

# 3

# Efficient Sorting

17 February 2022

Sebastian Wild

# Learning Outcomes

1. Know principles and implementation of *mergesort*.
2. Know principles and implementation of *quicksort*.
3. Know properties and *performance characteristics* of mergesort and quicksort.
4. Know the comparison model and understand the corresponding *lower bound*.
5. Understand *counting sort* and how it circumvents the comparison lower bound.
6. Understand and use the *parallel random-access-machine* model in its different variants.
7. Be able to *analyze* and compare simple shared-memory parallel algorithms by determining *parallel time and work*.
8. Understand efficient parallel *prefix sum* algorithms.
9. Be able to devise high-level description of *parallel quicksort and mergesort* methods.

## Unit 3: Efficient Sorting



# Outline

## 3 Efficient Sorting

3.1 Mergesort

3.2 Quicksort

3.3 Comparison-Based Lower Bound

3.4 Integer Sorting

3.5 Parallel computation

3.6 Parallel primitives

3.7 Parallel sorting

# Why study sorting?

- ▶ fundamental problem of computer science that is still not solved
- ▶ building brick of many more advanced algorithms
  - ▶ for preprocessing
  - ▶ as subroutine
- ▶ playground of manageable complexity to practice algorithmic techniques

Algorithm with optimal #comparisons in worst case?



Here:

- ▶ “classic” fast sorting method
- ▶ **parallel** sorting

# Part I

## *The Basics*

# Rules of the game

- ▶ **Given:**

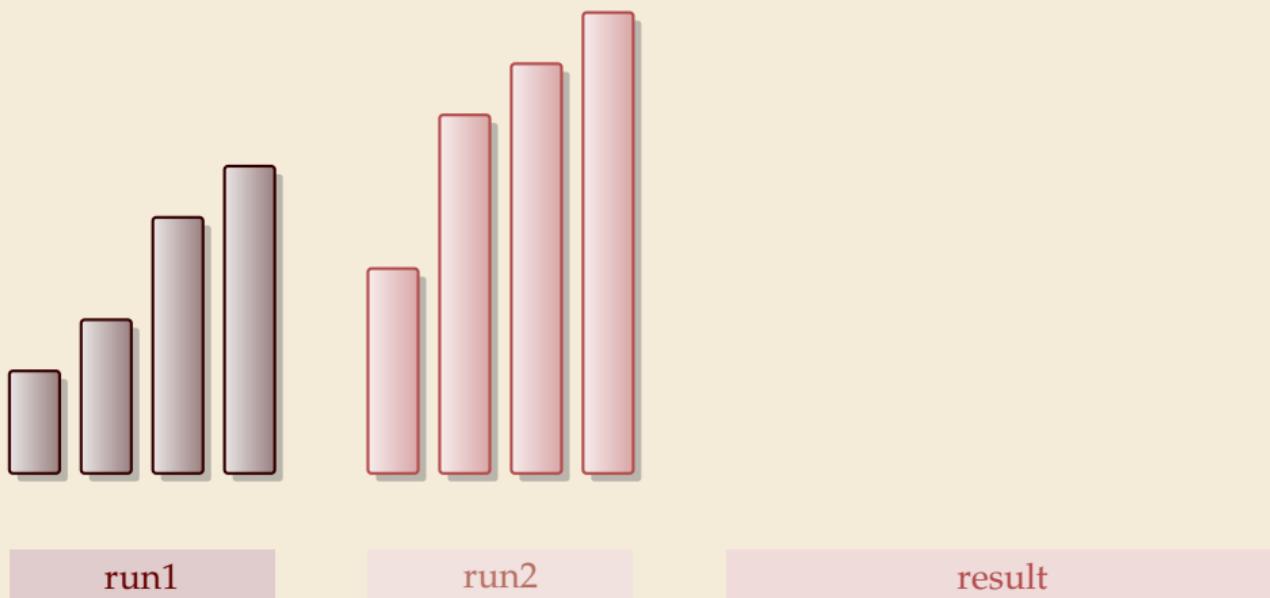
- ▶ array  $A[0..n - 1]$  of  $n$  objects
- ▶ a total order relation  $\leq$  among  $A[0], \dots, A[n - 1]$   
(a comparison function)

- ▶ **Goal:** rearrange (=permute) elements within  $A$ ,  
so that  $A$  is *sorted*, i. e.,  $A[0] \leq A[1] \leq \dots \leq A[n - 1]$

- ▶ for now:  $A$  stored in main memory (*internal sorting*)  
single processor (*sequential sorting*)

## 3.1 Mergesort

# Merging sorted lists

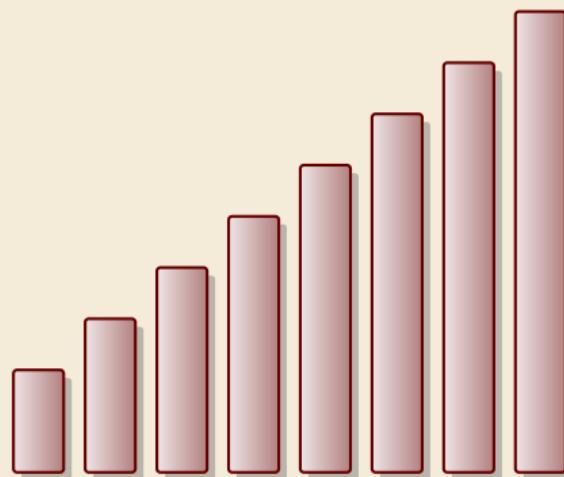


run1

run2

result

# Merging sorted lists



run1

run2

result

# Mergesort

---

```
1 procedure mergesort(A[l..r])
2     n := r - l + 1
3     if n ≤ 1 return
4     m := l + ⌊ n / 2 ⌋
5     mergesort(A[l..m - 1])
6     mergesort(A[m..r])
7     merge(A[l..m - 1], A[m..r], buf)
8     copy buf to A[l..r]
```

---

- ▶ recursive procedure; *divide & conquer*
- ▶ merging needs
  - ▶ temporary storage for result of same size as merged runs
  - ▶ to read and write each element twice (once for merging, once for copying back)

**Analysis:** count “element visits” (read and/or write)

$$C(n) = \begin{cases} 0 & n \leq 1 \\ C(\lfloor n/2 \rfloor) + C(\lceil n/2 \rceil) + 2n & n \geq 2 \end{cases}$$

same for best and worst case!

Simplification  $n = 2^k$

$$C(2^k) = \begin{cases} 0 & k \leq 0 \\ 2 \cdot C(2^{k-1}) + 2 \cdot 2^k & k \geq 1 \end{cases} = 2 \cdot 2^k + 2^2 \cdot 2^{k-1} + 2^3 \cdot 2^{k-2} + \dots + 2^k \cdot 2^1 = 2k \cdot 2^k$$

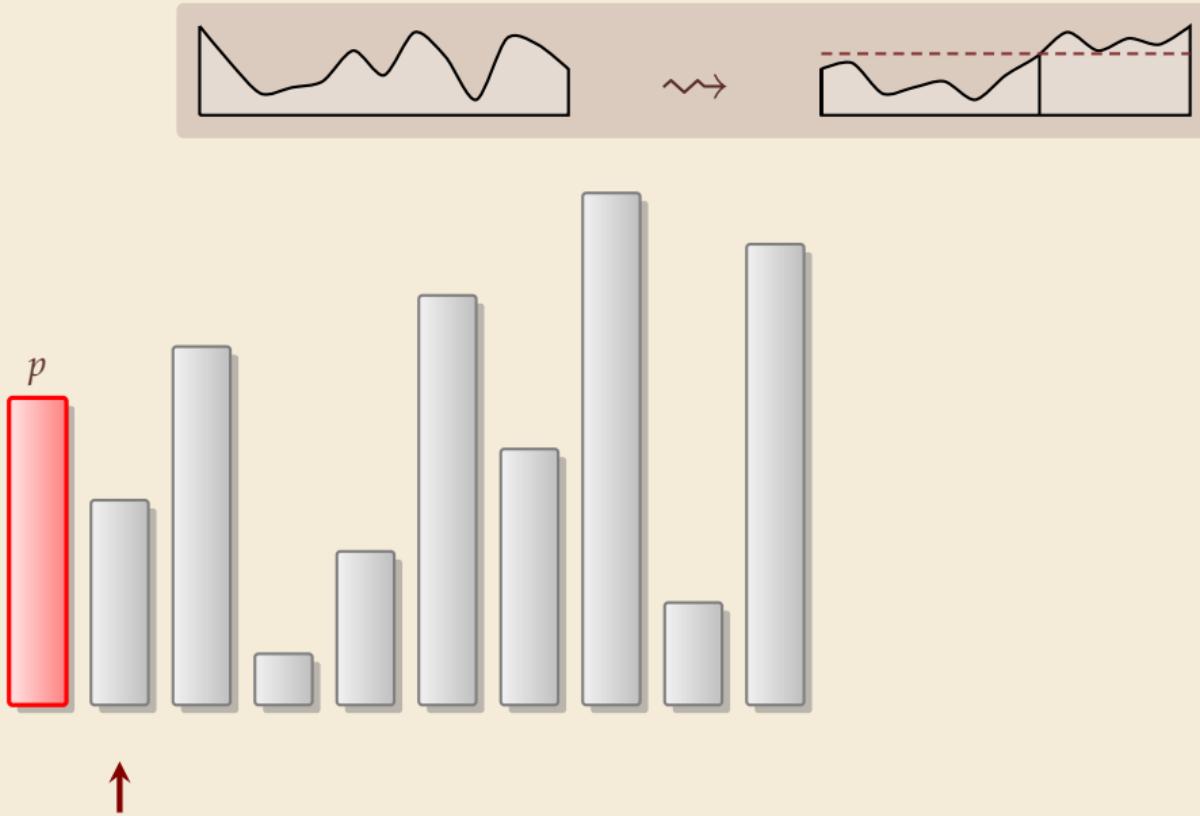
$$C(n) = 2n \lg(n) = \Theta(n \log n)$$

# Mergesort – Discussion

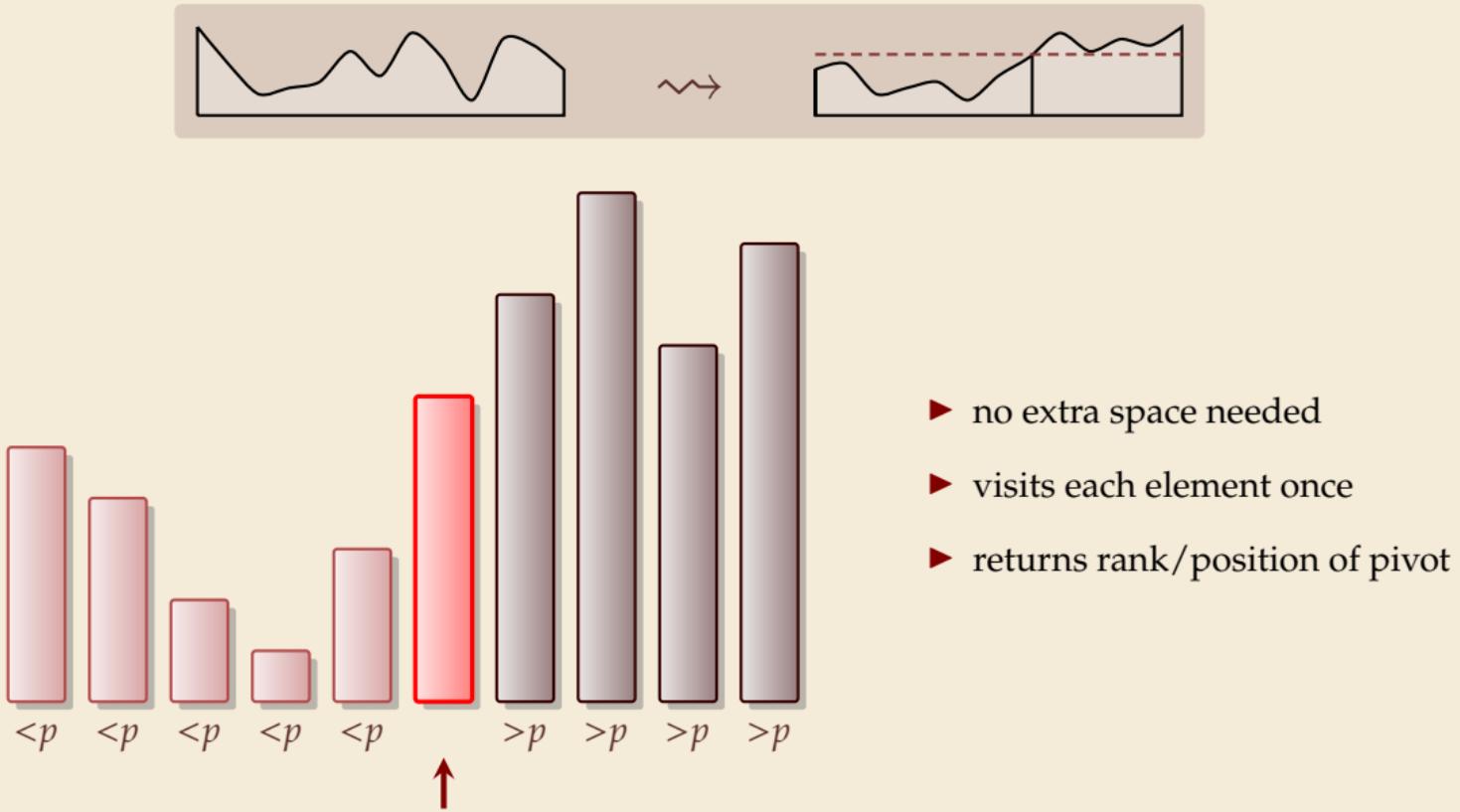
- 👍 optimal time complexity of  $\Theta(n \log n)$  in the worst case
- 👍 *stable* sorting method      i. e., retains relative order of equal-key items
- 👍 memory access is sequential (scans over arrays)
- 👎 requires  $\Theta(n)$  extra space
  - ↗ there are in-place merging methods,  
but they are substantially more complicated  
and not (widely) used

## 3.2 Quicksort

## Partitioning around a pivot



# Partitioning around a pivot



# Partitioning – Detailed code

Beware: details easy to get wrong; use this code!

---

```
1 procedure partition( $A, b$ )
2     // input: array  $A[0..n - 1]$ , position of pivot  $b \in [0..n - 1]$ 
3     swap( $A[0], A[b]$ )
4      $i := 0, j := n$ 
5     while true do
6         do  $i := i + 1$  while  $i < n$  and  $A[i] < A[0]$ 
7         do  $j := j - 1$  while  $j \geq 1$  and  $A[j] > A[0]$ 
8         if  $i \geq j$  then break (goto 8)
9         else swap( $A[i], A[j]$ )
10    end while
11    swap( $A[0], A[j]$ )
12    return  $j$ 
```

---

Loop invariant (5–10):

$A$	$p$	$\leq p$	$?$	$\geq p$
		$i$		$j$

# Quicksort

---

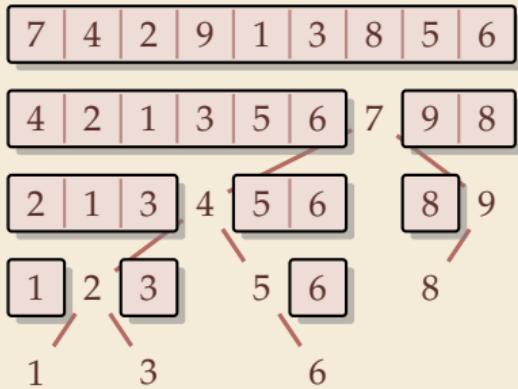
```
1 procedure quicksort( $A[l..r]$ )
2   if  $l \geq r$  then return
3    $b := \text{choosePivot}(A[l..r])$ 
4    $j := \text{partition}(A[l..r], b)$ 
5   quicksort( $A[l..j - 1]$ )
6   quicksort( $A[j + 1..r]$ )
```

---

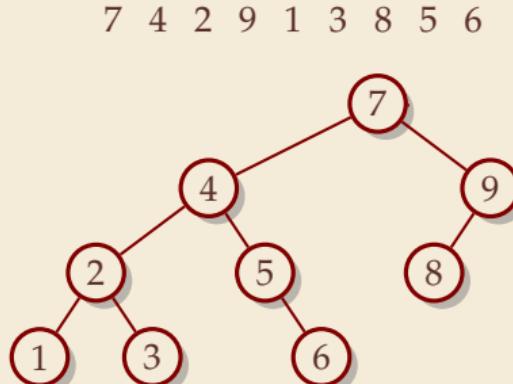
- ▶ recursive procedure; *divide & conquer*
- ▶ choice of pivot can be
  - ▶ fixed position  $\rightsquigarrow$  dangerous!
  - ▶ random
  - ▶ more sophisticated, e. g., median of 3

# Quicksort & Binary Search Trees

Quicksort



Binary Search Tree (BST)



- recursion tree of quicksort = binary search tree from successive insertion
- comparisons in quicksort = comparisons to built BST
- comparisons in quicksort  $\approx$  comparisons to search each element in BST

# Quicksort – Worst Case

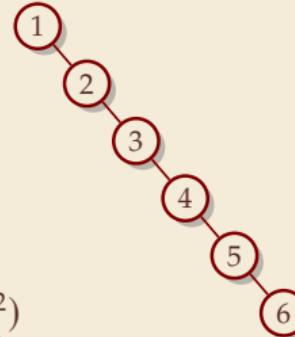
- ▶ Problem: BSTs can degenerate

- ▶ Cost to search for  $k$  is  $k - 1$

$$\rightsquigarrow \text{Total cost } \sum_{k=1}^n (k - 1) = \frac{n(n - 1)}{2} \sim \frac{1}{2}n^2$$

$\rightsquigarrow$  quicksort worst-case running time is in  $\Theta(n^2)$

terribly slow!



But, we can fix this:

## Randomized quicksort:

- ▶ choose a *random pivot* in each step

$\rightsquigarrow$  same as randomly *shuffling* input before sorting

# Randomized Quicksort – Analysis

- ▶  $C(n)$  = element visits (as for mergesort)
  - ~~ quicksort needs  $\sim 2 \ln(2) \cdot n \lg n \approx 1.39n \lg n$  *in expectation*
- ▶ also: very unlikely to be much worse:
  - e.g., one can prove:  $\Pr[\text{cost} > 10n \lg n] = O(n^{-2.5})$
  - distribution of costs is “concentrated around mean”
- ▶ intuition: have to be *constantly* unlucky with pivot choice

# Quicksort – Discussion

- thumb up fastest general-purpose method
- thumb up  $\Theta(n \log n)$  average case
- thumb up works *in-place* (no extra space required)
- thumb up memory access is sequential (scans over arrays)
- thumb down  $\Theta(n^2)$  worst case (although extremely unlikely)
- thumb down not a *stable* sorting method

Open problem: Simple algorithm that is fast, stable and in-place.

### 3.3 Comparison-Based Lower Bound

# Lower Bounds

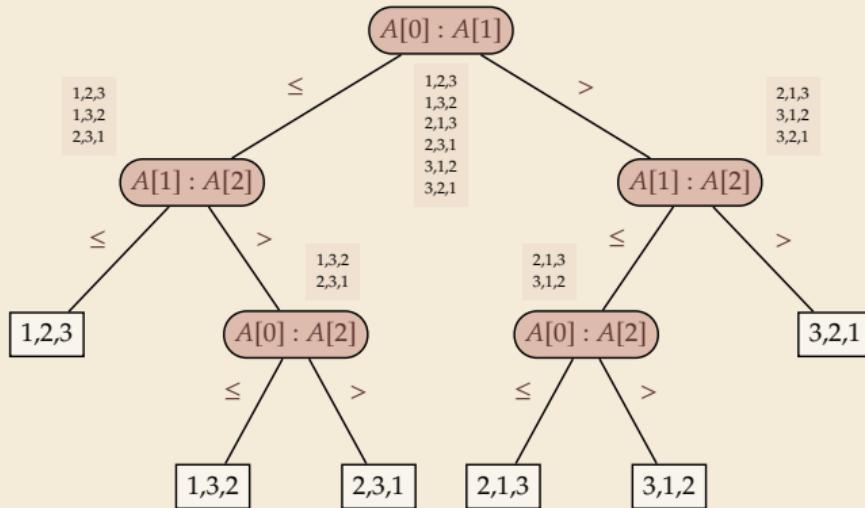
- ▶ **Lower bound:** mathematical proof that *no algorithm* can do better.
  - ▶ very powerful concept: bulletproof *impossibility* result
    - ≈ *conservation of energy* in physics
  - ▶ **(unique?) feature of computer science:**
    - for many problems, solutions are known that (asymptotically) **achieve the lower bound**
    - ~~ can speak of "*optimal* algorithms"
- ▶ To prove a statement about *all algorithms*, we must precisely define what that is!
- ▶ already know one option: the word-RAM model
- ▶ Here: use a simpler, more restricted model.

# The Comparison Model

- ▶ In the *comparison model* data can only be accessed in two ways:
    - ▶ comparing two elements
    - ▶ moving elements around (e.g. copying, swapping)
    - ▶ Cost: number of these operations.
  - ▶ This makes very few assumptions on the kind of objects we are sorting.
    - That's good!  
Keeps algorithms general!
  - ▶ Mergesort and Quicksort work in the comparison model.
- ~~ Every comparison-based sorting algorithm corresponds to a *decision tree*.
- ▶ only model comparisons    ~~ ignore data movement
  - ▶ nodes = comparisons the algorithm does
  - ▶ next comparisons can depend on outcomes    ~~ different subtrees
  - ▶ child links = outcomes of comparison
  - ▶ leaf = unique initial input permutation compatible with comparison outcomes

# Comparison Lower Bound

Example: Comparison tree for a sorting method for  $A[0..2]$ :



- ▶ Execution = follow a path in comparison tree.
- ~~> height of comparison tree = worst-case # comparisons
- ▶ comparison trees are *binary* trees
- ~~>  $\ell$  leaves ~~> height  $\geq \lceil \lg(\ell) \rceil$
- ▶ comparison trees for sorting method must have  $\geq n!$  leaves
- ~~> height  $\geq \lg(n!) \sim n \lg n$

more precisely:  $\lg(n!) = n \lg n - \lg(e)n + O(\log n)$

$\approx 1.4427$

- ▶ Mergesort achieves  $\sim n \lg n$  comparisons ~~> asymptotically comparison-optimal!
- ▶ Open (theory) problem: Sorting algorithm with  $n \lg n - \lg(e)n + o(n)$  comparisons?

## 3.4 Integer Sorting

# How to beat a lower bound

- ▶ Does the above lower bound mean, sorting always takes time  $\Omega(n \log n)$ ?
- ▶ **Not necessarily;** only in the *comparison model*!
  - ~~ Lower bounds show where to *change* the model!
- ▶ Here: sort  $n$  **integers**
  - ▶ can do *a lot* with integers: add them up, compute averages, ... (full power of word-RAM)
  - ~~ we are **not** working in the comparison model
  - ~~ *above lower bound does not apply!*
- ▶ but: a priori unclear how much arithmetic helps for sorting ...

# Counting sort

- ▶ Important parameter: size/range of numbers
  - ▶ numbers in range  $[0..U) = \{0, \dots, U - 1\}$  typically  $U = 2^b \rightsquigarrow b$ -bit binary numbers
- ▶ We can sort  $n$  integers in  $\Theta(n + U)$  time and  $\Theta(U)$  space when  $b \leq w$ :

word size

## Counting sort

---

```
1 procedure countingSort(A[0..n - 1])
2   // A contains integers in range [0..U].
3   C[0..U - 1] := new integer array, initialized to 0
4   // Count occurrences
5   for i := 0, ..., n - 1
6     C[A[i]] := C[A[i]] + 1
7   i := 0 // Produce sorted list
8   for k := 0, ..., U - 1
9     for j := 1, ..., C[k]
10      A[i] := k; i := i + 1
```

---

- ▶ count how often each possible value occurs
- ▶ produce sorted result directly from counts
- ▶ circumvents lower bound by using integers as array index / pointer offset

~ Can sort  $n$  integers in range  $[0..U)$  with  $U = O(n)$  in time and space  $\Theta(n)$ .

# Integer Sorting – State of the art

- ▶  $O(n)$  time sorting also possible for numbers in range  $U = O(n^c)$  for constant  $c$ .
  - ▶ radix sort with radix  $2^w$
- ▶ Algorithm theory
  - ▶ suppose  $U = 2^w$ , but  $w$  can be an arbitrary function of  $n$
  - ▶ how fast can we sort  $n$  such  $w$ -bit integers on a  $w$ -bit word-RAM?
    - ▶ for  $w = O(\log n)$ : linear time (*radix/counting sort*)
    - ▶ for  $w = \Omega(\log^{2+\varepsilon} n)$ : linear time (*signature sort*)
    - ▶ for  $w$  in between: can do  $O(n\sqrt{\lg \lg n})$  (very complicated algorithm)  
don't know if that is best possible!

\* \* \*

- ▶ for the rest of this unit: back to the comparisons model!

# Part II

*Sorting with many processors*

## 3.5 Parallel computation

# Types of parallel computation

£££ can't buy you more time . . . but more computers!

~ Challenge: Algorithms for *parallel* computation.

There are two main forms of parallelism:

## 1. shared-memory parallel computer $\leftarrow$ focus of today

- ▶  $p$  *processing elements* (PEs, processors) working in parallel
- ▶ **single big memory, accessible from every PE**
- ▶ communication via shared memory
- ▶ think: a big server, 128 CPU cores, terabyte of main memory

## 2. distributed computing

- ▶  $p$  PEs working in parallel
- ▶ each PE has **private** memory
- ▶ communication by sending **messages** via a network
- ▶ think: a cluster of individual machines

# PRAM – Parallel RAM

- ▶ extension of the RAM model (recall Unit 1)
- ▶ the  $p$  PEs are identified by ids  $0, \dots, p - 1$ 
  - ▶ like  $w$  (the word size),  $p$  is a parameter of the model that can grow with  $n$
  - ▶  $p = \Theta(n)$  is not unusual      maaany processors!
- ▶ the PEs all **independently** run a RAM-style program  
(they can use their id there)
- ▶ each PE has its own registers, but **MEM** is shared among all PEs
- ▶ computation runs in **synchronous** steps:  
in each time step, every PE executes one instruction

# PRAM – Conflict management



**Problem:** What if several PEs simultaneously overwrite a memory cell?

- ▶ **EREW-PRAM** (exclusive read, exclusive write)  
any **parallel access** to same memory cell is **forbidden** (crash if happens)
- ▶ **CREW-PRAM** (concurrent read, exclusive write)  
parallel **write** access to same memory cell is *forbidden*, but reading is fine
- ▶ **CRCW-PRAM** (concurrent read, concurrent write)  
concurrent access is allowed,  
need a rule for write conflicts:
  - ▶ common CRCW-PRAM:  
all concurrent writes to same cell must write *same* value
  - ▶ arbitrary CRCW-PRAM:  
some unspecified concurrent write wins
  - ▶ (more exist . . . )
- ▶ no single model is always adequate, but our default is CREW

# PRAM – Execution costs

Cost metrics in PRAMs

- ▶ **space:** total amount of accessed memory
- ▶ **time:** number of steps till all PEs finish      assuming sufficiently many PEs!  
sometimes called *depth* or *span*
- ▶ **work:** total #instructions executed on **all** PEs

Holy grail of PRAM algorithms:

- ▶ minimal time
- ▶ work (asymptotically) no worse than running time of best sequential algorithm  
~~ “*work-efficient*” algorithm: work in same  $\Theta$ -class as best sequential

# The number of processors

*Hold on, my computer does not have  $\Theta(n)$  processors! Why should I care for span and work!?*

## Theorem 3.1 (Brent's Theorem):

If an algorithm has span  $T$  and work  $W$  (for an arbitrarily large number of processors), it can be run on a PRAM with  $p$  PEs in time  $O(T + \frac{W}{p})$  (and using  $O(W)$  work). ◀

*Proof:* schedule parallel steps in round-robin fashion on the  $p$  PEs.

- ~~ span and work give guideline for *any* number of processors

## 3.6 Parallel primitives

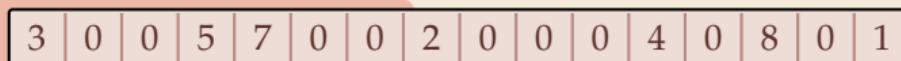
# Prefix sums

Before we come to parallel sorting, we study some useful building blocks.

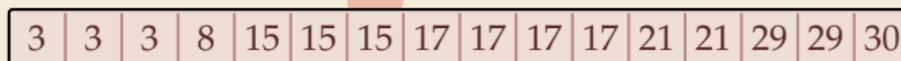
**Prefix-sum problem** (also: cumulative sums, running totals)

- ▶ Given: array  $A[0..n - 1]$  of numbers
- ▶ Goal: compute all prefix sums  $A[0] + \dots + A[i]$  for  $i = 0, \dots, n - 1$   
may be done “in-place”, i. e., by overwriting  $A$

**Example:**

input:  3 | 0 | 0 | 5 | 7 | 0 | 0 | 2 | 0 | 0 | 0 | 4 | 0 | 8 | 0 | 1

$\Sigma$

output: 

# Prefix sums – Sequential

- ▶ sequential solution does  $n - 1$  additions
- ▶ but: cannot parallelize them!
  - ⚡ data dependencies!
- ~~ need a different approach

Let's try a simpler problem first.

## Excuse: Sum

- ▶ Given: array  $A[0..n - 1]$  of numbers
- ▶ Goal: compute  $A[0] + A[1] + \dots + A[n - 1]$   
(solved by prefix sums)

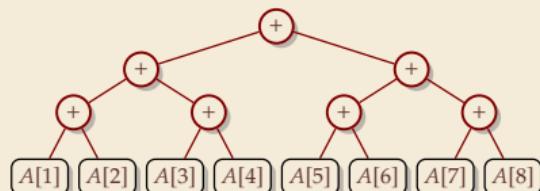
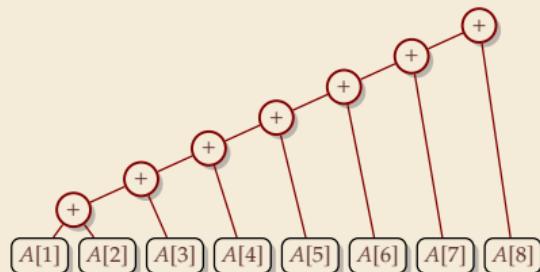
Any algorithm *must* do  $n - 1$  binary additions

- ~~ Height of tree = parallel time!

---

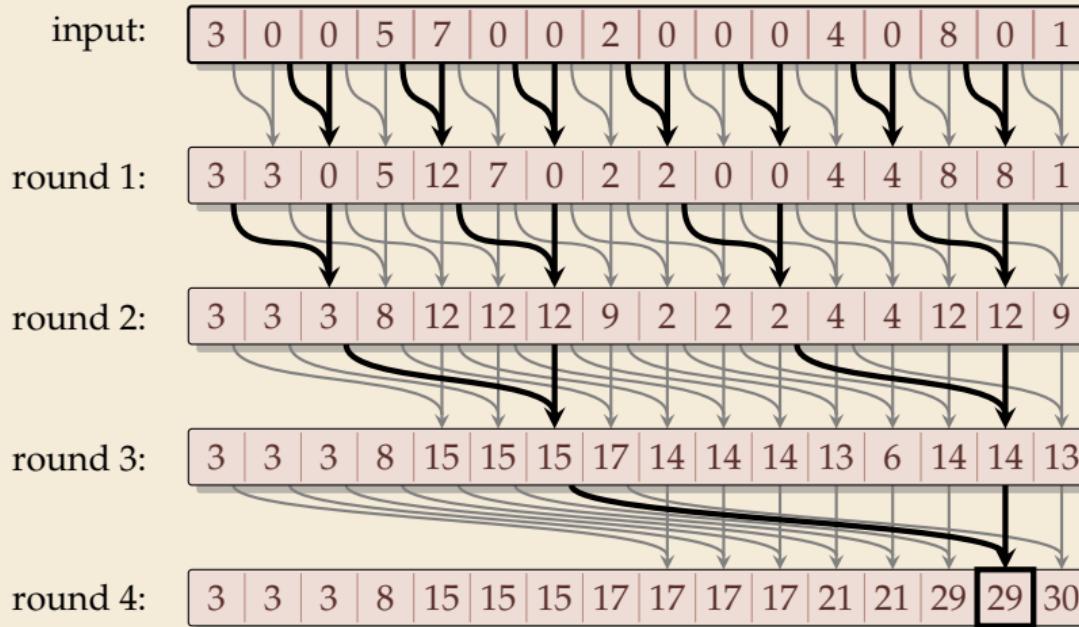
```
1 procedure prefixSum(A[0..n - 1])
2   for i := 1, ..., n - 1 do
3     A[i] := A[i - 1] + A[i]
```

---



# Parallel prefix sums

- Idea: Compute all prefix sums with balanced trees in parallel  
Remember partial results for reuse



## Parallel prefix sums – Code

- ▶ can be realized in-place (overwriting  $A$ )
- ▶ assumption: in each parallel step, all reads precede all writes

---

```
1 procedure parallelPrefixSums( $A[0..n - 1]$ )
2   for  $r := 1, \dots, \lceil \lg n \rceil$  do
3     step :=  $2^{r-1}$ 
4     for  $i := step, \dots, n - 1$  do in parallel
5        $x := A[i] + A[i - step]$ 
6        $A[i] := x$ 
7     end parallel for
8   end for
```

---

# Parallel prefix sums – Analysis

## ► Time:

- ▶ all additions of one round run in parallel
- ▶  $\lceil \lg n \rceil$  rounds
- $\rightsquigarrow \Theta(\log n)$  time      best possible!

## ► Work:

- ▶  $\geq \frac{n}{2}$  additions in all rounds (except maybe last round)
  - $\rightsquigarrow \Theta(n \log n)$  work
  - ▶ more than the  $\Theta(n)$  sequential algorithm!
- 
- ▶ Typical trade-off: greater parallelism at the expense of more overall work
  - ▶ For prefix sums:
    - ▶ can actually get  $\Theta(n)$  work in *twice* that time!
    - $\rightsquigarrow$  algorithm is slightly more complicated
    - ▶ instead here: linear work in *thrice* the time using “blocking trick”

# Work-efficient parallel prefix sums

standard trick to improve work: compute small blocks sequentially

1. Set  $b := \lceil \lg n \rceil$
2. For blocks of  $b$  consecutive indices, i. e.,  $A[0..b), A[b..2b), \dots$  do in parallel:  
compute local prefix sums sequentially
3. Use previous work-inefficient algorithm only on rightmost elements of block,  
i. e., to compute prefix sums of  $A[b - 1], A[2b - 1], A[3b - 1], \dots$
4. For blocks  $A[0..b), A[b..2b), \dots$  do in parallel:  
Add block-prefix sums to local prefix sums

Analysis:

► Time:

- 2. & 4.:  $\Theta(b) = \Theta(\log n)$  time
- 3.  $\Theta(\log(n/b)) = \Theta(\log n)$  times

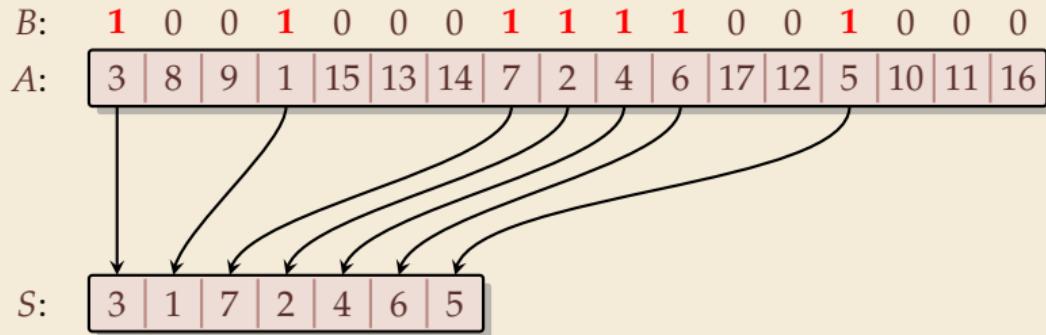
► Work:

- 2. & 4.:  $\Theta(b)$  per block  $\times \lceil \frac{n}{b} \rceil$  blocks  $\rightsquigarrow \Theta(n)$
- 3.  $\Theta\left(\frac{n}{b} \log\left(\frac{n}{b}\right)\right) = \Theta(n)$

# Compacting subsequences

How do prefix sums help with sorting? one more step to go ...

**Goal:** *Compact* a subsequence of an array



Use prefix sums on bitvector  $B$

↝ offset of selected cells in  $S$

---

```
1 parallelPrefixSums( $B$ )
2 for  $j := 0, \dots, n - 1$  do in parallel
3   if  $B[j] == 1$  then  $S[B[j] - 1] := A[j]$ 
4 end parallel for
```

---

## 3.7 Parallel sorting

# Parallel quicksort

Let's try to parallelize quicksort

- ▶ recursive calls can run in parallel (data independent)
- ▶ our sequential partitioning algorithm seems hard to parallelize
- ▶ but can split partitioning into *rounds*:
  1. **comparisons**: compare all elements pivot (in parallel), store bitvector
  2. compute prefix sums of bit vectors (in parallel as above)
  3. **compact** subsequences of small and large elements (in parallel as above)

# Parallel quicksort – Code

---

```
1 procedure parQuicksort( $A[l..r]$ )
2      $b := \text{choosePivot}(A[l..r])$ 
3      $j := \text{parallelPartition}(A[l..r], b)$ 
4     in parallel {  $\text{parQuicksort}(A[l..j - 1])$ ,  $\text{parQuicksort}(A[j + 1..r])$  }
5
6 procedure parallelPartition( $A[l..r]$ ,  $b$ )
7      $\text{swap}(A[n - 1], A[b]); p := A[n - 1]$ 
8     for  $i = 0, \dots, n - 2$  do in parallel
9          $S[i] := [A[i] \leq p]$  //  $S[i]$  is 1 or 0
10         $L[i] := 1 - S[i]$ 
11    end parallel for
12    in parallel {  $\text{parallelPrefixSum}(S[0..n - 2])$ ;  $\text{parallelPrefixSum}(L[0..n - 2])$  }
13     $j := S[n - 2] + 1$ 
14    for  $i = 0, \dots, n - 2$  do in parallel
15         $x := A[i]$ 
16        if  $x \leq p$  then  $A[S[i] - 1] := x$ 
17        else  $A[j + L[i]] := x$ 
18    end parallel for
19     $A[j] := p$ 
20    return  $j$ 
```

---

# Parallel quicksort – Analysis

## ► Time:

- ▶ partition: all  $O(1)$  time except prefix sums  $\rightsquigarrow \Theta(\log n)$  time
- ▶ quicksort: expected depth of recursion tree is  $\Theta(\log n)$ 
  - $\rightsquigarrow$  total time  $O(\log^2(n))$  in expectation

## ► Work:

- ▶ partition:  $O(n)$  time except prefix sums  $\rightsquigarrow \Theta(n \log n)$  work
  - $\rightsquigarrow$  quicksort  $O(n \log^2(n))$  work in expectation
- ▶ using a work-efficient prefix-sums algorithm yields (expected) work-efficient sorting!

## Parallel mergesort

- ▶ As for quicksort, recursive calls can run in parallel ✓
- ▶ how about merging sorted halves  $A[l..m - 1]$  and  $A[m..r]$ ?
- ▶ Must treat elements independently.
  - ▶ correct position of  $x$  in sorted output =  $\text{rank}$  of  $x$  breaking ties by position in  $A$   
 $\text{rank}$  of  $x$  #elements  $\leq x$
  - ▶  $\# \text{ elements} \leq x = \# \text{ elements from } A[l..m - 1] \text{ that are } \leq x + \# \text{ elements from } A[m..r] \text{ that are } \leq x$
  - ▶ Note: rank in own run is simply the index of  $x$  in that run
  - ▶ find rank in *other* run by binary search
    - ~~ can move it to correct position

# Parallel mergesort – Analysis

## ► Time:

- ▶ merge:  $\Theta(\log n)$  from binary search, rest  $O(1)$
- ▶ mergesort: depth of recursion tree is  $\Theta(\log n)$ 
  - ~~ total time  $O(\log^2(n))$

## ► Work:

- ▶ merge:  $n$  binary searches  $\rightsquigarrow \Theta(n \log n)$ 
  - ~~ mergesort:  $O(n \log^2(n))$  work
- ▶ work can be reduced to  $\Theta(n)$  for merge
  - ▶ do full binary searches only for regularly sampled elements
  - ▶ ranks of remaining elements are sandwiched between sampled ranks
  - ▶ use a sequential method for small blocks, treat blocks in parallel
  - ▶ (detailed omitted)

## Parallel sorting – State of the art

- ▶ more sophisticated methods can sort in  $O(\log n)$  parallel time on CREW-RAM
- ▶ practical challenge: small units of work add overhead
- ▶ need a lot of PEs to see improvement from  $O(\log n)$  parallel time
- ~~ implementations tend to use simpler methods above
  - ▶ check the Java library sources for interesting examples!  
`java.util.Arrays.parallelSort(int[])`