.
- Example:
the parallel reduction algorithm is not cost optimal, because an optimal **sequential** algorithm has complexity $O(n)$

Cost Optimality

PRAM Sum

	Operations	Processors	Time	Cost
SUM _n RAM	$n - 1$	1	$n - 1$	
	$\Theta(n)$	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$
SUM _n PRAM	$n - 1$	$n / 2$	$\lceil \log n \rceil$	
	$\Theta(n)$	$\Theta(n)$	$\Theta(\log n)$	$\Theta(n \cdot \log n)$
SUM _n PRAM _{opt}	$n - 1$	p	$\lceil \log n \rceil$	
	$\Theta(n)$???	$\Theta(\log n)$	$\Theta(n)$

-> to reduce cost, number of processors in parallel algorithm must be reduced !

Cost Optimal Parallel Algorithms

Note

- If the total number of operations performed by the parallel algorithm is of the same complexity class as an optimal sequential algorithm, then a **cost optimal algorithm does exist**.
- Example:
The parallel reduction algorithm performs $n - 1$ additions, which is the same as the optimal sequential algorithm, therefore a cost optimal variant of the parallel reduction algorithm exists

What would be the **minimum number of processors** needed to perform the $n - 1$ operations with logarithmic time complexity?

- want: $(n - 1) / p = \Theta(\log n)$
- hence: $n - 1 = p \cdot \Theta(\log n) \Rightarrow p = \Theta(n / \log n)$

How can we verify, that there is indeed a cost optimal parallel reduction algorithm with logarithmic time complexity?

Brent's Theorem

Theorem [Brent 1974]

- Given A , a **parallel algorithm** with computation **time t** , if parallel algorithm **A** performs **m computational operations**, then **p processors** can execute algorithm A in **time $t + (m - t) / p$** .

Proof:

- s_i ... number of computational operations performed by parallel algorithm A at step i where $1 \leq i \leq t$

- By definition
$$\sum_{i=1}^t s_i = m$$

- using p processors, step i can be simulated in time $\lceil s_i / p \rceil$
- total execution time of A using p processors

$$\sum_{i=1}^t \left\lceil \frac{s_i}{p} \right\rceil \leq \sum_{i=1}^t \frac{s_i + p - 1}{p} = \sum_{i=1}^t \frac{p}{p} + \sum_{i=1}^t \frac{s_i - 1}{p} = \frac{t + (m - t)}{p}$$

Brent's Theorem

Applied to parallel reduction algorithm:

Execution time using $\lfloor n / \log n \rfloor$ processors is

$$\frac{\lceil \log n \rceil}{t} + \frac{\frac{m}{n-1} - \frac{t}{\lceil \log n \rceil}}{\lfloor \frac{n}{\log n} \rfloor} = \Theta(\log n + \log n - \frac{\log n}{n} - \frac{\log^2 n}{n}) = \Theta(\log n)$$

reducing number of processors from n to $\lfloor n / \log n \rfloor$ does not change complexity of parallel algorithm!

PRAM can add n values in $\Theta(\log n)$ time using $\lfloor n / \log n \rfloor$ processors

- during first few iterations, each processor emulates a set of processors adding to the execution time, but not increasing overall complexity beyond $\Theta(\log n)$
- during later iterations (when no more than $\lfloor n / \log n \rfloor$ processors are needed, the algorithm is identical to the original PRAM algorithm

Brent's Theorem

Applied to prefix sums algorithm:

- number of operations performed in iteration i is $n - 2^i$
- total number of operations

$$\sum_{i=0}^{\lceil \log n \rceil - 1} (n - 2^i) = n * \log n - (2^{\lceil \log n \rceil} - 1) = \Theta(n * \log n)$$

- optimal sequential algorithm would require $n - 1$ operations to find all prefix sums, therefore given algorithm is not cost optimal
 - to reduce cost, number of processors in parallel algorithm must be reduced
 - but in prefix sums a simple emulation (as performed in previous example) would increase execution time and cost will remain the same
 - therefore a strategy must be elaborated in which processors work more efficiently

Cost Optimal Algorithm for Prefix Sums

Prefix sums of n values given p processors, where $p < n - 1$

- divide n values into p sets, each containing no more than $\lceil n / p \rceil$ values
- the **first $p - 1$ processors** use **best sequential** algorithm to compute **sums** of their $\lceil n / p \rceil$ values requiring $\lceil n / p \rceil - 1$ steps
- processors **compute prefix** sums of these **subtotals** in $\lceil \log(p - 1) \rceil$ time
- each processor **uses sum of values** in **lower blocks** to **compute prefix sums** of its block of values requiring $\lceil n / p \rceil$ additions

Complexity of this algorithm

- total execution time is $\lceil n / p \rceil - 1 + \lceil \log(p - 1) \rceil + \lceil n / p \rceil = \Theta(n / p + \log p)$

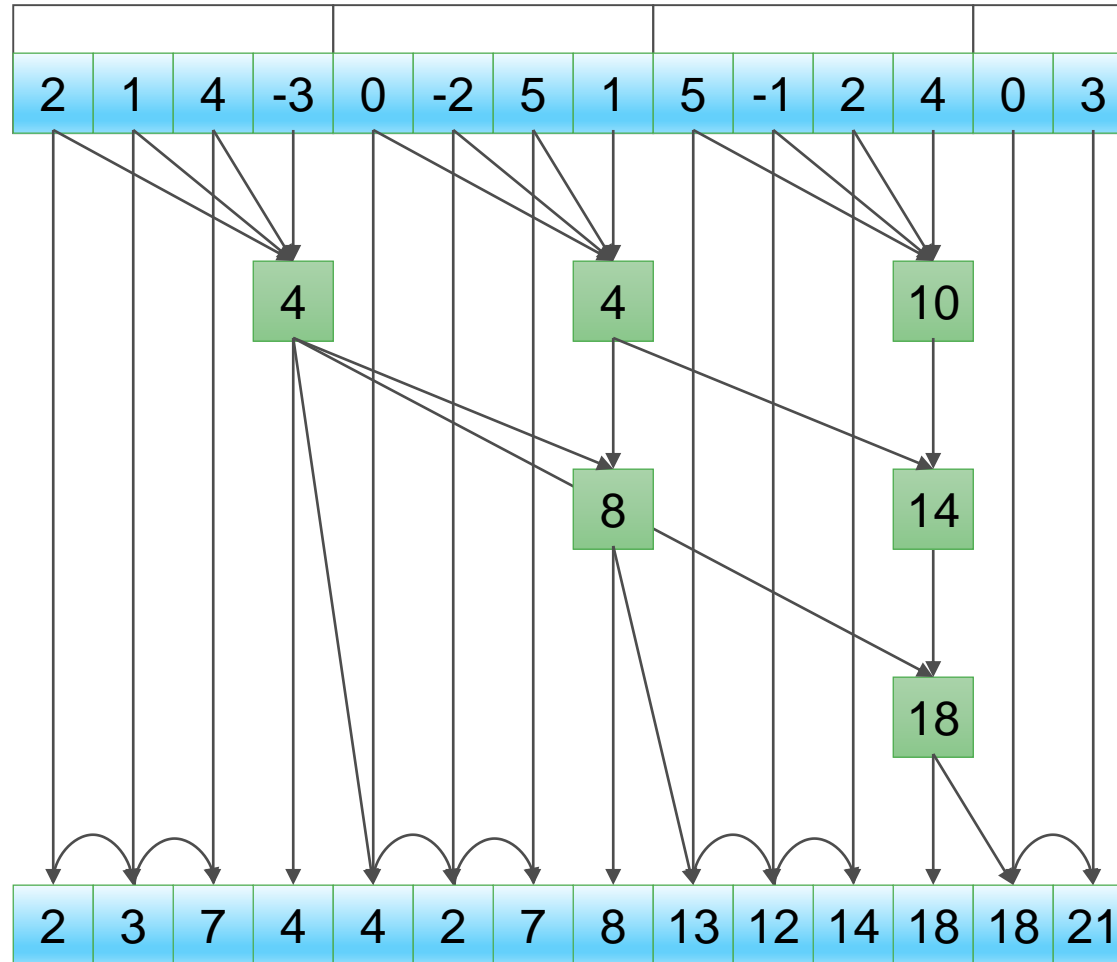
- applying Brent's Theorem:

$$\Theta \left(\underbrace{\left(\frac{n}{p} + \log p \right)}_t + \frac{\overbrace{n + p \log p}^m - \overbrace{\left(\frac{n}{p} + \log p \right)}^t}{\underbrace{p}_p} \right) = \Theta \left(\frac{n}{p} + \log p \right)$$

- cost = $\Theta(n + p \cdot \log p)$

which is cost optimal, if p is small enough, in particular execution time is minimized and algorithm has optimal cost when $p = \Omega(n / \log n)$

Cost Optimal Algorithm for Prefix Sums



Speedup Performance Laws

Let $W = \sum_{i=1}^m W_i$ denote the total amount of work in a parallel program, where W_i denotes the amount of work that can be performed in parallel using i processors (i is also called the degree of parallelism, DOP)

Let Δ denote the computing capacity (e.g. execution rate in MIPS) of a single processor

Then $T(n)$, the execution time of a parallel program using n processors, is given by

$$T(n) = \sum_{i=1}^m \frac{W_i}{i\Delta} \left\lceil \frac{i}{n} \right\rceil$$

The execution time on a single-processor is given by

$$T(1) = \sum_{i=1}^m \frac{W_i}{\Delta}$$

Fixed Load Speedup Factor (Amdahl)

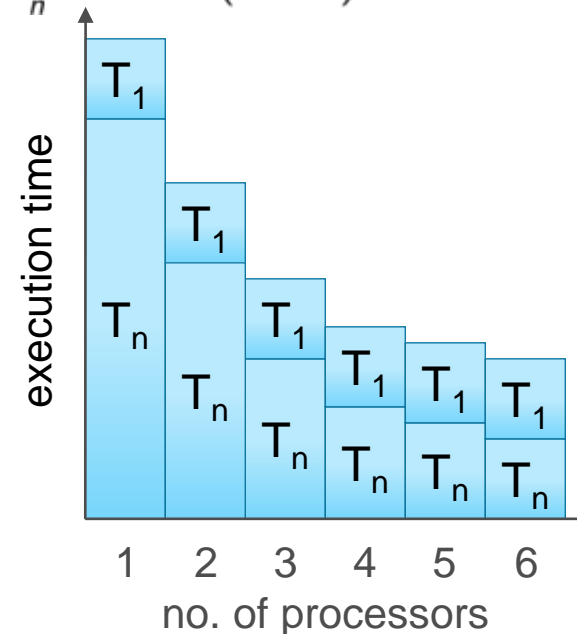
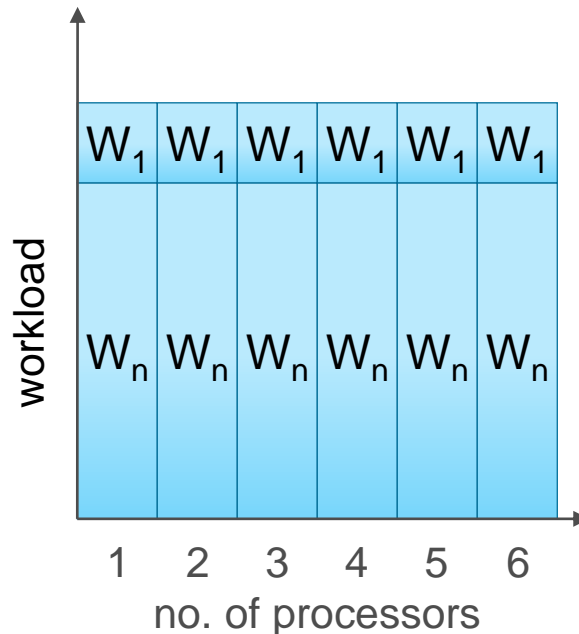
The fixed load speedup factor is defined as the ratio of $T(1)$ to $T(n)$

$$S_n = \frac{T(1)}{T(n)} = \frac{\sum_{i=1}^m W_i}{\sum_{i=1}^m \frac{W_i}{i\Delta} \lceil \frac{i}{n} \rceil}$$

Assuming that the parallel computer either operates in sequential mode ($i=1$) or in parallel mode ($i=n$), the speedup expression is simplified to

$$S_n = \frac{W_1 + W_n}{W_1 + \frac{W_n}{n}} = \frac{1}{\alpha + (1 - \alpha) * n}$$

normalisation!
 $W_i + W_n = \alpha + (1 - \alpha) = 1$
 Amdahl's Law

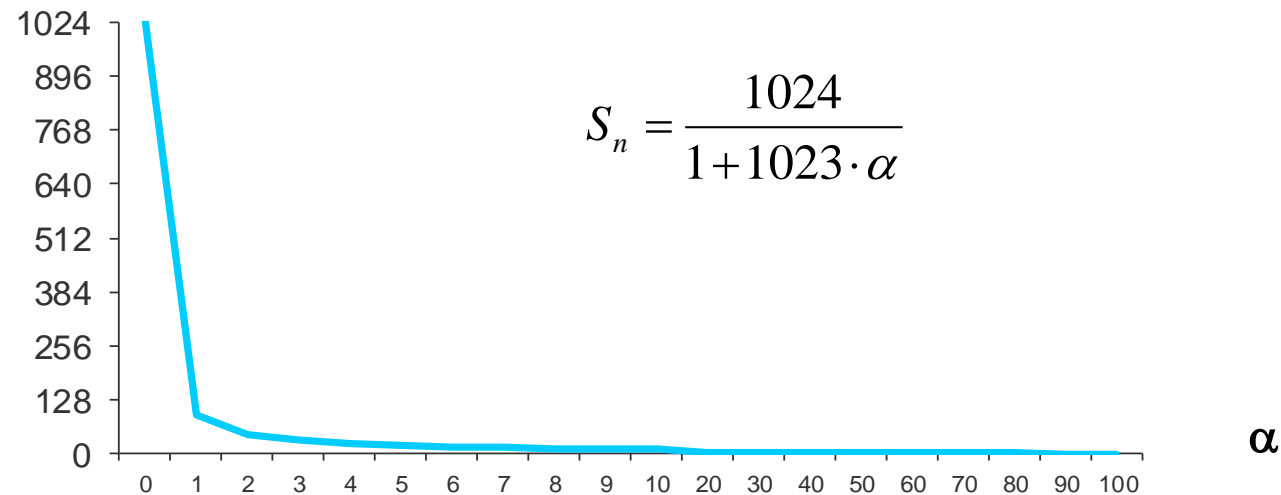


Fixed Load Speedup Factor

Sequential Bottleneck

- when the number of processors increases, speedup is bounded by $1/\alpha$, i.e. by the percentage of sequential code

Fixed Load Speedup on a 1024 processor machine as a function of the sequential fraction



Fixed Time Speedup (Gustafson)

Idea: solve the largest problem size possible on a larger machine with about the same execution time

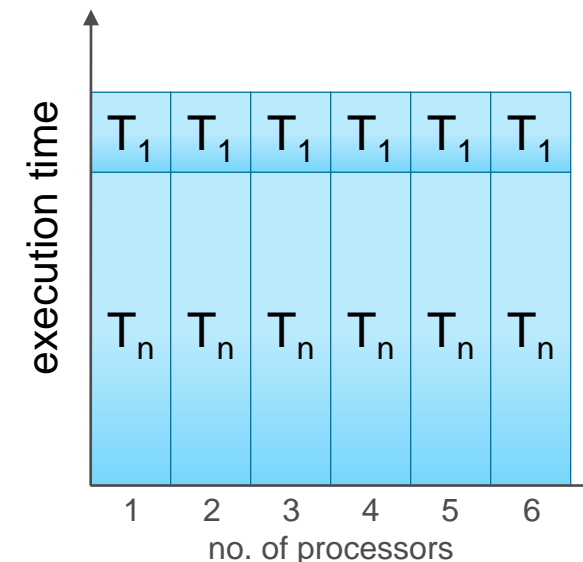
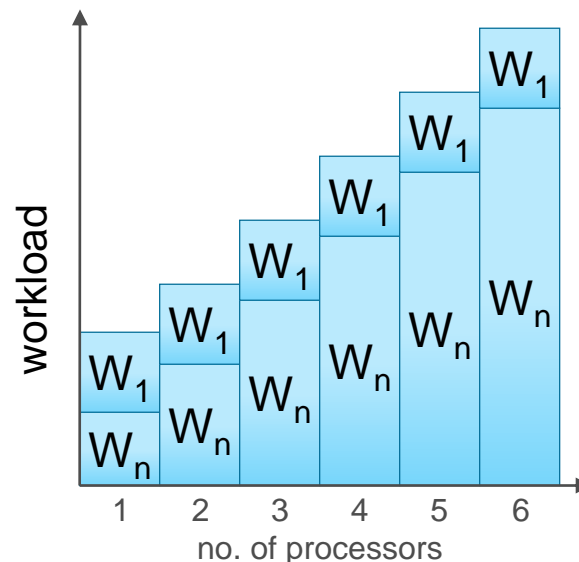
Let m' denote the maximum degree of parallelism in the scaled problem and W'_i the scaled workload with DOP i , then fixed time speedup is given by

$$S'_n = \frac{\sum_{i=1}^{m'} W'_i}{\sum_{W'_i} \lceil \frac{i}{n} \rceil} = \frac{\sum_{i=1}^{m'} W'_i}{\sum_{i=1}^m W_i}$$

parallel execution time of scaled problem corresponds to execution time of original problem, i.e. $T'(n) = T(1)$

$$S'_n = \frac{W'_1 + W'_n}{W_1 + W_n} = \frac{W_1 + n * W_n}{W_1 + W_n} = \alpha + n * (1 - \alpha) = n - (n - 1)\alpha$$

normalisation!
 $W_i + W_n = \alpha + (1 - \alpha) = 1$



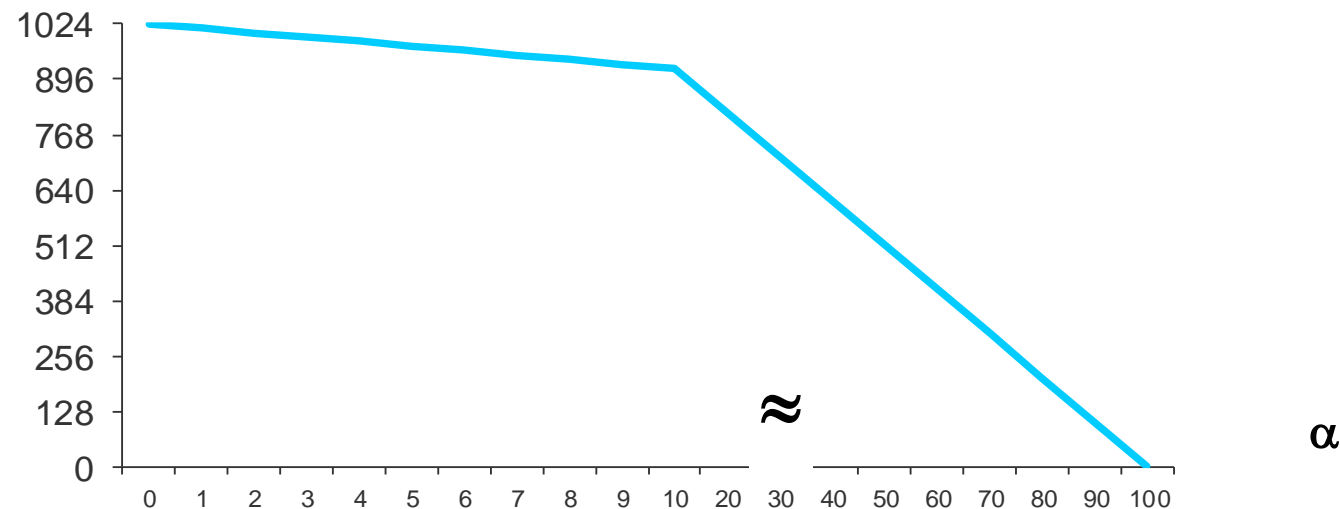
Fixed Time Speedup

sequential fraction is no longer a bottleneck

when the number of processors increases, linear speedup can be achieved, all processors are kept busy by increasing the problem size

Fixed Time Speedup on a 1024 processor machine as a function of the sequential fraction


$$S'_n = 1024 - 1023 \cdot \alpha$$



Fixed Memory Speedup

Idea: solve the largest problem size possible limited only by available memory capacity

Let m^* denote the maximum degree of parallelism in the memory-scaled problem and W_i^* the scaled workload with DOP i , then fixed memory speedup is given by

$$S_n^* = \frac{\sum_{i=1}^{m^*} W_i^*}{\sum_{i=1}^{m^*} \frac{W_i^*}{i} \lceil \frac{i}{n} \rceil}$$


memory bound

special case with DOP either 1 or n :

$$S_n^* = \frac{W_1^* + W_n^*}{W_1^* + \frac{W_n^*}{n}} = \frac{W_1 + G(n) * W_n}{W_1 + \frac{G(n) * W_n}{n}}$$

where $G(n)$ reflects the increase in workload as memory increases n times

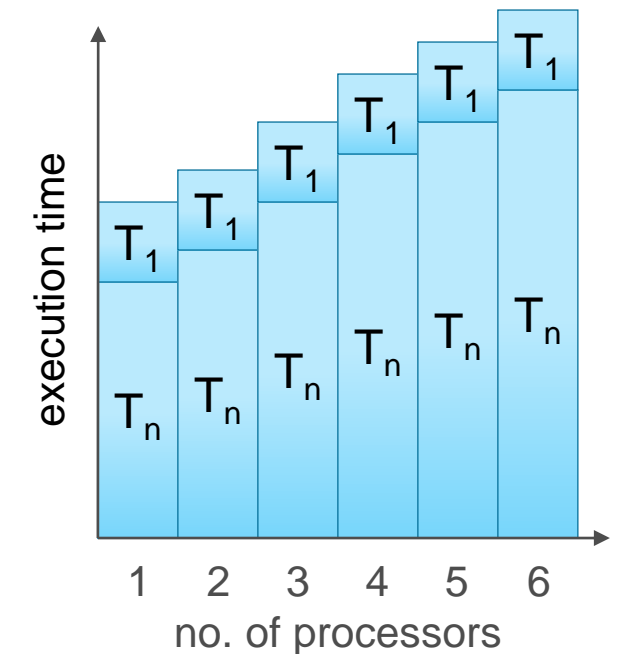
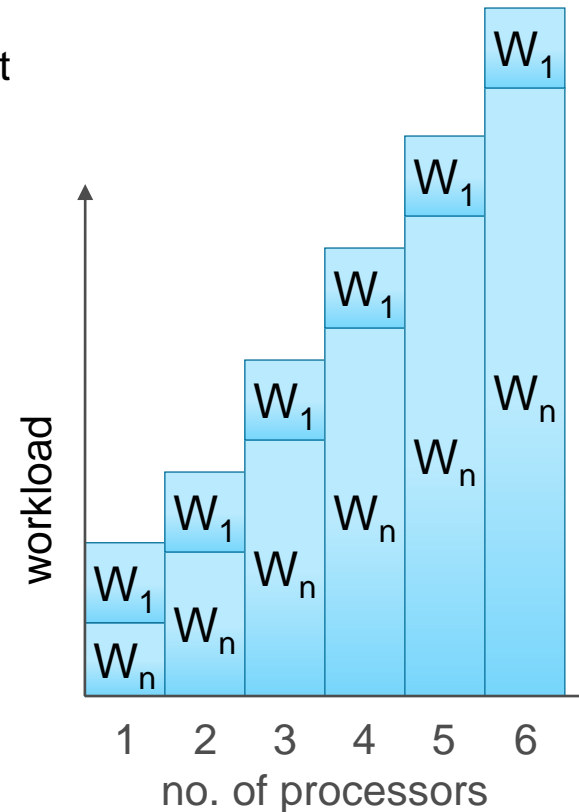
- Case 1: $G(n) = 1$ corresponds to fixed load model
- Case 2: $G(n) = n$ corresponds to fixed time model
- Case 3: $G(n) > n$ workload grows faster than memory, will give a higher speedup (and better resource utilization) than fixed time model

Fixed Memory Speedup

fixed memory model assumes a scaled workload, but allows a slight increase in execution time

increase in workload (problem size) is memory bound

fixed time model can be moved very close to fixed memory model if available memory is fully utilized



PRAM Algorithms



Algorithms and Data Structures 2, 340300
Lecture – 2023W
Univ.-Prof. Dr. Alois Ferscha, teaching@pervasive.jku.at