3 Efficient Sorting

24 February 2021

Sebastian Wild

version 2021-03-02 13:57 H

Outline

3 Efficient Sorting

Why study sorting?

- fundamental problem of computer science that is still not solved

  Algorithm with optimal #comparisons in worst case?

- building brick of many more advanced algorithms
  - for preprocessing
  - as subroutine
- playground of manageable complexity to practice algorithmic techniques

Here:

- "classic" fast sorting method
- **parallel** sorting

Part I

The Basics

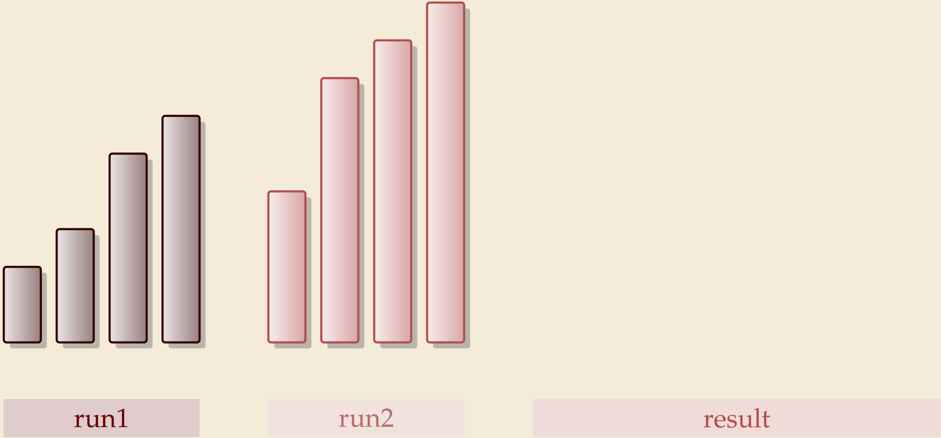
Rules of the game

- **Given:**
  - array $A[0..n-1]$ of n objects
  - a total order relation $\leq$ among $A[0], \ldots, A[n-1]$
    (a comparison function)
- **Goal:** rearrange (=permute) elements within A,
  so that A is *sorted*, i. e., $A[0] \leq A[1] \leq \cdots \leq A[n-1]$

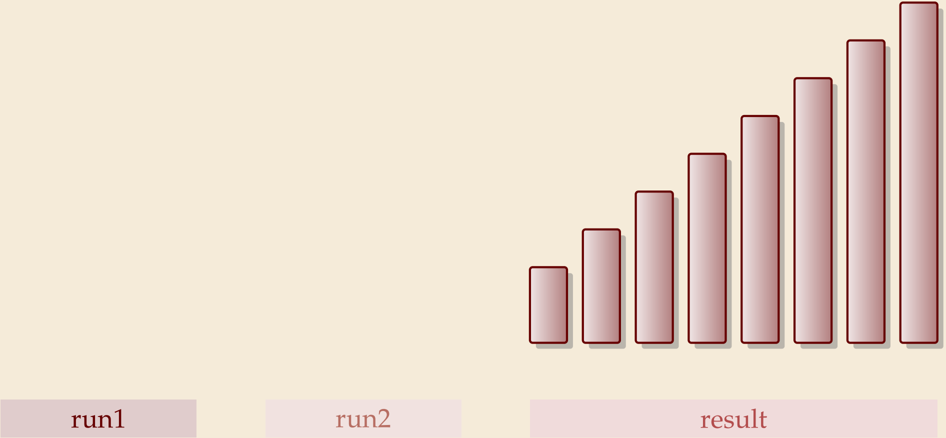
- for now: A stored in main memory (*internal sorting*)
  single processor (*sequential sorting*)

3.1 Mergesort

Merging sorted lists



Merging sorted lists



Mergesort

procedure mergesort($A[l..r]$)
  $n := r - l + 1$
  if $n \geq 1$ **return**
  $m := l + \left\lfloor \frac{n}{2} \right\rfloor$
  mergesort($A[l..m-1]$)
  mergesort($A[m..r]$)
  merge($A[l..m-1]$, $A[m..r]$, buf)
  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 $\boxed{n = 2^k}$

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

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

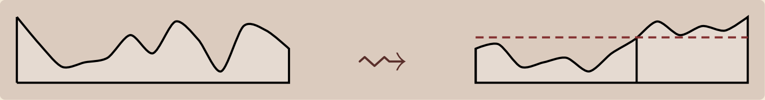
Mergesort – Discussion

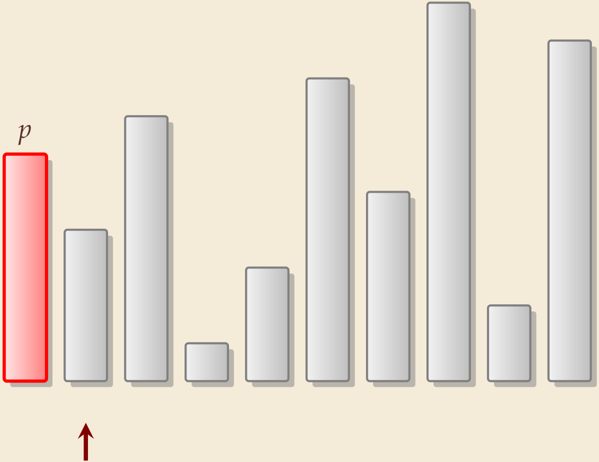
- optimal time complexity of $\Theta(n \log n)$ in the worst case
- *stable* sorting method — i. e., retains relative order of equal-key items
- memory access is sequential (scans over arrays)
- requires $\Theta(n)$ extra space

there are in-place merging methods,
but they are substantially more complicated
and not (widely) used

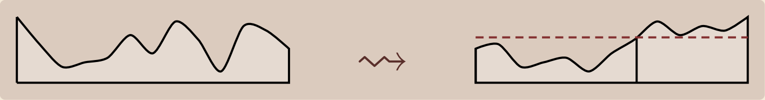
3.2 Quicksort

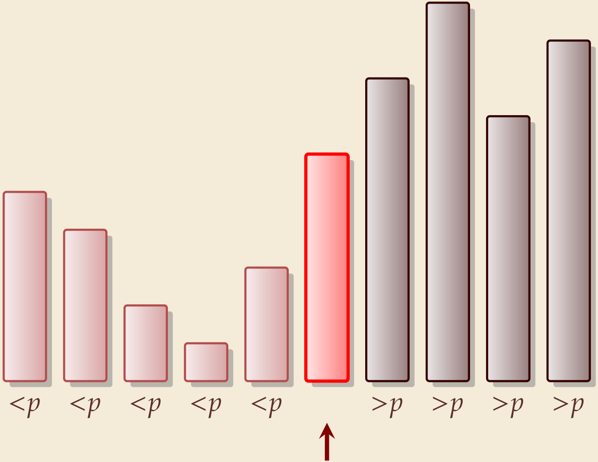
Partitioning around a pivot





Partitioning around a pivot





- no extra space needed
- visits each element once
- returns rank/position of 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 ∈ [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 ≥ 1 and A[j] > A[0]
8          if i ≥ 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

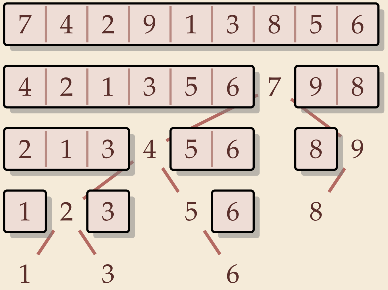
Quicksort

```
procedure quicksort(A[l..r])
    if l ≥ r then return
    b := choosePivot(A[l..r])
    j := partition(A[l..r], b)
    quicksort(A[l..j − 1])
    quicksort(A[j + 1..r])
```

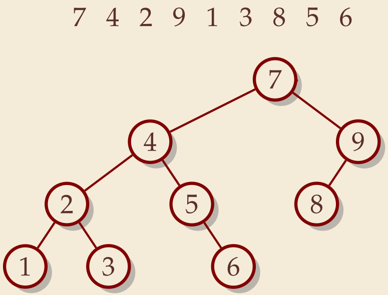
- recursive procedure; *divide & conquer*
- choice of pivot can be
  - fixed position $\rightsquigarrow$ dangerous!
  - random
  - more sophisticated, e. g., median of 3

Quicksort & Binary Search Trees

Quicksort

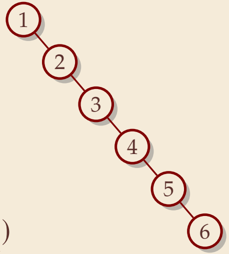


Binary Search Tree (BST)



- 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 $k-1$

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

$\rightsquigarrow$ 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)

$\rightsquigarrow$ quicksort needs $\sim 2\ln(2) \cdot n \lg n \approx 1.39 n \lg n$ *in expectation*

- also: very unlikely to be much worse:
  e. g., one can prove: $\Pr[\text{cost} > 10 n \lg n] \;=\; O(n^{-2.5})$
  distribution of costs is "concentrated around mean"
- intuition: have to be *constantly* unlucky with pivot choice

Quicksort – Discussion

- 👍 fastest general-purpose method
- 👍 $\Theta(n \log n)$ average case
- 👍 works *in-place* (no extra space required)
- 👍 memory access is sequential (scans over arrays)
- 👎 $\Theta(n^2)$ worst case (although extremely unlikely)
- 👎 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
    $\approx$ *conservation of energy* in physics
  - **(unique?) feature of computer science:**
    for many problems, solutions are known that (asymptotically) **achieve the lower bound**

  $\rightsquigarrow$ 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.

That's good!
Keeps algorithms general!

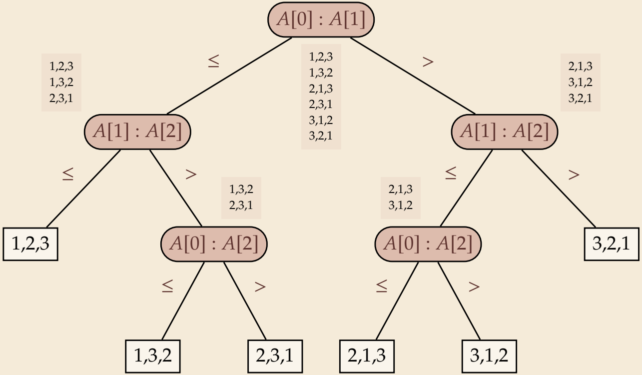
- This makes very few assumptions on the kind of objects we are sorting.

- Mergesort and Quicksort work in the comparison model.

$\rightsquigarrow$ Every comparison-based sorting algorithm corresponds to a ***decision tree***.
  - only model comparisons $\rightsquigarrow$ ignore data movement
  - nodes $=$ comparisons the algorithm does
  - next comparisons can depend on outcomes $\rightsquigarrow$ 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.

  $\rightsquigarrow$ height of comparison tree = worst-case # comparisons

- comparison trees are *binary* trees

  $\rightsquigarrow$ ℓ leaves $\rightsquigarrow$ height $\geq \lceil \lg(\ell) \rceil$

- comparison trees for sorting method must have $\geq n!$ leaves

  $\rightsquigarrow$ height $\geq \lg(n!) \sim n \lg n$

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

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

  ≈ 1.4427

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!*
  - $\rightsquigarrow$ 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)
  - $\rightsquigarrow$ we are **not** working in the comparison model
  - $\rightsquigarrow$ *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 $[0..U) = \{0, \ldots, U-1\}$ typically $U = 2^b$ $\rightsquigarrow$ b-bit binary numbers
- We can sort n integers in $\Theta(n+U)$ time and $\Theta(U)$ space when $\boxed{b \le w}$:

word size

Counting sort

```
procedure countingSort(A[0..n − 1])
    // A contains integers in range [0..U).
    C[0..U − 1] := new integer array, initialized to 0
    // Count occurrences
    for i := 0, . . . , n − 1
        C[A[i]] := C[A[i]] + 1
    i := 0 // Produce sorted list
    for k := 0, . . . U − 1
        for j := 1, . . . C[k]
            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

$\rightsquigarrow$ 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 of many processors

3.5 Parallel computation

Types of parallel computation

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

$\rightsquigarrow$ Challenge: Algorithms for *parallel* computation.

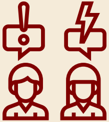
There are two main forms of parallelism:

1. **shared-memory parallel computer** ← *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 $0, \ldots, p-1$
  - 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
  $\rightsquigarrow$ *"work-efficient"* algorithm: work in same Θ-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.

$\rightsquigarrow$ 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 $A[0] + \cdots + A[i]$ for $i = 0, \ldots, n-1$
  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
							Σ									
output:	3	3	3	8	15	15	15	17	17	17	17	21	21	29	29	30

Prefix sums – Sequential

- sequential solution does $n-1$ additions
- but: cannot parallelize them!
  ϟ data dependencies!

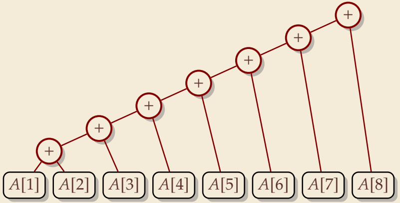
⇝ need a different approach

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

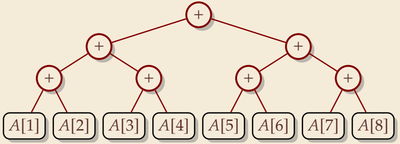
Let's try a simpler problem first.

Excursion: Sum

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



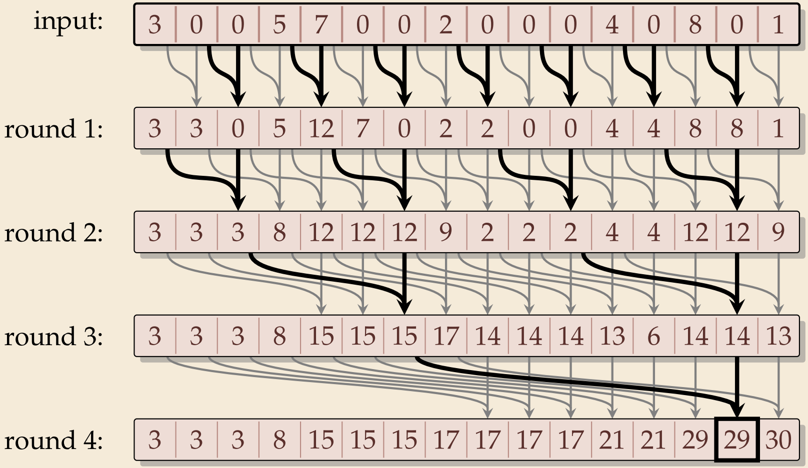
Any algorithm *must* do $n-1$ binary additions



⇝ Depth of tree = parallel time!

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

```
procedure parallelPrefixSums(A[0..n − 1])
    for r := 1, . . . ⌈lg n⌉ do
        step := 2^(r−1)
        for i := step, . . . n − 1 do in parallel
            A[i] := A[i] + A[i − step]
        end parallel for
    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., $A[0..b)$, $A[b..2b)$, … 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 $A[b-1]$, $A[2b-1]$, $A[3b-1]$, …
4. For blocks $A[0..b)$, $A[b..2b)$, … do in parallel: Add block-prefix sums to local prefix sums

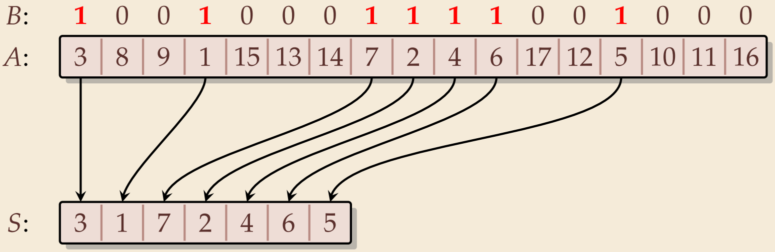
Analysis:

- **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

$\leadsto$ offset of selected cells in S

```
parallelPrefixSums(B)
for j := 0, ..., n − 1 do in parallel
        if B[j] == 1 then S[B[j] − 1] := A[j]
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

```
procedure parQuicksort(A[l..r])
    b := choosePivot(A[l..r])
    j := parallelPartition(A[l..r], b)
    in parallel { parQuicksort(A[l..j − 1]), parQuicksort(A[j + 1..r]) }

procedure parallelPartition(A[l..r], b)
    swap(A[n − 1], A[b]);  p := A[n − 1]
    for i = 0, . . . , n − 2 do in parallel
        S[i] := [A[i] ≤ p]      // S[i] is 1 or 0
        L[i] := 1 − S[i]
    end parallel for
    in parallel { parallelPrefixSum(S[0..n − 2]);  parallelPrefixSum(L[0..n − 2]) }
    j := S[n − 2] + 1
    for i = 0, . . . , n − 2 do in parallel
        x := A[i]
        if x ≤ p then A[S[i] − 1] := x
        else A[j + L[i]] := x
    end parallel for
    A[j] := p
    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 = *rank* of x (#elements $\le x$)
  - # elements $\le x$ = # elements from $A[l..m-1]$ that are $\le x$ + # elements from $A[m..r]$ that are $\le x$ (breaking ties by position in A)
  - Note: rank in own run is simply the index of x in that run
  - find rank in *other* run by binary search

  $\rightsquigarrow$ can move it to correct position

Parallel mergesort – Analysis

- **Time:**
  - merge: $\Theta(n)$ from binary search, rest $O(1)$
  - mergesort: depth of recursion tree is $\Theta(\log n)$

  $\leadsto$ total time $O(\log^2(n))$
- **Work:**
  - merge: n binary searches $\leadsto$ $\Theta(n \log n)$

  $\leadsto$ 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

$\rightsquigarrow$ implementations tend to use simpler methods above
  - check the Java library sources for interesting examples!
    `java.util.Arrays.parallelSort(int[])`