

Parallel Algorithms

- Examples
- Concepts & Definitions
- Analysis of Algorithms

Lemma

- Any complete binary tree with n leaves has
 - internal nodes = $n-1$ (i.e., $2n-1$ total nodes)
 - height = $\log_2 n$
- **Exercise: Prove it.**

Warming up

- Consider the BTIN (Binary Tree Interconnected Network) computational model. Suppose the tree has n leaves (and hence $2n-1$ processors).
- If we have n numbers stored at the leaves, how can we obtain the sum?
- How can we obtain the max or min?
- How can we propagate a number stored at the root to all leaves?

Warming up

- Suppose we have $n-1$ numbers stored at $n-1$ arbitrary leaves. How can we move these numbers to the $n-1$ internal nodes?
- If the leftmost $n/2$ leaves have numbers, how can we move them to the rightmost leaves?
- How many steps does each of the above computations require?

Example 1.4

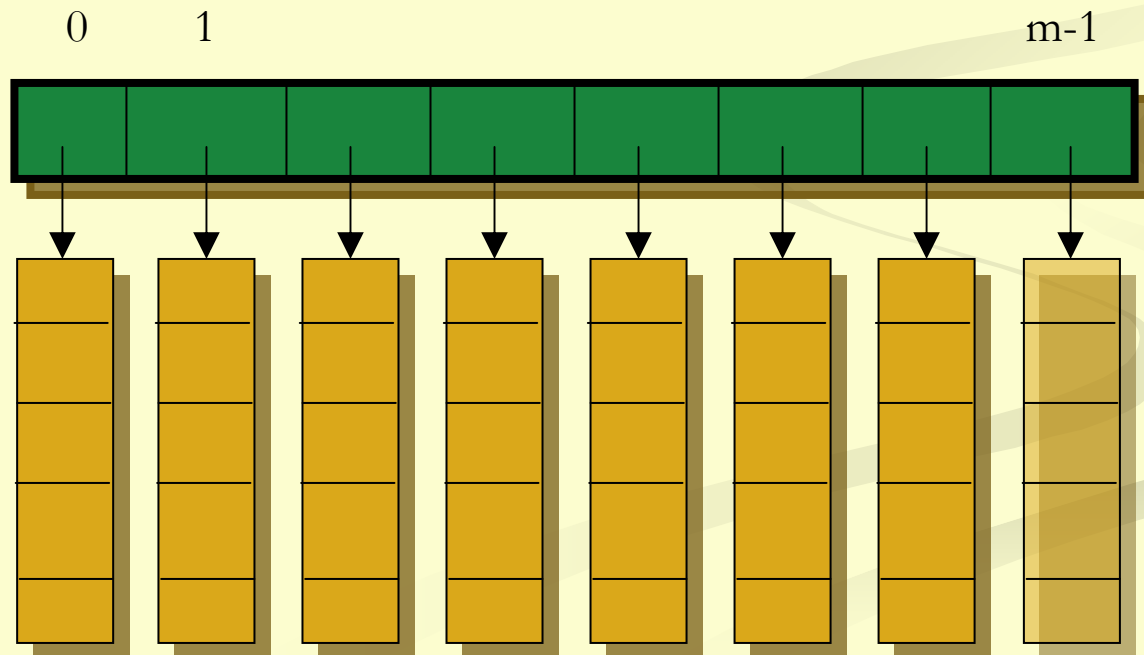
Grouping in a shared-Memory PC

- Given a sequence of pairs $\{(x_1, d_1), \dots, (x_n, d_n)\}$ where $x_i \in \{0, 1, \dots, m-1\}$, $m < n$, and d_i is an arbitrary datum.
- By pigeonhole principle several x_i will be repeated because $m < n$. Write a parallel algorithm to group these pairs according to the x_i 's.

Example 1.4

Grouping in a shared-Memory Model

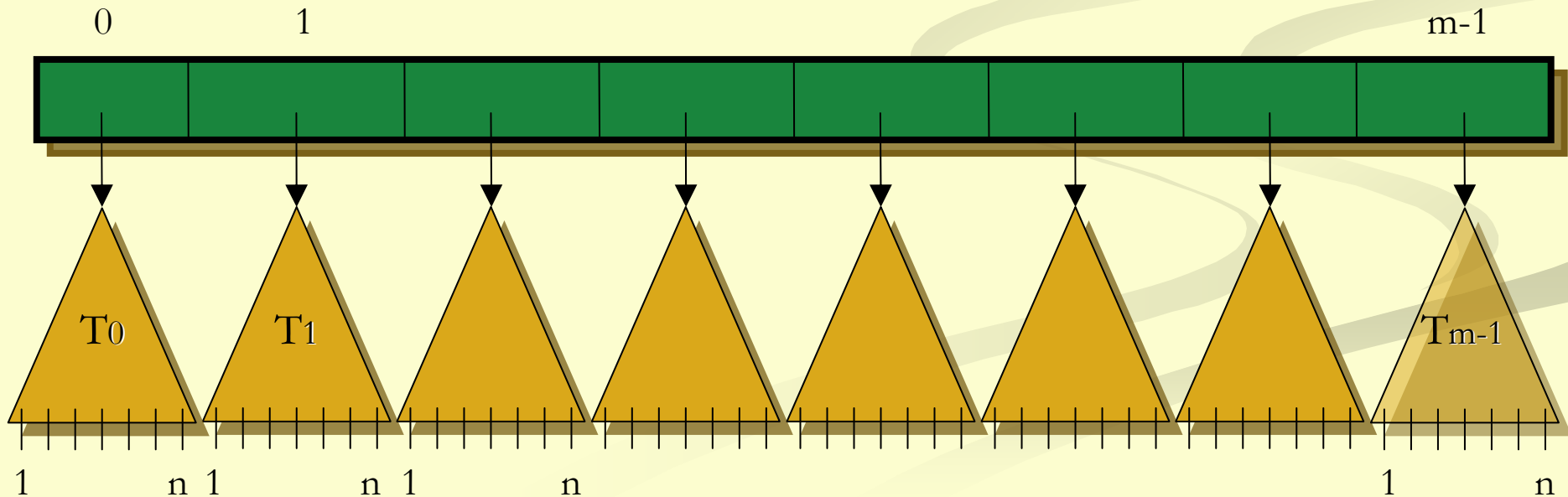
- Sequential algorithm: for each step i , read x and insert it in the hash table.
- Time = n steps Memory = $\Theta(n)$



Example 1.4: Parallel algorithm

Grouping in a shared-Memory Model

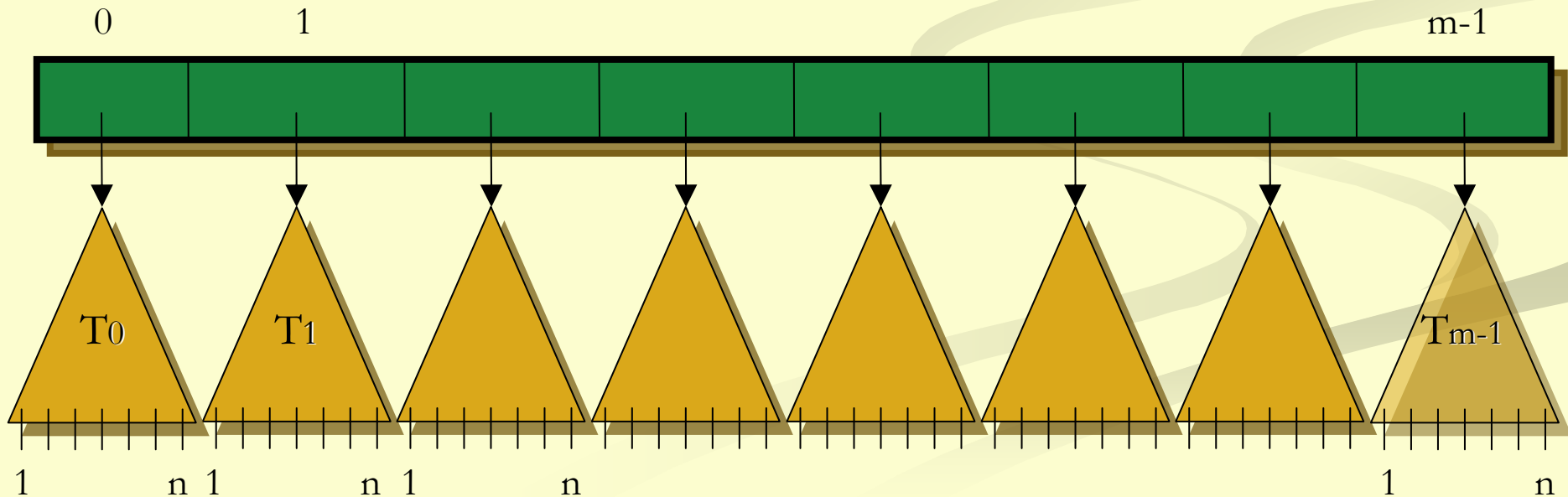
- Shared memory with n processors P_1, P_2, \dots, P_n
- Memory = $m(2n-1)$
- Think about m complete **BT** T_0, T_1, \dots, T_{m-1} each with n leaves numbered $1, 2, \dots, n$ which corresponds to P_1, \dots, P_n .



Example 1.4: Parallel algorithm

Grouping in a shared-Memory Model

- **Phase 1:** Each processor P_i will read the pair (x_i, d_i) and insert it in the leaf i that belongs to the tree T_{x_i}
- **Phase 2:** Each processor P_i will try to move the pair (x_i, d_i) higher up in its tree until it can go no higher as follows:



Example 1.4: Parallel algorithm

Grouping in a shared-Memory Model

Shifting-up rule:

- If node u is free, then the pair in the right child (if any) takes precedence in moving to u over the pair in the left child (if any).

Example 1.4: Parallel algorithm

Analysis

- Since in shared memory parallel computer we have common program for all processors and they execute synchronously, then
- **Phase 1:** takes 1 step only
- **Phase 2:** it takes $\log_2 n$ steps only because each tree has height = $\log_2 n$.
- **Total** = $\log_2 n + 1$ steps
- Extra empty cells in the $m(2n-1)$ memory can be released.

Example: 1.5

Pipelining database in a BTIN model

- In a BTIN with n leaves (processors) containing n distinct records each of the form (k, d) where k is a key and d is a datum.
- Suppose that the root receives a query to retrieve the record whose key is K (if it exists)
- Write a parallel algorithm.

Example: 1.5

Pipelining database in a BTIN model

- **Sequential Algorithm:** Use **binary search algorithm** after sorting the records according to the keys.
- Time = $\Theta(n \log n)$

Example: 1.5: Parallel Algorithm

Pipelining database in a BTIN model

- The root sends the key K to its children which they send subsequently to their children and so on.
- Until it reaches the leaves where it is compared with the keys they stored there.
- The leaf that contains the key is going to send up the corresponding record to the root through its parent and grandparents. Other leaves will send null message.
- When a parent receives a record from one of its children then it will send the same record to its parent; otherwise it will send null.
- and so on ...

Example: 1.5

Analysis

- This is called pipeline technique.
- All processor have the same program working asynchronously when they receive messages from parents or children.
- Time = $2 \log_2 n$ steps to send the key down the tree and receive back the record.

Example: 1.5: Parallel Algorithm

Pipelining database in a BTIN model

- What if we make m queries K_1, K_2, \dots, K_m ?
- **Solution:**
- Sends them sequentially one after another
- **Total time** = $2 \log_2 n + m - 1$

Example 1.6

Prefix (Partial) Sums

- Given n numbers x_0, x_1, \dots, x_{n-1} where n is a power of 2.
- Compute the partial sums for all $k = 0, 1, \dots, n-1$

$$S_k = x_0 + x_1 + \dots + x_k$$

Example 1.6

Prefix (Partial) Sums

- Sequential Algorithm:

We need to make the unavoidable $n-1$ additions.

Example 1.6: Parallel Algorithm

Prefix (Partial) Sums

- For $i = 0, 1, \dots, n-1$, let initially $S_i = x_i$
- Then for $j = 0, \dots, \log_2 n - 1$, let
$$S_i \leftarrow S_i + S_{i-2^j} \quad \text{until } 2^j = i$$
- This is can be done using the **combinatorial Circuit model** with $n(\log_2 n + 1)$ processors distributed over $\log_2 n + 1$ columns and n rows.
- at each step we add the one that is at distance equal to twice the distance we use in the previous step.

Example 1.6: Parallel Algorithm

Analysis

- The number of processors in the model is $n(\log_2 n + 1)$
- The number of columns is $\log_2 n + 1$
- Each processor does at most one step (addition).
- The processors in any fixed column work in parallel.
- Time = $\log_2 n + 1$ additions.

Summary

- At the cost of **increasing the computation power (the number of processors & memory)** we may be able to **decrease the computation time drastically**.

!!Expensive computations!!

- Is it worth it?


Is it worth it?

- Parallel computation is done mainly to speed up computations, while requiring huge processing power.
- To produce Tera FLOPS, the number of CPUs in a massively parallel computer can reach 100s of 1000s.
- This is OK if parallel computing is cheap enough comparing to the critical time reduction of the problem we are solving.
- In the close future, TFLOPS will be available by a single Intel Chip!!

Is it worth it?

- This is indeed the case with many applications in medicine, business, and science that
 - Process huge database,
 - Deal with live streams from huge number of sources
 - require huge number of iterations

Analysis of Parallel Algorithms

- Complexity of algorithms is measured by
 - Time
 - Parallel steps
 - Number of CPUs
- 
- The background of the slide features several light-colored, wavy, horizontal lines that sweep across the bottom right portion of the frame, creating a sense of motion and depth.

Elementary Steps

1. **Computational Steps:** basic arithmetic or logical operations performed within a processor, e.g., comparisons, additions, swapping, ..etc
 - Each takes a constant number of time units

Elementary Steps

2. **Routing Steps:** steps used by the algorithm to move data from one processors to another via shared-memory or interconnections.
 - It is different in shared-memory models than in the interconnection models.


Routing in Shared-memory

- **In shared-memory models:** the interchange is done by accessing the common memory
- It is assumed that it can be done in
 - constant time in the uniform model
(unrealistic but easy to assume)
 - $O(\log M)$ in the non-uniform model with memory of size M .

Routing in Interconnections

- **In interconnection models:** the routing step is measured by the number of links the message has to follow in order to reach the destination processor.
- That is, it is the distance of the shortest path between the source and the destination processors.

Performance Measures

1. Running time
 2. Speedup
 3. Work
 4. Cost
 1. Cost optimality
 2. Efficiency
 5. Success ratio
- 

1. Running Time

- It is measured by the **number of elementary steps** (computational and routings) the algorithm does **from the time the first processor starts to work to the finishing time** of last processor.
- Example:
 - **P1 performs 13 steps, then idle for 3 steps, then 4 steps more.**
 - **P2 performs 11 steps continuously**
 - **P3 performs 5 steps, then 11 steps more.**
 - **Total = 20 steps**

1. Running Time

- The running time depends on the size of input and the number of processors which may depend on input.
- Some time we write t_p to denote the running time of a parallel algorithm that runs on a computer with p processors.

2. Speedup

- The speedup is defined by

$$S(1,p) = t_1 / t_p$$

$$= \frac{\text{the best time known for seq. alg.}}{\text{the time for par. alg. with } p \text{ CPUs}}$$

Speedup Folklore Theorem

- **Theorem:** The speedup $S(1,p) \leq p$

Proof:

- The parallel algorithm can be simulated by a sequential one in $t_p \times p$ time (in the trivial way)
- Since t_1 is an optimal time then $t_1 \leq t_p \times p$
- That is $t_1 / t_p \leq p$ Right ?
- **Well not really!!!** True only for algorithms that can be simulated by sequential computers in $t_p \times p$ time

Speedup Folklore Theorem

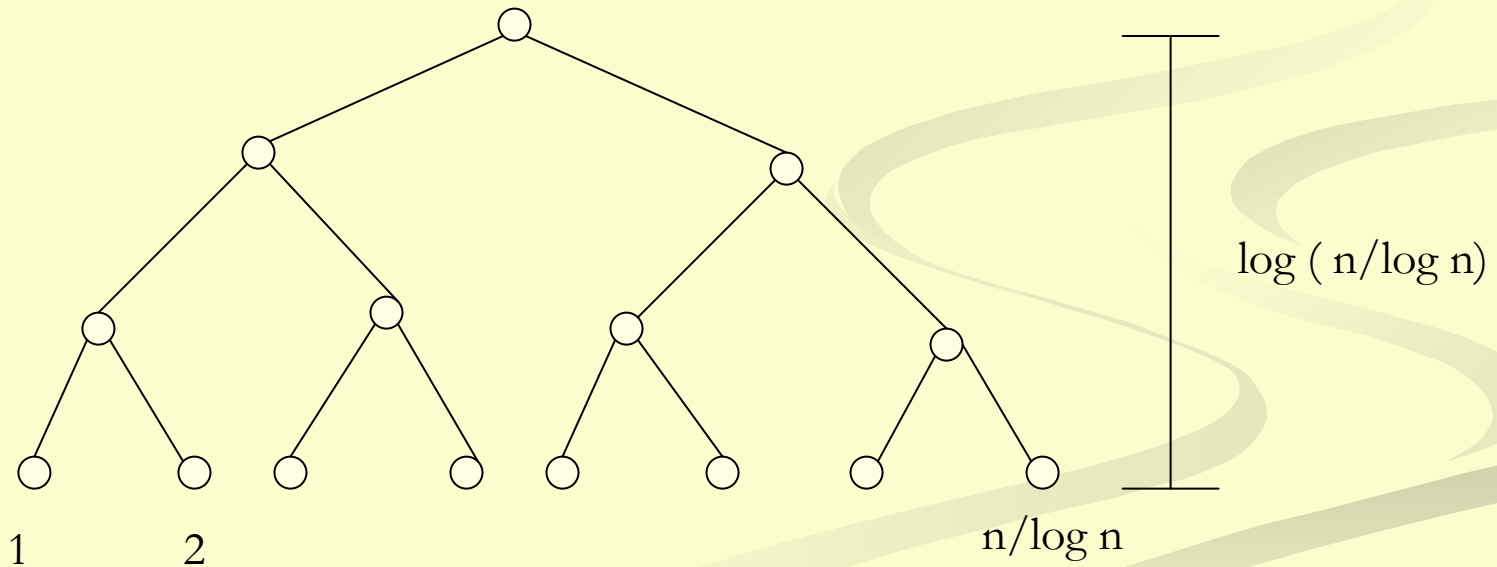
- **Theorem:** The speedup $S(1,p) \leq p$
- **Conclusions:**
- **First,** $t_p \geq t_1/p$
- This means that the running time of any parallel algorithm t_p cannot be better than t_1/p
- **Second, a good parallel algorithm** is the one whose speedup **is very close** to the number of processors.

Example 1.14: Adding n numbers

- Adding n numbers can be done in sequential computer by using $n-1$ additions.
- In parallel computer, it can be done in $O(\log n)$ steps by using BTIN model.
- The tree has $n/\log n$ leaves.
- Each leaf processor adds $\log n$ numbers
- Each parent adds the number of its children and so on.
- The root will contain the result.

Analysis

- Time to add numbers in each processor = $O(\log n)$ steps
- Time to propagate = $O(\log n)$ steps
- **Speedup = $O(n/\log n)$**



Example 1.15:

Searching in Shared Memory Model

- Given a number x and an array $A[1..n]$ containing n distinct numbers sorted increasingly, all stored in a memory.
- Write a parallel algorithm that searches for x in the list and returns its position.

Example 1.15:

Searching in Shared Memory Model

- **Sequential algorithm:** binary search algorithm can solve the problem in $O(\log n)$ time.
- However, one can achieve $O(1)$ parallel time!
- In shared memory model with n processors let each processor P_i compares x with the cell $A[i]$
- Specify a location in the memory call it **answer**; initially **answer=0**.
- Any processor that finds x will write its index in the location **answer**.
- I.e. if **answer=i**, then P_i finds x in $A[i]$

Example 1.15:

Analysis

- Surely the overall time is $O(1)$ steps.
- $\text{Speedup} = O(\log n) \leq n = \# \text{ of processors used}$
- Is it possible to use only $O(\log n)$ processors to achieve the same performance?

How about searching in the BTIN model?

- Use the BTIN model with n leaves (and $2n-1$ total processors)
- The leaf processor P_i holds $A[i]$
- The root will take the input x and propagates it to the leaves and the answer will return back to the root.
- Any parallel algorithm has to take at least $\Omega(\log n)$ to just traverse the links between the processors.
- **Speedup = $O(1)$** in the best which **way smaller** than $2n-1$ the number of processors.

Having said that ...

- It appears that the speedup Theorem **is not always true** specially if the parallel computer has many different stream inputs which can't be simulated properly in a sequential computer.
- **Counter Example: read 1.17.**

Slowdown Folklore Theorem

- **Theorem:** if a certain problem can be solved with p processors in t_p time and with q processors in t_q time where $q < p$, then

$$t_p \leq t_q \leq t_p + p t_p / q$$

- That is when the number of CPUs decreases from p to q then the running time can slowdown by a factor of $(1+p/q)$ in the worst case.
- Or when the number of CPUs increases from q to p then the running time can be reduced by a factor of $1/(1+p/q)$ in the best case.

Idea of Proof

- Suppose that you have a parallel algorithm that runs on a computer with p processors.
- To run the same algorithm on a computer with q processors you need to **distribute** the tasks that the **p processors** do (**at most $p t_p$ steps**) into the **q processors** as evenly as possible.
- Thus, each processor will have to do at most $p t_p / q$ steps And so **$t_q \leq t_p + p t_p / q$.**
- For detail See Page 18

Slowdown Folklore Theorem (Brent's Theorem)

- Notice that if $q=1$, then

$$1 \leq t_1 / t_p = S(1,p) \leq 1 + p$$

- The Slowdown theorem is not always true specially when the distribution of input data or the communications impose overhead on the running time.
- See Example 1.19

3. Number of Processors

... Why?

- Algorithms (with same running time and on same models) but with less number of processors are preferred (less expensive).
- Sometimes optimal times and speedups can be achieved with certain number of processors
- A minimum number of processors may be required to have successful computations
- Slowdown and speedup theorems show that number of processors is important.
- Certain computational models may not accommodate the required number of processors. (e.g., perfect squares or prime number)
- In combinatorial circuits each CPU is used at most once. That gives an upper bound on the time.

4. The Work

- **The Work** is defined to be the **exact total** number of elementary steps executed by all processors.
- **Running time = maximum** of elementary steps used by any processor
- **Exercise:** Think about the combinatorial circuit model.

5. The Cost

- The cost $C(n)$ is an upper bound on the total number of elementary steps used by all processors, and defined as

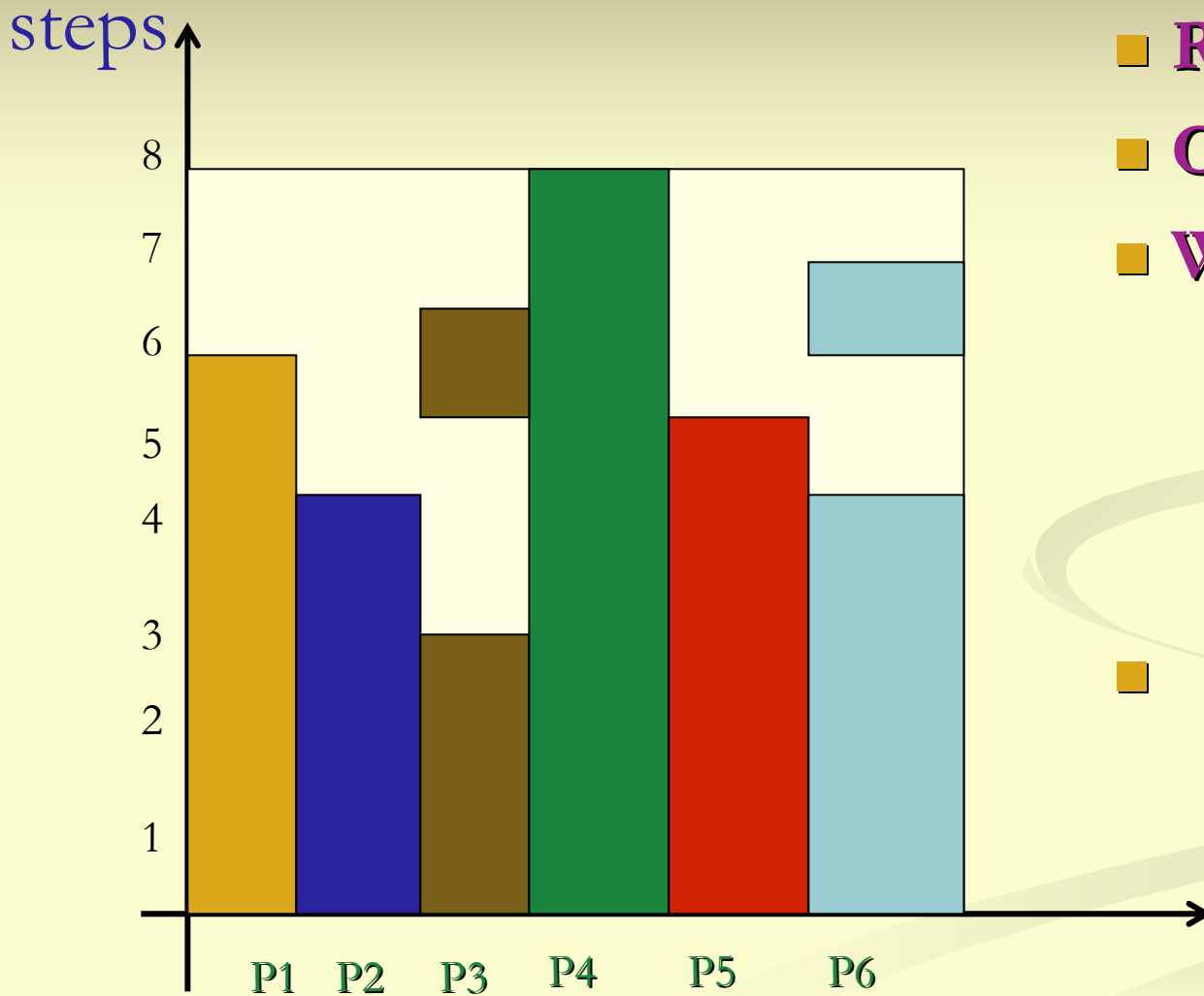
$$C(n) = t(n) \times p(n), \text{ where}$$

$t(n)$ = the running time

$p(n)$ = number of processors

- **Note:** not all processors are necessarily active during the $t(n)$ time units.
- In combinatorial circuit model: $C(n) = p(n)$ by definition.

Example



■ Running time = 8

■ Cost = $8 \times 6 = 48$

■ Work = $6+4+4$
 $+8+5+5$
 $= 32$

■ $RT \leq Work \leq Cost$

5.1 Cost Optimality

- Notice that the cost is indeed the worst case running time needed to simulate a parallel algorithm on a sequential computer (if it can be done).
- For the following: we restricted ourself to “simulate-able” parallel algorithms only.
- 1. If $\Omega(f(n))$ number of steps are needed to solve a problem sequentially, and the cost of a parallel algorithm is $O(f(n))$, then we say that the algorithm is asymptotically cost optimal.

5.1 Cost Optimality

- Recall Example 1.14 of Adding n numbers via BTIN. We used $p = O(n/\log n)$ processors to achieve $O(\log n)$ running time.
- So the cost = $O(n)$.
- But adding any n numbers need $\Omega(n)$ sequential steps. Thus the cost is optimal.
- **Notice:** if we use BTIN with n leaves, the cost is $O(n \log n)$ which is not optimal.

This means that ..

- If $\Omega(f(n))$ is a lower bound on the required number of steps to solve a problem of size n , then $\Omega(f(n)/p)$ is a lower bound on the running time of parallel algorithm with p processors.

This follows from the speedup theorem which says that the reduction in the running time is by at most a factor of $1/p$.

- Example: Any parallel algorithm that uses n processors needs $\Omega(\log n)$ steps to sort n numbers, because sequential sorting needs $\Omega(n \log n)$ steps.

5.1 Cost Optimality

2. The cost of a parallel algorithm is not optimal if an equivalent sequential algorithm exists whose worst case running time is better than the cost.
- Recall Example 1.4 of grouping n pairs into m groups. We used n processors in a shared memory model to solve the problem in $O(\log n)$ time.
 - The cost is $O(n \log n)$ steps which is **not optimal** because the sequential algorithm uses $O(n)$ steps.

5.1 Cost Optimality

3. Unknown cost optimality is possible when we have parallel algorithm whose cost matches the best known sequential running time but we don't know if the sequential running time is optimal.
- **Example:** Matrix multiplication requires $\Omega(n^2)$ steps. The best known sequential algorithm takes $\Omega(n^c)$ where $2 < c < 2.38$. We don't know if it is optimal though.

5.2 Efficiency

- The efficiency of a parallel algorithm is defined by

$$E(1,p) = t_1 / (p t_p), \text{ where}$$

- t_1 is the running time of the best known sequential algorithm,
- t_p is the running time of the parallel algorithm that runs on a computer with p processors.

5.2 Efficiency

- If the parallel algorithm is within **our restriction**, then $E(1,p) \leq 1$.
- If $E(1,p) < 1$, the parallel algorithm is not cost optimal ... NOT GOOD
- If $E(1,p) = 1$ and t_1 is optimal, then the parallel algorithm is cost optimal ... GOOD
- If $E(1,p) > 1$, then the simulated sequential algorithm (if doable!) is faster than the parallel algorithm. ... IDEAL
- Read Example 1.26

Summary

- It is unfair to compare the running time of a parallel algorithm t_p to the running time of the best known sequential algorithm t_1 .
- We should compare the cost= $p t_p$ to t_1 .

6. Success Ratio

- Consider algorithms that solve problems correctly with certain probabilities.
- Let $\text{Pr}(p)$ = the probability of success that a parallel algorithm solves correctly a given problem.
- Let $\text{Pr}(1)$ = the probability of success that a sequential algorithm solves correctly the same given problem.
- The **success ratio** is defined by

$$\text{Sr}(1,p) = \text{Pr}(p) / \text{Pr}(1)$$

6. Success Ratio

- The **success ratio** is defined by

$$sr(1,p) = \Pr(p) / \Pr(1)$$

- The **scaled success ratio** is

$$ssr(1,p) = \Pr(p) / (p \times \Pr(1))$$

- **Usually:** $sr(1,p) \leq p$ and $ssr(1,p) \leq 1$