

3

Efficient Sorting

17 February 2022

Sebastian Wild

Learning Outcomes

1. Know principles and implementation of *mergesort* and *quicksort*.
2. Know properties and *performance characteristics* of mergesort and quicksort.
3. Know the comparison model and understand the corresponding *lower bound*.
4. Understand *counting sort* and how it circumvents the comparison lower bound.
5. Know ways how to exploit *presorted* inputs.
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*

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 Adaptive Sorting
- 3.6 Python's list sort
- 3.7 Parallel computation
- 3.8 Parallel primitives
- 3.9 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
- ▶ exploit **partially sorted** inputs
- ▶ **parallel** sorting

Part I

The Basics

Rules of the game

► Given:

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

(a comparison function)

Python: elements support \leq operator (`__lt__()`)

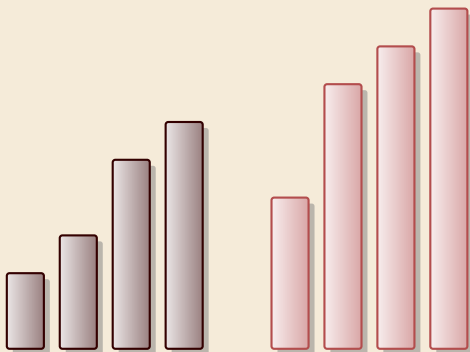
Java: Comparable class (`x.compareTo(y) <= 0`)

- **Goal:** rearrange (i. e., 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

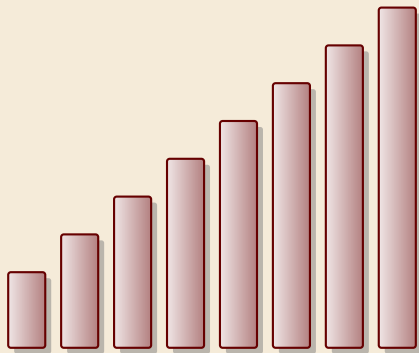


run1

run2

result

Merging sorted lists



run1

run2

result

Mergesort

```
1 procedure mergesort(A[l..r])
2   n := r - l
3   if n ≤ 1 return
4   m := l + ⌊ $\frac{n}{2}$ ⌋
5   mergesort(A[l..m])
6   mergesort(A[m..r])
7   merge(A[l..m], 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 $n = 2^k$

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

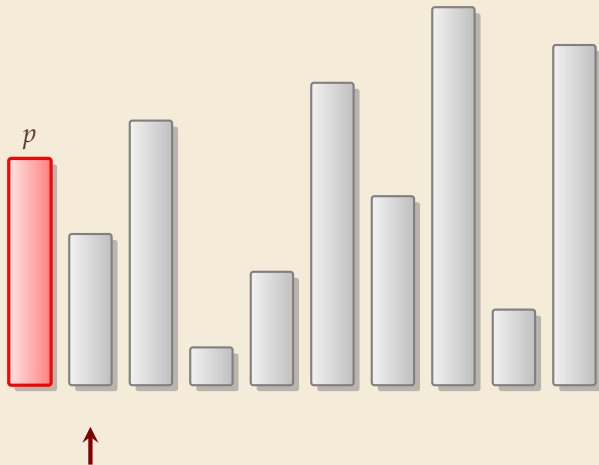
$$C(n) = 2n \lg(n) = \Theta(n \log n)$$

Mergesort – Discussion

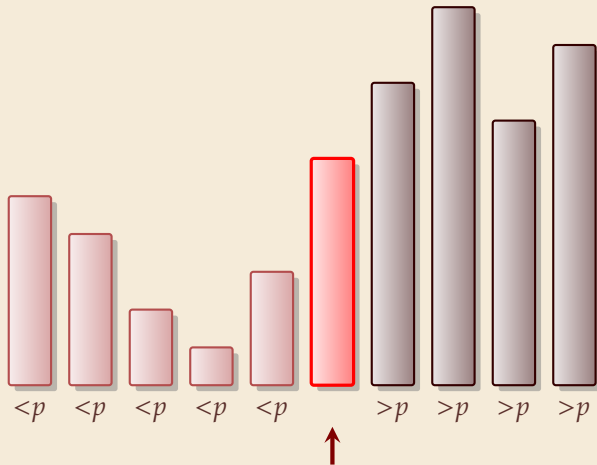
- 👍 optimal time complexity of $\Theta(n \log n)$ in the worst case
- 👍 *stable* sorting method i. e., retains relative order of equal-key items
- 👍 memory access is sequential (scans over arrays)
- 👎 requires $\Theta(n)$ extra space
 - there are in-place merging methods,
but they are substantially more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

3.2 Quicksort

Partitioning around a pivot



Partitioning around a pivot



- ▶ no extra space needed
- ▶ visits each element once
- ▶ returns rank/position of pivot

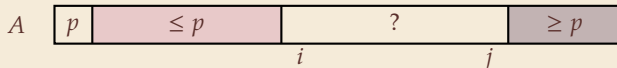
Partitioning – Detailed code

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

(if you ever have to)

```
1 procedure partition( $A, b$ )
2   // input: array  $A[0..n]$ , position of pivot  $b \in [0..n]$ 
3   swap( $A[0], A[b]$ )
4    $i := 0, \quad j := n$ 
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 11)
9     else swap( $A[i], A[j]$ )
10  end while
11  swap( $A[0], A[j]$ )
12  return  $j$ 
```

Loop invariant (5–10):



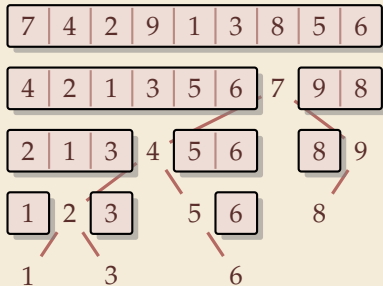
Quicksort

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

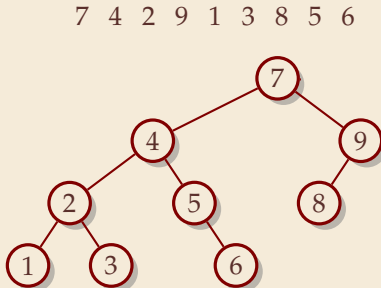
- ▶ recursive procedure; *divide & conquer*
- ▶ choice of pivot can be
 - ▶ fixed position \rightsquigarrow dangerous!
 - ▶ random
 - ▶ more sophisticated, e. g., median of 3

Quicksort & Binary Search Trees

Quicksort



Binary Search Tree (BST)



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

Quicksort – Worst Case

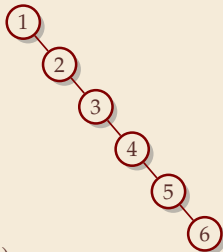
► Problem: BSTs can degenerate

► Cost to search for k is $k - 1$

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

~> 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

~> 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

- 👍 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.

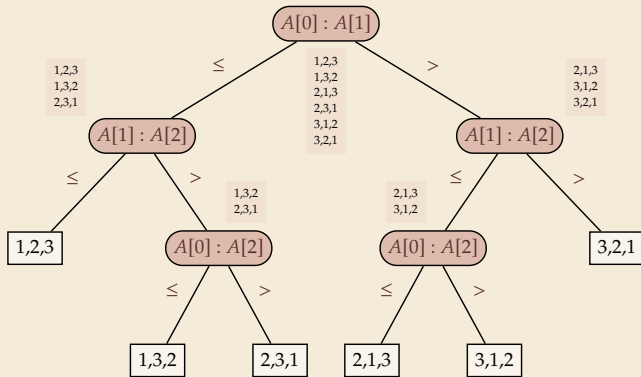
The Comparison Model

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

That's good!
Keeps algorithms general!

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

↪ ℓ 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: $\lg(n!) = n \lg n - \lg(e)n + O(\log n)$

► 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?

↪ ≈ 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*!
 - ~> 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( $A[0..n)$ )  
2   //  $A$  contains integers in range  $[0..U)$ .  
3    $C[0..U) :=$  new integer array, initialized to 0  
4   // Count occurrences  
5   for  $i := 0, \dots, n - 1$   
6      $C[A[i]] := C[A[i]] + 1$   
7    $i := 0$  // Produce sorted list  
8   for  $k := 0, \dots, U - 1$   
9     for  $j := 1, \dots, C[k]$   
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

\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

Exploiting presortedness

3.5 Adaptive Sorting

Adaptive sorting

- ▶ Comparison lower bound also holds for the *average case* $\rightsquigarrow \lfloor \lg(n!) \rfloor$ cmps necessary
- ▶ Mergesort and Quicksort from above use $\sim n \lg n$ cmps even in best case



Can we do better if the input is already “almost sorted”?

Scenarios where this may arise naturally:

- ▶ Append new data as it arrives, regularly sort entire list (e. g., log files, database tables)
- ▶ Compute summary statistics of time series of measurements that change slowly over time (e. g., weather data)
- ▶ Merging locally sorted data from different servers (e. g., map-reduce frameworks)

\rightsquigarrow Ideally, algorithms should *adapt* to input: *the more sorted the input, the faster the algorithm*
... but how to do that!?

Warmup: check for sorted inputs

- ▶ Any method could first check if input already completely in order!
 - 👍 Best case becomes $\Theta(n)$ with $n - 1$ comparisons!
 - 👎 Usually $n - 1$ extra comparisons and pass over data “wasted”
 - 👎 Only catches a single, extremely special case ...
- ▶ For divide & conquer algorithms, could check in each recursive call!
 - 👍 Potentially exploits partial sortedness!
 - 👎 usually adds $\Omega(n \log n)$ extra comparisons



For Mergesort, can instead check before merge with a **single** comparison

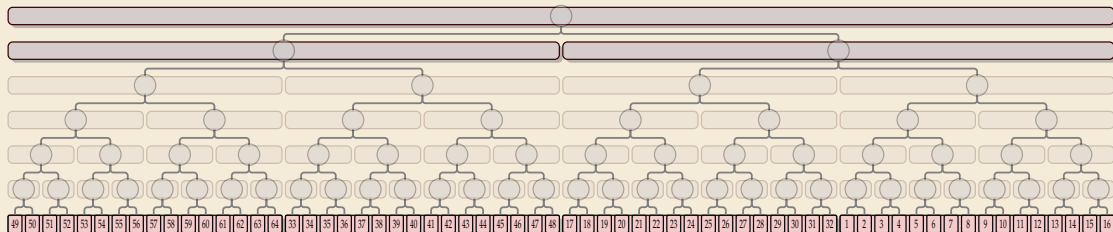
- ▶ If last element of first run \leq first element of second run, skip merge

How effective is this idea?

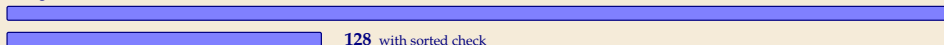
```
1 procedure mergesortCheck( $A[l..r]$ )
2    $n := r - l$ 
3   if  $n \leq 1$  return
4    $m := l + \lfloor \frac{n}{2} \rfloor$ 
5   mergesortCheck( $A[l..m]$ )
6   mergesortCheck( $A[m..r]$ )
7   if  $A[m - 1] > A[m]$ 
8     merge( $A[l..m]$ ,  $A[m..r]$ ,  $buf$ )
9   copy  $buf$  to  $A[l..r]$ 
```

Mergesort with sorted check – Analysis

- Simplified cost measure: *merge cost* = size of output of merges
 \approx number of comparisons
 \approx number of memory transfers / cache misses
- Example input: $n = 64$ numbers in sorted *runs* of 16 numbers each:



Merge costs:



384 Standard Mergesort

128 with sorted check

Sorted check can help a lot!

Alignment issues

► In previous example, each run of length ℓ saved us $\ell \lg(\ell)$ in merge cost.

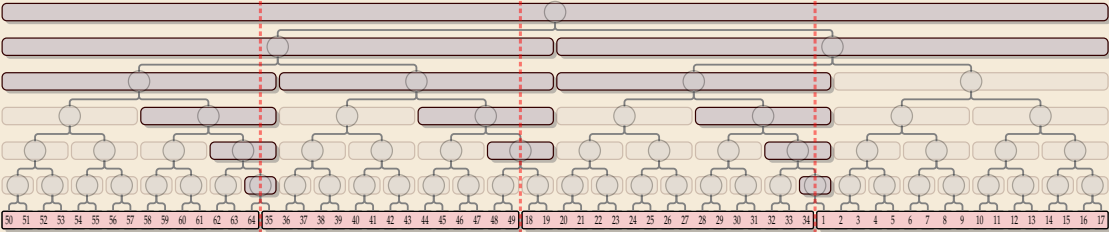
= exactly the cost of *creating* this run in mergesort had it not already existed

↪ best savings we can hope for!

↪ Are overall merge costs $\mathcal{H}(\ell_1, \dots, \ell_r) := \underbrace{n \lg(n)}_{\text{mergesort}} - \underbrace{\sum_{i=1}^r \ell_i \lg(\ell_i)}_{\text{savings from runs}} ?$

$\ell_i = \text{length of } i\text{th run}$

Unfortunately, not quite:



Merge costs:



384 Standard Mergesort



216 with sorted check



127.8 $\mathcal{H}(15, 15, 17, 17)$

Natural Bottom-Up Mergesort

- Can we do better by explicitly detecting runs?

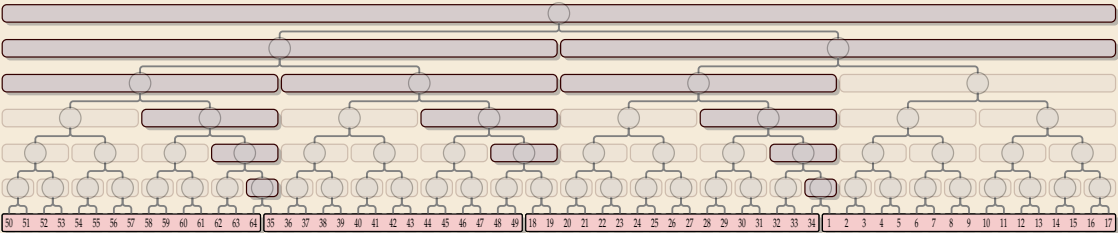
```
1 procedure bottomUpMergesort( $A[0..n]$ )
2    $Q := \text{new Queue}$  // runs to merge
3   // Phase 1: Enqueue singleton runs
4   for  $i = 0, \dots, n - 1$  do
5      $Q.\text{enqueue}((i, i))$ 
6   // Phase 2: Merge runs level-wise
7   while  $\neg Q.\text{isEmpty}()$ 
8      $Q' := \text{new Queue}$ 
9     while  $Q.\text{size}() \geq 2$ 
10       $(i_1, j_1) := Q.\text{dequeue}()$ 
11       $(i_2, j_2) := Q.\text{dequeue}()$ 
12       $\text{merge}(A[i_1..j_1], A[i_2..j_2], \text{buf})$ 
13      copy buf to  $A[i_1..j_2]$ 
14       $Q'.\text{enqueue}((i_1, j_2))$ 
15   if  $\neg Q.\text{isEmpty}()$ 
16      $Q'.\text{enqueue}(Q.\text{dequeue}())$ 
17    $Q := Q'$ 
```

```
1 procedure naturalMergesort( $A[0..n]$ )
2    $Q := \text{new Queue}; i := 0$ 
3   for  $i < n$  do
4      $j := i$ 
5     while  $A[j + 1] \geq A[j]$  do  $j := j + 1$ 
6      $Q.\text{enqueue}((i, j)); i := j$ 
7   while  $\neg Q.\text{isEmpty}()$ 
8      $Q' := \text{new Queue}$ 
9     while  $Q.\text{size}() \geq 2$ 
10       $(i_1, j_1) := Q.\text{dequeue}()$ 
11       $(i_2, j_2) := Q.\text{dequeue}()$ 
12       $\text{merge}(A[i_1..j_1], A[i_2..j_2], \text{buf})$ 
13      copy buf to  $A[i_1..j_2]$ 
14       $Q'.\text{enqueue}((i_1, j_2))$ 
15   if  $\neg Q.\text{isEmpty}()$ 
16      $Q'.\text{enqueue}(Q.\text{dequeue}())$ 
17    $Q := Q'$ 
```

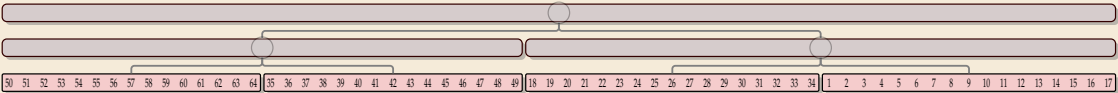
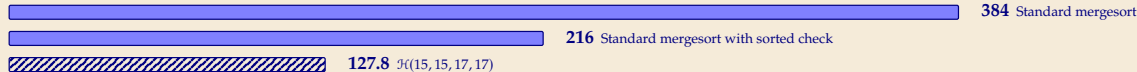
find run (i, j)
starting at i

Natural Bottom-Up Mergesort – Analysis

- Works well runs of roughly equal size, regardless of alignment ...



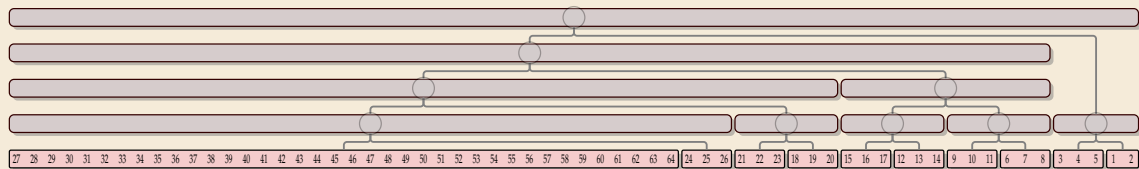
Merge costs:



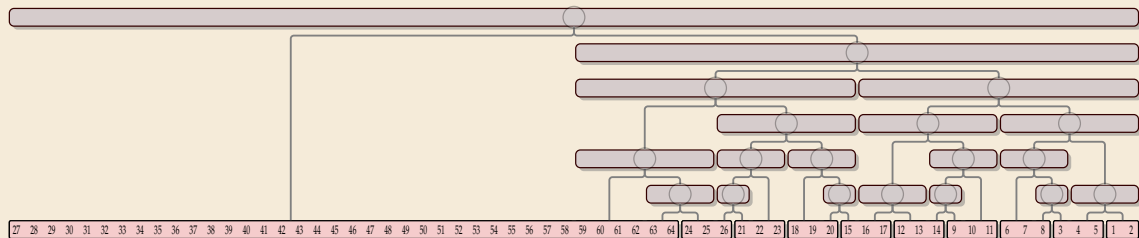
128 Natural bottom-up mergesort

Natural Bottom-Up Mergesort – Analysis [2]

► ... but less so for uneven run lengths



246 Natural bottom-up mergesort



196 Standard mergesort with sorted check

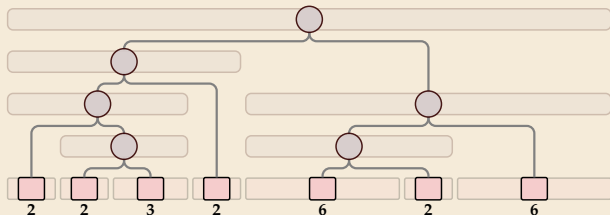
... can't we have both at the same time?!


Good merge orders

◀ Let's take a step back and breathe.

► Conceptually, there are two tasks:

1. Detect and use existing runs in the input $\rightsquigarrow \ell_1, \dots, \ell_r$ (easy) ✓
2. Determine a favorable **order of merges of runs** ("automatic" in top-down mergesort)



Merge cost = total area of 
= total length of paths to all array entries
= $\sum_{w \text{ leaf}} \text{weight}(w) \cdot \text{depth}(w)$

well-understood problem
with known algorithms

\rightsquigarrow

optimal merge tree
= optimal **binary search tree**
for leaf weights ℓ_1, \dots, ℓ_r
(optimal expected search cost)

Nearly-Optimal Mergesort

Nearly-Optimal Mergesorts: Fast, Practical Sorting Methods That Optimally Adapt to Existing Runs

J. Ian Munro

University of Waterloo, Canada
munro@uwaterloo.ca
<https://orcid.org/0000-0002-7165-7988>

Sebastian Wild

University of Waterloo, Canada
wild@uwaterloo.ca
<https://orcid.org/0000-0002-6061-9177>

Abstract

We present two stable mergesort variants, “peeksort” and “powersort”, that exploit existing runs and find nearly-optimal merging orders with negligible overhead. Previous methods either require substantial effort for determining the merging order (Takaoka 2009; Barley & Navarro 2013) or do not have an optimal worst-case guarantee (Peters 2002; Auger, Nicam & Protopas 2013; Buss & Kneip 2018). We demonstrate that our methods are competitive in terms of running time with state-of-the-art implementations of stable sorting methods.

2012 ACM Subject Classification Theory of computation → Sorting and searching

Keywords and phrases adaptive sorting, nearly-optimal binary search trees, Timsort

Digital Object Identifier 10.4230/LIPICs.ESA.2018.63

Related Version arXiv: 1805.04154 (extended version with appendices)

Supplement Material zenodo: 1241162 (code to reproduce running time study)

Funding This work was supported by the Natural Sciences and Engineering Research Council of Canada and the Canada Research Chairs Programme.

1 Introduction

Sorting is a fundamental building block for numerous tasks and ubiquitous in both the theory and practice of computing. While practical and theoretically (close-to) optimal comparison-based sorting methods are known, *instance-optimal sorting*, i.e., methods that adapt to the actual input and exploit specific structural properties if present, is still an area of active research. We survey some recent developments in Section 1.1.

Many different structural properties have been investigated in theory. Two of them have also found wide adoption in practice, e.g., in Oracle’s Java runtime library: adapting to the presence of duplicate keys and using existing sorted segments, called *runs*. The former is achieved by a so-called *last-given* partitioning variant of quicksort [8], which is also used in the OpenBSD implementation of *qsort* from the C standard library. It is an *unstable* sorting method, though, i.e., the relative order of elements with equal keys might be destroyed in the process. It is hence used in Java solely for primitive-type arrays.

© J. Ian Munro and Sebastian Wild.
Reprinted under Creative Commons License CC-BY
2018 Annual European Symposium on Algorithms (ESA 2018).
Editors: Yossi Azar, Michael Baur, and Gregorio Alonso, Article No. 63, pp. 63:1–63:13
LIPICs International Proceedings in Informatics
LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

- ▶ In 2018, with Ian Munro, I combined research on nearly-optimal BSTs with mergesort

⇒ 2 new algorithms: *Peeksort* and *Powersort*

- ▶ both adapt provably optimally to existing runs even in worst case:
 $\text{mergcost} \leq \mathcal{H}(\ell_1, \dots, \ell_r) + 2n$
- ▶ both are lightweight extensions of existing methods with negligible overhead
- ▶ both fast in practice

Peeksort

- ▶ based on top-down mergesort

- ▶ “peek” at middle of array & find closest run boundary

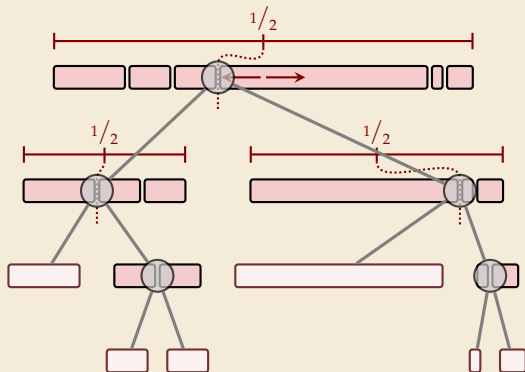


- ↪ split there and recurse
(instead of at midpoint)

- ▶ can avoid scanning runs repeatedly:
 - ▶ find full run straddling midpoint
 - ▶ remember length of known runs at boundaries



- ↪ with clever recursion, scan each run only once.



Peeksort – Code

```
1 procedure peeksort( $A[\ell..r]$ ,  $\Delta_\ell$ ,  $\Delta_r$ )
2   if  $r - \ell \leq 1$  then return
3   if  $\ell + \Delta_\ell == r \vee \ell == r + \Delta_r$  then return
4    $m := \ell + \lfloor (r - \ell)/2 \rfloor$ 
5    $i := \begin{cases} \ell + \Delta_\ell & \text{if } \ell + \Delta_\ell \geq m \\ \text{extendRunLeft}(A, m) & \text{else} \end{cases}$ 
6    $j := \begin{cases} r + \Delta_r \leq m & \text{if } r + \Delta_r \leq m \leq m \\ \text{extendRunRight}(A, m) & \text{else} \end{cases}$ 
7    $g := \begin{cases} i & \text{if } m - i < j - m \\ j & \text{else} \end{cases}$ 
8    $\Delta_g := \begin{cases} j - i & \text{if } m - i < j - m \\ i - j & \text{else} \end{cases}$ 
9   peeksort( $A[\ell..g]$ ,  $\Delta_\ell$ ,  $\Delta_g$ )
10  peeksort( $A[g..r]$ ,  $\Delta_g$ ,  $\Delta_r$ )
11  merge( $A[\ell, g]$ ,  $A[g..r]$ , buf)
12  copy buf to  $A[\ell..r]$ 
```

► Parameters:



► initial call:

peeksort($A[0..n]$, Δ_0 , Δ_n) with
 $\Delta_0 = \text{extendRunRight}(A, 0)$
 $\Delta_n = n - \text{extendRunLeft}(A, n)$

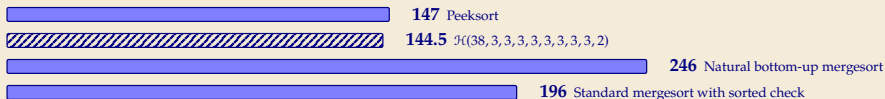
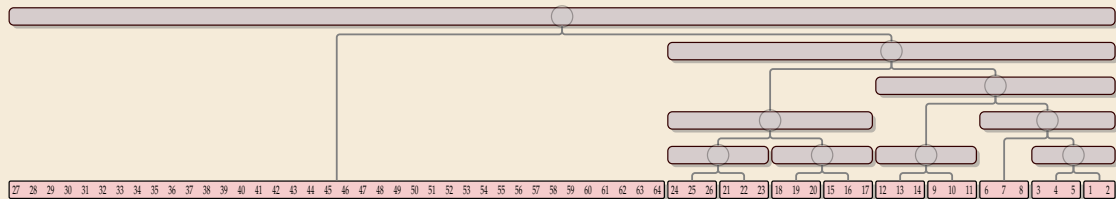
► helper procedure

```
1 procedure extendRunRight( $A[0..n]$ ,  $i$ )
2    $j := i + 1$ 
3   while  $j < n \wedge A[j - 1] \leq A[j]$ 
4      $j := j + 1$ 
5   return  $j$ 
```

(extendRunLeft similar)

Peeksort – Analysis

- Consider tricky input from before again:



- One can prove: Mergecost always $\leq \mathcal{H}(\ell_1, \dots, \ell_r) + 2n$

~> We can have the best of both worlds!

3.6 Python's list sort

Sorting in Python

► CPython

- *Python* is only a specification of a programming language
- The Python Foundation maintains *CPython* as the official reference implementation of the Python programming language
- If you don't specifically install something else, python will be CPython

► part of Python are `list.sort` resp. `sorted` built-in functions

- implemented in C
- use *Timsort*, custom Mergesort variant by Tim Peters



Sept 2021: **Python uses Powersort!**
in CPython 3.11 and PyPy 7.3.6

msg400864 -
(view)

Author: Tim Peters (tim.peters) * 🌐

Date:
2021-09-01
19:43

I created a PR that implements the powersort merge strategy:

<https://github.com/python/cpython/pull/28108>

Across all the time this issue report has been open, that strategy continues to be the top contender. Enough already ;-) It's indeed a more difficult change to make to the code, but that's in relative terms. In absolute terms, it's not at all a hard change.

Laurent, if you find that some variant of ShiversSort actually runs faster than that, let us know here! I'm a big fan of Vincent's innovations too, but powersort seems to do somewhat better "on average" than even his length-adaptive ShiversSort (and implementing that too would require changing code outside of `merge_collapse()`).

Timsort (original version)

```

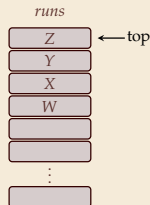
1 procedure Timsort( $A[0..n]$ )
2    $i := 0$ ;  $runs := \text{new Stack}()$ 
3   while  $i < n$ 
4      $j := \text{ExtendRunRight}(A, i)$ 
5      $runs.push(i, j)$ ;  $i := j$ 
6     while rule A/B/C/D applicable
7       merge corresponding runs
8   while  $runs.size() \geq 2$ 
9     merge topmost 2 runs
  
```

- ▶ above shows the core algorithm;
many more algorithm engineering tricks

Advantages:

- ▶ profits from existing runs
- ▶ *locality of reference* for merges

- ▶ **But: not** optimally adaptive! (next slide)
Reason: Rules A–D (Why exactly these?!)



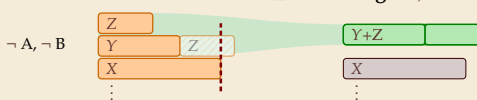
Rule A: $Z > X \rightsquigarrow \text{merge}(X, Y)$



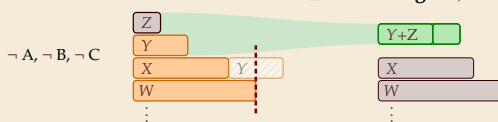
Rule B: $Z \geq Y \rightsquigarrow \text{merge}(Y, Z)$



Rule C: $Y + Z \geq X \rightsquigarrow \text{merge}(Y, Z)$

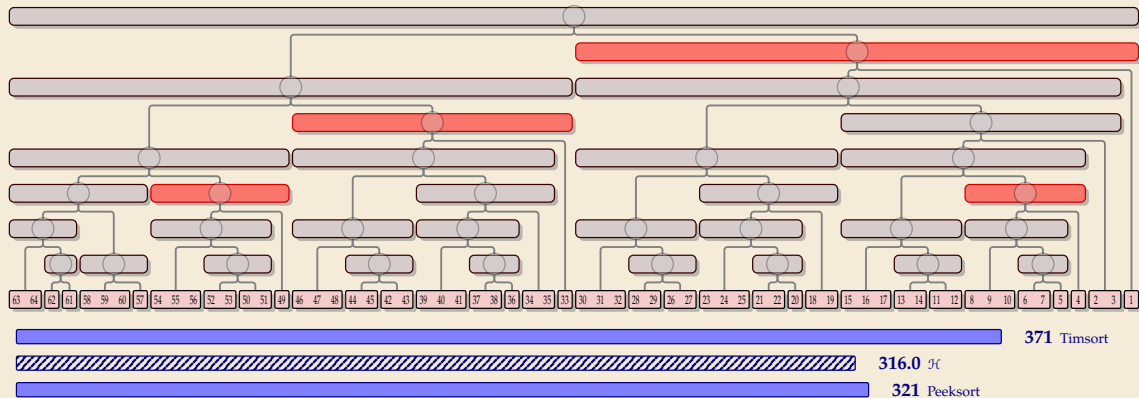


Rule D: $X + Y \geq W \rightsquigarrow \text{merge}(Y, Z)$



Timsort bad case

- On certain inputs, Timsort's merge rules don't work well:

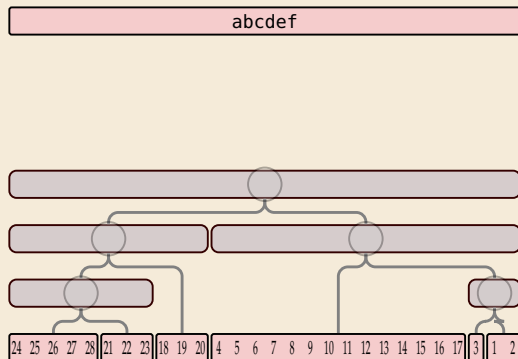


- As n increases, Timsort's cost approach $1.5 \cdot \mathcal{H}$, i. e., 50% more merge costs than necessary
 - intuitive problem: regularly very unbalanced merges

Powersort

↪ Timsort's *merge rules* aren't great, but overall algorithm has appeal . . . can we keep that?


```
1 procedure Powersort( $A[0..n]$ )
2    $i := 0$ ;  $runs := \text{new Stack}()$ 
3    $j := \text{ExtendRunRight}(A, i)$ 
4    $runs.push(i, j)$ ;  $i := j$ 
5   while  $i < n$ 
6      $j := \text{ExtendRunRight}(A, i)$ 
7      $p := \text{power}(runs.top(), (i, j), n)$ 
8     while  $p \leq \text{topmost power}$ 
9       merge topmost 2 runs
10     $runs.push(i, j)$ ;  $i := j$ 
11  while  $runs.size() \geq 2$ 
12    merge topmost 2 runs
```



Powersort – Computing powers

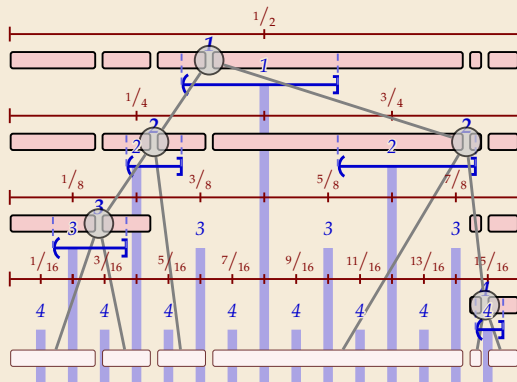
► Computing the power of (the node between) two runs $A[i_1..j_1]$ and $A[i_2..j_2]$

► $\text{⌈} \text{---} \text{⌋}$ = normalized midpoint interval

► power = min ℓ s.t. $\text{⌈} \text{---} \text{⌋}$
contains $c \cdot 2^{-\ell}$ 

```

1 procedure power( $((i_1, j_1), (i_2, j_2), n)$ )
2    $n_1 := j_1 - i_1 + 1$ 
3    $n_2 := j_2 - i_2 + 1$ 
4    $a := \frac{i_1 + \frac{1}{2}n_1 - 1}{n}$ 
5    $b := \frac{i_2 + \frac{1}{2}n_2 - 1}{n}$  // interval  $(a, b]$ 
6    $\ell := 0$ 
7   while  $\lfloor a \cdot 2^\ell \rfloor \neq \lfloor b \cdot 2^\ell \rfloor$ 
8      $\ell := \ell + 1$ 
9   return  $\ell$ 
  
```



Powersort – Discussion



Retains all advantages of Timsort

- ▶ good locality in memory accesses
- ▶ no recursion
- ▶ all the tricks in Timsort



optimally adapts to existing runs



minimal overhead for finding merge order

Part III

Sorting with of many processors

3.7 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 ← *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
 \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.

↪ span and work give guideline for *any* number of processors

3.8 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 $A[0..n)$ 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
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

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!
⚡ data dependencies!

↪ need a different approach

Let's try a simpler problem first.

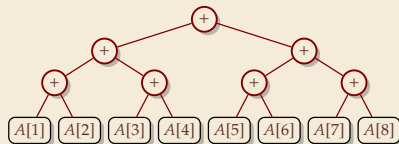
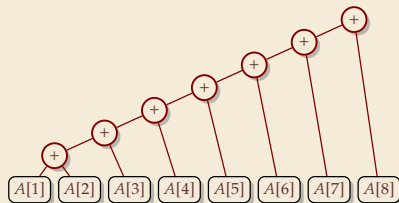
Excursion: Sum

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

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

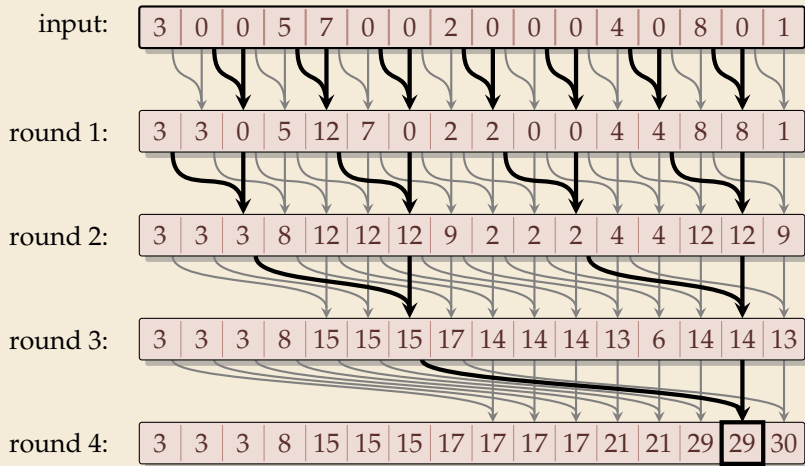
↪ Height of tree = parallel time!

```
1 procedure prefixSum( $A[0..n)$ )  
2   for  $i := 1, \dots, n - 1$  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( $A[0..n]$ )
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

~> $\Theta(\log n)$ time best possible!

► Work:

- $\geq \frac{n}{2}$ additions in all rounds (except maybe last round)

~> $\Theta(n \log n)$ 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!

~> 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

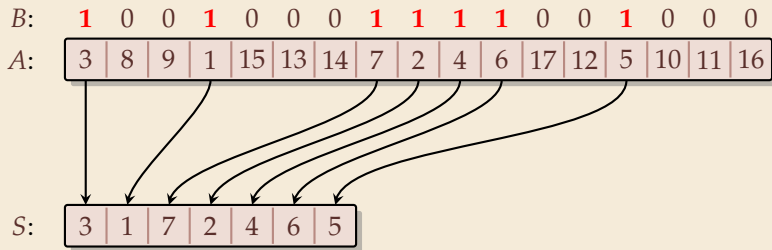
► **Work:**

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

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 C := B // copy B
2 parallelPrefixSums(C)
3 for j := 0, ..., n - 1 do in parallel
4     if B[j] == 1 then S[C[j] - 1] := A[j]
5 end parallel for
```

3.9 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 { parQuicksort( $A[l..j]$ ), parQuicksort( $A[j + 1..r]$ ) }
5
6 procedure parallelPartition( $A[0..n]$ ,  $b$ )
7   swap( $A[n - 1]$ ,  $A[b]$ );  $p := A[n - 1]$ 
8   for  $i = 0, \dots, n - 2$  do in parallel
9      $S[i] := \lfloor A[i] \leq p \rfloor$  //  $S[i]$  is 1 or 0
10     $L[i] := 1 - S[i]$ 
11  end parallel for
12  in parallel { parallelPrefixSum( $S[0..n - 2]$ ); 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 $A[l..m)$ and $A[m..r)$?
 - ▶ Must treat elements independently.
 - ▶ correct position of x in sorted output = $\overset{\text{\#elements} \leq x}{\text{rank}}$ of x breaking ties by position in A
 - ▶ $\# \text{ elements} \leq x = \# \text{ elements from } A[l..m) \text{ that are } \leq x + \# \text{ elements from } A[m..r) \text{ that are } \leq x$
 - ▶ 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 ~> $\Theta(n \log n)$

~> 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[])`