Chapter 10
Parallel Algorithms

Algorithm Theory
WS 2017/18

Fabian Kuhn

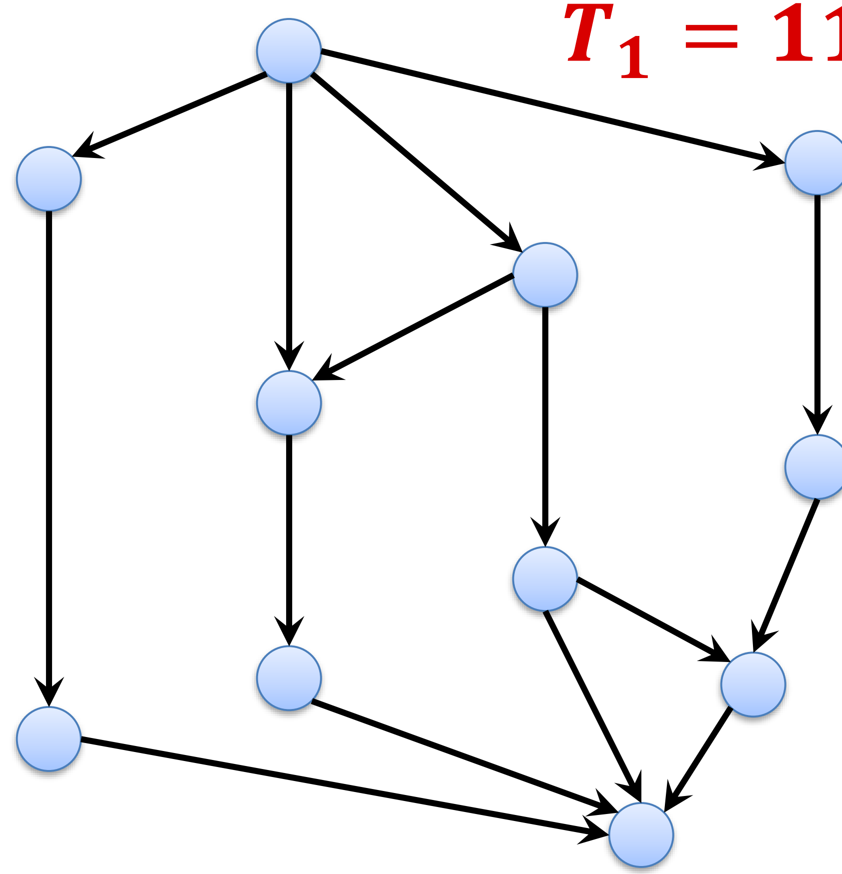
Parallel Computations

$\boldsymbol{T_p}$: time to perform comp. with p procs

- T_1: **work** (total # operations)
  - Time when doing the computation sequentially
- T_∞: **critical path / span**
  - Time when parallelizing as much as possible
- **Lower Bounds**:

$$T_p \geq \frac{T_1}{p}, \qquad T_p \geq T_\infty$$

$$T_1 = 11$$



$$T_\infty = 5$$

Parallel Computations

$\boldsymbol{T_p}$: time to perform comp. with p procs

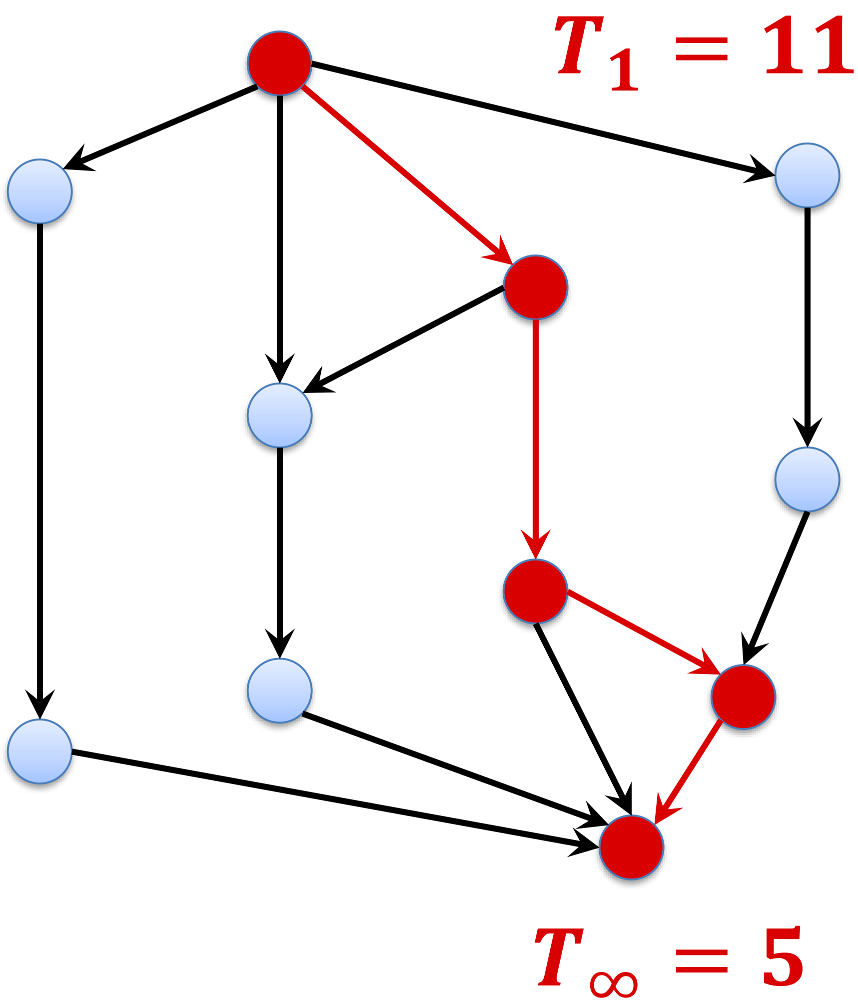
- **Lower Bounds**:

$$T_p \geq \frac{T_1}{p}, \qquad T_p \geq T_\infty$$

- **Parallelism**: $\frac{T_1}{T_\infty}$
  - maximum possible speed-up

- **Linear Speed-up**:

$$\frac{T_p}{T_1} = \Theta(p)$$



Brent's Theorem

Brent's Theorem: On p processors, a parallel computation can be performed in time

$$T_p \leq \frac{T_1 - T_\infty}{p} + T_\infty.$$

Corollary: Greedy is a 2-approximation algorithm for scheduling.

Corollary: As long as the number of processors $p = O(T_1/T_\infty)$, it is possible to achieve a linear speed-up.

PRAM

Back to the PRAM:

- Shared random access memory, synchronous computation steps
- The PRAM model comes in variants…

EREW (exclusive read, exclusive write):

- Concurrent memory access by multiple processors is not allowed
- If two or more processors try to read from or write to the same memory cell concurrently, the behavior is not specified

CREW (concurrent read, exclusive write):

- Reading the same memory cell concurrently is OK
- Two concurrent writes to the same cell lead to unspecified behavior
- This is the first variant that was considered (already in the 70s)

PRAM

The PRAM model comes in variants…

CRCW (concurrent read, concurrent write):

- Concurrent reads and writes are both OK
- Behavior of concurrent writes has to specified
  - Weak CRCW: concurrent write only OK if all processors write 0
  - Common-mode CRCW: all processors need to write the same value
  - Arbitrary-winner CRCW: adversary picks one of the values
  - Priority CRCW: value of processor with highest ID is written
  - Strong CRCW: largest (or smallest) value is written
- The given models are ordered in strength:

  weak $\leq$ common-mode $\leq$ arbitrary-winner $\leq$ priority $\leq$ strong

Prefix Sums

- The following works for any associative binary operator $\oplus$:

  associativity: $(a \oplus b) \oplus c = a \oplus (b \oplus c)$

All-Prefix-Sums: Given a sequence of n values $a_1, \ldots, a_n$, the all-prefix-sums operation w.r.t. $\oplus$ returns the sequence of prefix sums:

$$s_1, s_2, \ldots, s_n = a_1, a_1 \oplus a_2, a_1 \oplus a_2 \oplus a_3, \ldots, a_1 \oplus \cdots \oplus a_n$$

- Can be computed efficiently in parallel and turns out to be an important building block for designing parallel algorithms

Example: Operator: $+$, input: $a_1, \ldots, a_8 = 3, 1, 7, 0, 4, 1, 6, 3$

$s_1, \ldots, s_8 =$

Computing the Sum

- Let's first look at $s_n = a_1 \oplus a_2 \oplus \cdots \oplus a_n$
- Parallelize using a binary tree:

Computing the Sum

Lemma: The sum $s_n = a_1 \oplus a_2 \oplus \cdots \oplus a_n$ can be computed in time $O(\log n)$ on an EREW PRAM. The total number of operations (total work) is $O(n)$.

Proof:

Corollary: The sum s_n can be computed in time $O(\log n)$ using $O(n/\log n)$ processors on an EREW PRAM.

Proof:

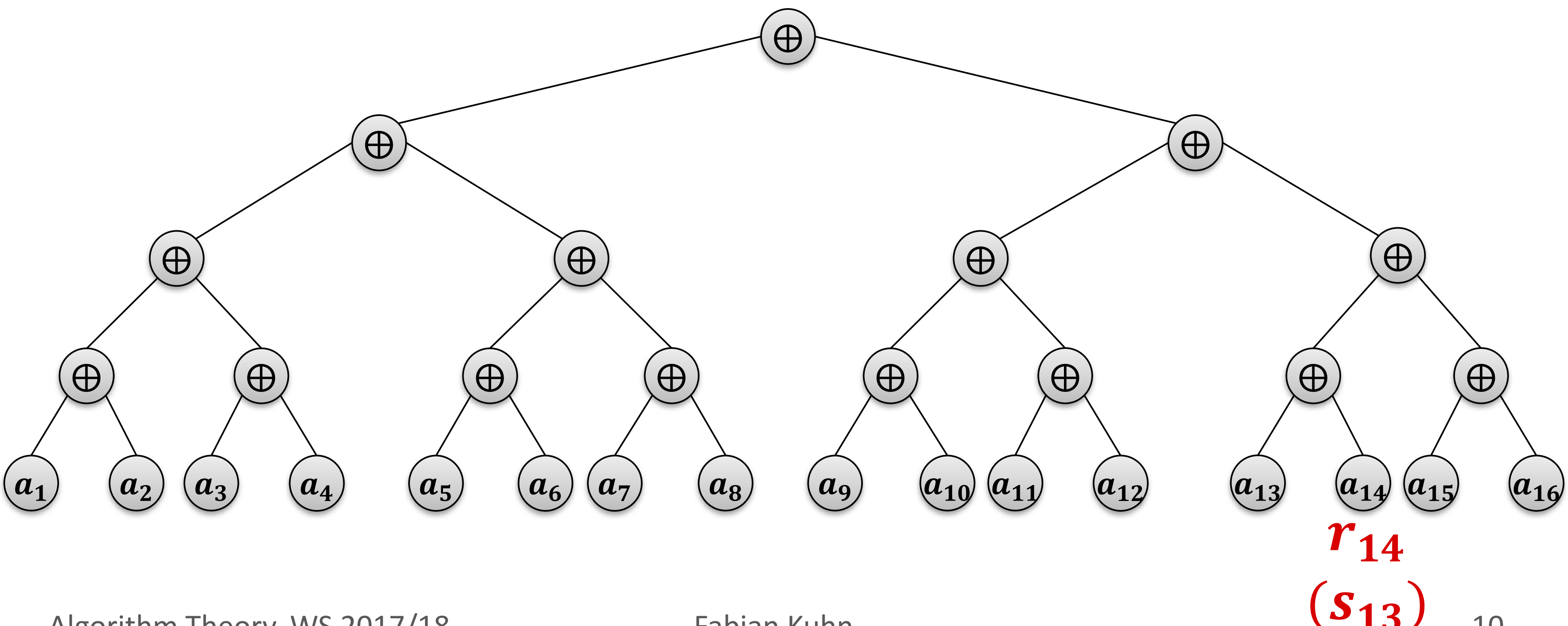
- Follows from Brent's theorem ($T_1 = O(n)$, $T_\infty = O(\log n)$)

Getting The Prefix Sums

- Instead of computing the sequence $s_1, s_2, \dots, s_n$ let's compute $r_1, \dots, r_n = 0, s_1, s_2, \dots, s_{n-1}$ (0: neutral element w.r.t. $\oplus$)

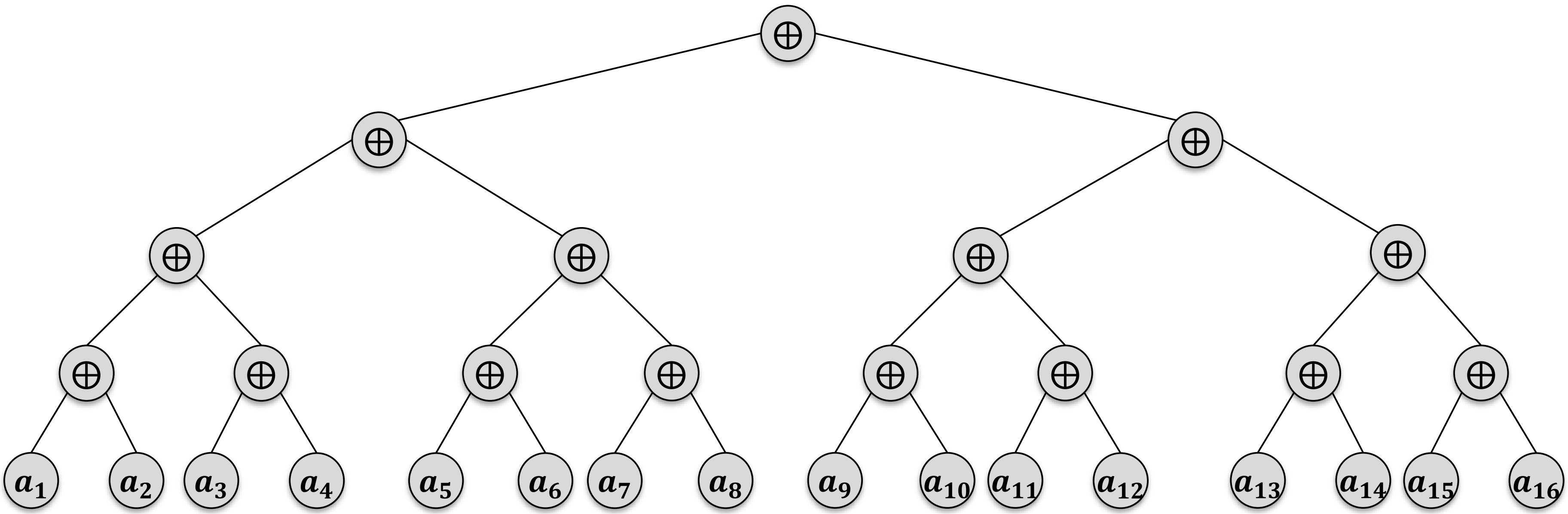
$$r_1, \dots, r_n = 0, a_1, a_1 \oplus a_2, \dots, a_1 \oplus \cdots \oplus a_{n-1}$$

- Together with s_n, this gives all prefix sums
- Prefix sum $r_i = s_{i-1} = a_1 \oplus \cdots \oplus a_{i-1}$:



Getting The Prefix Sums

Claim: The prefix sum $r_i = a_1 \oplus \cdots \oplus a_{i-1}$ is the sum of all the leaves in the left sub-tree of ancestor u of the leaf v containing a_i such that v is in the right sub-tree of u.



Computing The Prefix Sums

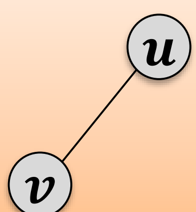
For each node v of the binary tree, **define $r(v)$ as follows:**

- $r(v)$ is the sum of the values a_i at the leaves in all the left sub-trees of ancestors u of v such that v is in the right sub-tree of u.

For a leaf node v holding value a_i: $\boldsymbol{r(v) = r_i = s_{i-1}}$

For the root node: $\boldsymbol{r(\mathrm{root}) = 0}$

For all other nodes v:

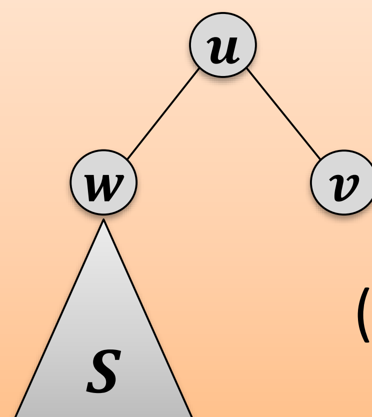


v is the left child of u:

$$r(v) = r(u)$$

v is the right child of u:

(u has left child w)



$$r(v) = r(u) + S$$

(S: sum of values in sub-tree of w)

Computing The Prefix Sums

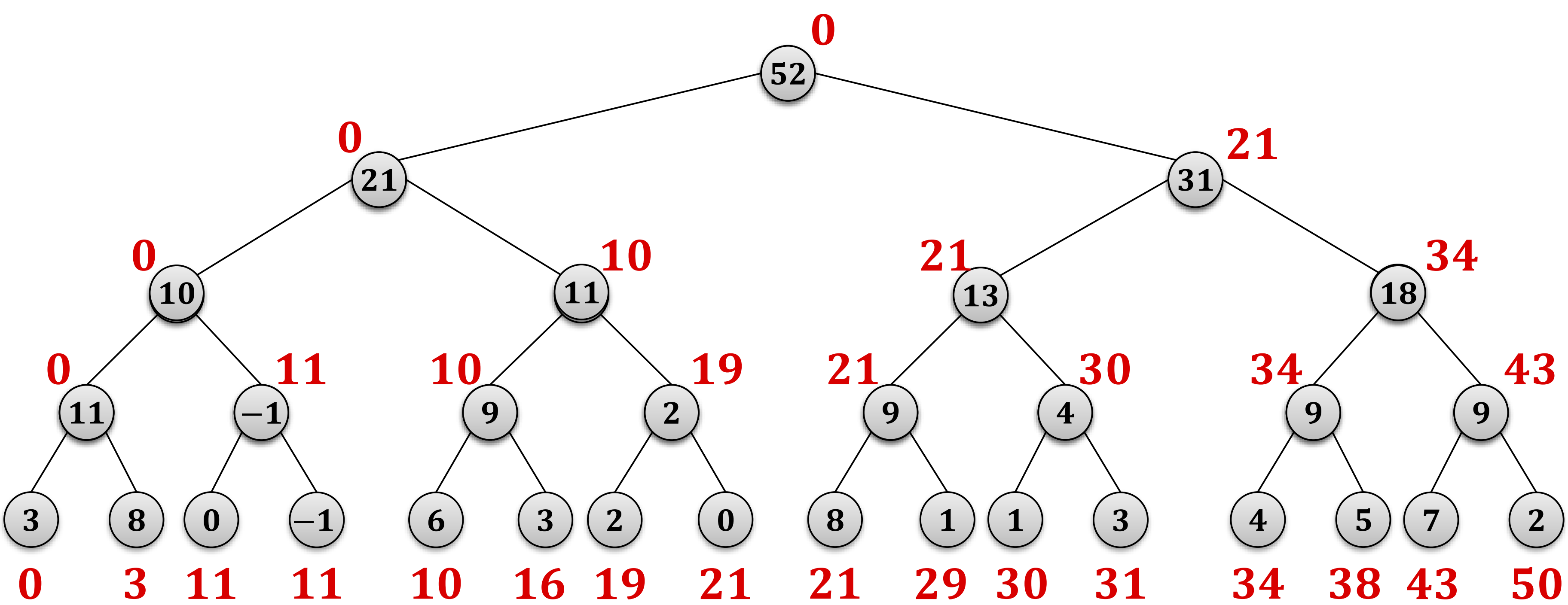
- leaf node v holding value a_i: $\boldsymbol{r(v) = r_i = s_{i-1}}$
- root node: $\boldsymbol{r(\mathbf{root}) = 0}$
- Node v is the left child of u: $r(v) = r(u)$
- Node v is the right child of u: $r(v) = r(u) + S$
  - Where: $S =$ sum of values in left sub-tree of u

Algorithm to compute values $\boldsymbol{r(v)}$:

1. Compute sum of values in each sub-tree (bottom-up)
   - Can be done in parallel time $O(\log n)$ with $O(n)$ total work
2. Compute values $r(v)$ top-down from root to leaves:
   - To compute the value $r(v)$, only $r(u)$ of the parent u and the sum of the left sibling (if v is a right child) are needed
   - Can be done in parallel time $O(\log n)$ with $O(n)$ total work

Example

1. Compute sums of all sub-trees
   – Bottom-up (level-wise in parallel, starting at the leaves)
2. Compute values $r(v)$
   – Top-down (starting at the root)



Computing Prefix Sums

Theorem: Given a sequence $a_1, \dots, a_n$ of n values, all prefix sums $s_i = a_1 \oplus \cdots \oplus a_i$ (for $1 \leq i \leq n$) can be computed in time $O(\log n)$ using $O(n/\log n)$ processors on an EREW PRAM.

Proof:

- Computing the sums of all sub-trees can be done in parallel in time $O(\log n)$ using $O(n)$ total operations.
- The same is true for the top-down step to compute the $r(v)$
- The theorem then follows from Brent's theorem:

$$T_1 = O(n), \qquad T_\infty = O(\log n) \qquad \Longrightarrow \qquad T_p < T_\infty + \frac{T_1}{p}$$

Remark: This can be adapted to other parallel models and to different ways of storing the value (e.g., array or list)

Parallel Quicksort

- Key challenge: parallelize partition

pivot

5	14	18	8	19	21	3	1	25	17	11	4	20	10	26	2	9	13	23	16

partition

5	14	8	3	1	11	4	10	2	9	13	16	18	19	21	25	17	20	26	23

- How can we do this in parallel?
- For now, let's just care about the values $\leq$ pivot
- What are their new positions

Using Prefix Sums

- Goal: Determine positions of values $\leq$ pivot after partition

pivot

5	14	18	8	19	21	3	1	25	17	11	4	20	10	26	2	9	13	23	16

1	1	0	1	0	0	1	1	0	0	1	1	0	1	0	1	1	1	0	1

prefix sums

1	2	2	3	3	3	4	5	5	5	6	7	7	8	8	9	10	11	11	12

partition

5	14	8	3	1	11	4	10	2	9	13	16	18	19	21	25	17	20	26	23

Applying to Quicksort

Theorem: On an EREW PRAM, using p processors, randomized quicksort can be executed in time T_p (in expectation and with high probability), where

$$T_p = O\left(\frac{n \log n}{p} + \log^2 n\right).$$

Proof:

Remark:

- We get optimal (linear) speed-up w.r.t. to the sequential algorithm for all $p = O(n/\log n)$.

Partition Using Prefix Sums

- The positions of the entries $>$ pivot can be determined in the same way
- **Prefix sums:** $T_1 = O(n), \quad T_\infty = O(\log n)$
- **Remaining computations:** $T_1 = O(n), \quad T_\infty = O(1)$
- **Overall:** $T_1 = O(n), \quad T_\infty = O(\log n)$

Lemma: The partitioning of quicksort can be carried out in parallel in time $O(\log n)$ using $O\left(\frac{n}{\log n}\right)$ processors.

Proof:

- By Brent's theorem: $T_p \leq \frac{T_1}{p} + T_\infty$

Other Applications of Prefix Sums

- Prefix sums are a very powerful primitive to design parallel algorithms.
  - Particularly also by using other operators than "+"

Example Applications:

- Lexical comparison of strings
- Add multi-precision numbers
- Evaluate polynomials
- Solve recurrences
- Radix sort / quick sort
- Search for regular expressions
- Implement some tree operations
- …

Prefix Sums in Linked Lists

Given: Linked list L of length n in the following way

- Elements are in an array A of length n in an unordered way
- Each array element $A[i]$ also contains a next pointer
- Pointer $first$ to the first element of the list

Goal: Compute all prefix sums w.r.t. to the order given by the list

2-Ruling Set of a Linked List

Given a linked list, select a subset of the entries such that

- No two neighboring entries are selected
- For every entry that is not selected, either the predecessor or the successor is selected
  - i.e., between two consecutive selected entries there are at least one and at most two unselected entries

- We will see that a 2-ruling set of a linked list can be computed efficiently in parallel

Using 2-Ruling Sets to Get Prefix Sums

Observations:

- To compute the prefix sums of an array/list of numbers, we need a binary tree such that the numbers are at the leaves and an in-order traversal of the tree gives the right order
- The algorithm can be generalized to non-binary trees

Using 2-Ruling Sets to Get Prefix Sums

Basic Idea:

- Use 2-ruling sets to build a tree of logarithmic depth

Using 2-Ruling Sets to Get Prefix Sums

Lemma: If a 2-Ruling Set of a list of length N can be computed in parallel with $w(N)$ work and $d(N)$ depth, all prefix sums of a list of length n can be computed in parallel with

- Work $O\big(w(n) + w(n/2) + w(n/4) + \cdots + w(1)\big)$
- Depth $O\big(d(n) + d(n/2) + d(n/4) + \cdots + d(1)\big)$

Proof Sketch:

Prefix Sums in Linked Lists

Log-Star Function:

- For $i \geq 1$: $\log_2^{(i)} x = \log_2\left(\log_2^{(i-1)} x\right)$, and $\log_2^{(0)} x = x$
- For $x > 2$: $\log^* x \coloneqq \min\{i : \log^{(i)} x \leq 2\}$, for $x \leq 2$: $\log^* x \coloneqq 1$

Lemma: A 2-ruling set of a linked list of length n can be computed in parallel with work $O(n \cdot \log^* n)$ and span $O(\log^* n)$.

- i.e., in time $O(\log^* n)$ using $O(n)$ processors
  – We will first see how to apply this and prove it afterwards…

Prefix Sums in Linked Lists

Lemma: A 2-ruling set of a linked list of length n can be computed in parallel with work $O(n \cdot \log^* n)$ and span $O(\log^* n)$.

Theorem: All prefix sums of a linked list of length n can be computed in parallel with total work $O(n \cdot \log^* n)$ and span $O(\log n \cdot \log^* n)$.

- i.e., in time $O(\log n \cdot \log^* n)$ using $O(n/\log n)$ processors.

Computing 2-Ruling Sets

- Instead of computing a 2-ruling set, we first compute a coloring of the list:
  - each list element gets a color s.t. adjacent elements get different colors
- Each element initially has a unique $\log n$-bit label in $\{1, \dots, N\}$
  - can be interpreted as an initial coloring with N colors

Algorithm runs in phases:

- Each phase: compute new coloring with smaller number of colors

We will show that

- #phases to get to $O(1)$ colors is $O(\log^* n)$
- each phase has $O(n)$ work and $O(1)$ depth

Reducing the number of colors

Assume that we start with a coloring with colors $\{0, \dots, x-1\}$

Reducing the number of colors

Assume that we start with a coloring with colors $\{0, \ldots, x-1\}$

Reducing the number of colors

Assume that we start with a coloring with colors $\{0, \dots, x-1\}$