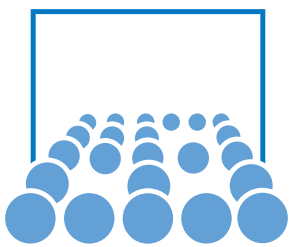
Fundamental Algorithms

Chapter 3: More Sorting

Michael Bader

Winter 2011/12



Are Mergesort and Quicksort optimal?

Definition

Comparison sorts are sorting algorithms that use only comparisons (i.e. tests as $\leq, =, >, \ldots$) to determine the relative order of the elements.

Examples:

- InsertSort, BubbleSort
- MergeSort, (Randomised) Quicksort

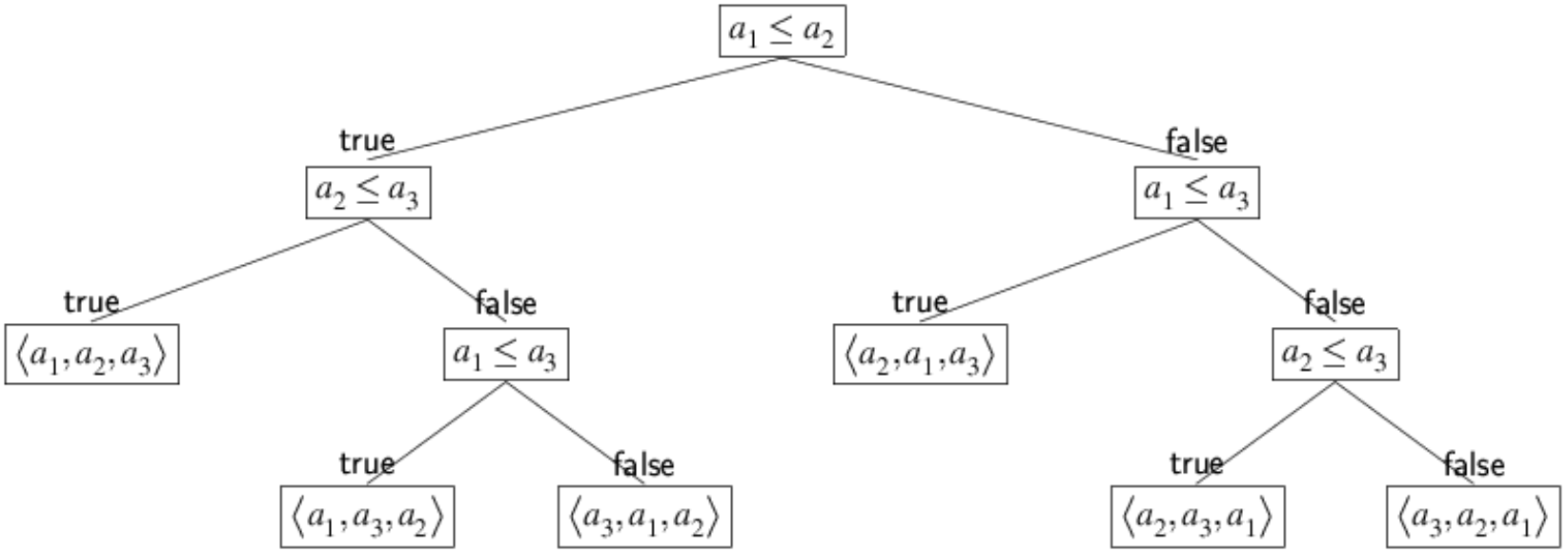
Question:

Is $T(n) \in \Theta(n \log n)$ the best we can get (in the worst/average case)?

Decision Trees

Definition

A **decision tree** is a binary tree in which each internal node is annotated by a comparison of two elements.
The leaves of the decision tree are annotated by the respective permutations that will put an input sequence into sorted order.



Decision Trees – Properties

Each comparison sort can be represented by a decision tree:

- a path through the tree represents a sequence of comparisons
- sequence of comparisons depends on results of comparisons
- can be pretty complicated for Mergesort, Quicksort, . . .

A decision tree can be used as a comparison sort:

- if every possible permutation is annotated to at least one leaf of the tree!
- if (as a result) the decision tree has at least n! (distinct) leaves.

A Lower Bound for the Complexity of Comparison Sorts

- A binary tree of height h (h the length of the longest path) has at most 2^h leaves.
- To sort n elements, the decision tree needs $n!$ leaves.

Theorem

Any decision tree that sorts n elements has height $\Omega(n \log n)$.

Proof:

- h comparisons in the worst case are equivalent to a decision tree of height h
- with h comparisons, we can sort n elements (at best), if

$$n! \leq 2^h \quad \Leftrightarrow \quad h \geq \log(n!) \in \Omega(n \log n)$$

- because:

$$h \geq log(n!) \geq \log\left(n^{n/2}\right) = \frac{n}{2} \log n$$

Optimality of Mergesort and Quicksort

Corollaries:

- MergeSort is an optimal comparison sort in the worst/average case
- QuickSort is an optimal comparison sort in the average case

Consequences and Alternatives:

- comparison sorts can be faster than MergeSort, but only by a constant factor
- comparison sorts can not be asymptotically faster
- sorting algorithms might be faster, if they can exploit additional information on the size of elements
- examples: **BucketSort**, CountingSort, RadixSort

Bucket Sort

Basic Ideas and Assumptions:

- pre-sort numbers in buckets that contain all numbers within a certain interval
- hope (assume) that input elements are evenly distributed and thus uniformly distributed to buckets
- sort buckets and concatenate them

Requires "Buckets":

- can hold arbitrary numbers of elements
- can insert elements efficiently: in $O(1)$ time
- can concatenate buckets efficiently: in $O(1)$ time
- remark: linked lists will do

Implementation of BucketSort

```
BucketSort (A:Array[1..n]) {

    Create Array B[0..n-1] of Buckets;
    //assume all Buckets B[i] are empty at first

    for i from 1 to n do {
        insert A[i] into Bucket B[floor(n * A[i])];
    }

    for i from 0 to n-1 do {
        sort Bucket B[i] ;
    }

    concatenate Buckets B[0], B[1], ..., B[n-1] into A
}
```

Number of Operations of BucketSort

Operations:

- n operations to distribute n elements to buckets
- plus effort to sort all buckets

Best Case:

- if each bucket gets 1 element, then $\Theta(n)$ operations are required

Worst Case:

- if one bucket gets all elements, then $T(n)$ is determined by the sorting algorithm for the buckets

Bucketsort – Average Case Analysis

- probability that bucket i contains k elements:

$$P(n_i = k) = \binom{n}{k} \left(\frac{1}{n}\right)^k \left(1 - \frac{1}{n}\right)^{n-k}$$

- expected mean and variance for such a distribution:

$$E[n_i] = n \cdot \frac{1}{n} = 1 \qquad \text{Var}[n_i] = n \cdot \frac{1}{n}\left(1 - \frac{1}{n}\right) = \left(1 - \frac{1}{n}\right)$$

- InsertionSort for buckets $\Rightarrow \leq cn^2 \in O(n_i^2)$ operations per bucket
- expected operations to sort one bucket:

$$\bar{T}(n_i) \leq \sum_{k=0}^{n-1} P(n_i = k) \cdot ck^2 = cE[n_i^2]$$

Bucketsort – Average Case Analysis (2)

- theorem from statistics:

$$E[X^2] = E[X]^2 + \mathrm{Var}(X)$$

- expected operations to sort one bucket:

$$\bar{T}(n_i) \leq cE[n_i^2] = c\left(E[n_i]^2 + \mathrm{Var}[n_i]\right) = c\left(1^2 + 1 - \frac{1}{n}\right) \in \Theta(1)$$

- expected operations to sort all buckets:

$$\bar{T}(n) = \sum_{i=0}^{n-1} \bar{T}(n_i) \leq c \sum_{i=0}^{n-1} \left(2 - \frac{1}{n}\right) \in \Theta(n)$$

(note: expected value of the sum is the sum of expected values)

Part II

Sorting in Parallel

A (naive?) Example: AccumulateSort

```
AccumulateSort (A:Array[1..n]) {

    Create Array P[1..n] of Integer;
    // all P[i]=0 at start

    for 1 <= i,j <= n and i<j do in parallel {
        if A[i] > A[j]
        then P[i] := P[i]+1
        else P[j] := P[j]+1;
    }

    for i from 1 to n do in parallel {
        A[ P[i] ] := A[i];
    }
}
```

AccumulateSort – Discussion

Idea:

- do all $\binom{n}{2}$ comparisons at once and in parallel
- use $\binom{n}{2}$ processors
- count “wins” for each element to obtain its position
- complexity: $T_{AS} = \Theta(1)$ on $n(n-1)/2$ processors

Assumptions:

- all read accesses to A can be done in parallel
- increments of P[i] and P[j] can be done in parallel
- second for-loop is executed after the first one (on all processors)
- all moves A[P[i]] := A[i] happen in one atomic step
  (no overwrites due to sequential execution)

Towards Parallel Algorithms

A First Set of Problems and Questions:

- parallel read access to variables possible?
- parallel write access (or increments?) to variables possible?
- are parallel/global copy statements realistic?
- how do we synchronise parallel executions?

Reality vs. Theory:

- on real hardware: probably lots of restrictions
  (e.g., no parallel reads/writes; no global operations on or access to memory)
- in theory: if there were no such restrictions, how far can we get?
- or: for different kinds of restrictions, how far can we get?

MergeSort in Parallel

```
MergeSortPar(A:Array[1..n]) {
 if n > 1 then {
    m := floor(n/2);

    do in parallel {
      create array L[1...m];
      for i from 1 to m do { L[i] := A[i]; }
      MergeSort(L);
    |
      create array R[1...n-m];
      for i from 1 to n-m do { R[i] := A[m+i]; }
      MergeSort(R);
    };

    Merge(L,R,A);
 }
}
```

Parallel MergeSort

Idea:

- parallelise “divide-and-conquer”:
  recursive calls can be done in parallel
- use $p/2$ processors for each of the recursive calls
  (if p processors are available)

Merging in Parallel?

- can Merge be executed in parallel?
- by how many processors?

Can Merge be Parallelised?

```
Merge (L:Array[1..p], R:Array[1..q], A:Array[1..n]) {
// merge the sorted arrays L and R into A (sorted)
// we presume that n=p+q
   i:=1; j:=1:
   for k from 1 to n do {
      if i > p
         then { A[k]:=R[j]; j=j+1; }
      else if j > q
         then { A[k]:=L[i]; i:=i+1; }
      else if L[i] < R[j]
         then { A[k]:=L[i]; i:=i+1; }
         else { A[k]:=R[j]; j:=j+1; }
   }
}
```

Problem: inherently sequential progress through arrays A, L, R

Odd-Even Merge

Ideas:

- start with a two sorted lists of length $n/2$:

2	3	4	7	1	5	6	8

- consider elements with odd and even index:

2	3	4	7	1	5	6	8

- sort odd- and even-indexed elements separately:

1	3	2	5	4	7	6	8

Observations

- final sequence is nearly sorted (only pairwise exchange required)
- odd- and even-indexed elements can be processed in parallel

Correctness of the Final Exchange Step

Claim (after odd/even sort):

- exchanges of a_{2i} and a_{2i+1} are sufficient for sorting

Counting Argument: x an odd-indexed element: $x = a_{2i+1}$

- exactly i odd-indexed elements are smaller than x (sorted lists)
- $d_l, d_r =$ number of odd-indexed elements $< x$ in left/right half
  $\Rightarrow i = d_l + d_r$
- $v_l, v_r =$ number of even-indexed elements $< x$ in left/right half
- x in left half: $v_l = d_l$, $v_r \in \{d_r, d_r - 1\}$
- x in right half: $v_l \in \{d_l, d_l - 1\}$, $v_r = d_r$
- **consequence:** $v_l + v_r \in \{d_l + d_r, d_l + d_r - 1\} = \{i, i - 1\}$

Correctness of the Final Exchange Step (2)

Counting Argument:

- count even- and odd-indexed elements $< x$ in both halves
- $v_l + v_r \in \{d_l + d_r, d_l + d_r - 1\} = \{i, i-1\}$

Possible Scenarios:

- $v_l + v_r = i \Rightarrow$ exactly i even elements $< x$
  $\Rightarrow i$-th even-indexed element $a_{2i} < x \rightarrow$ **OK**
- $v_l + v_r = i - 1 \Rightarrow$ exactly $i - 1$ even elements $< x$
  therefore: $a_{2(i-1)} < x$, but $a_{2i} > x \rightarrow$ **exchange**
- in both cases:
  $a_{2(i+1)} > x$ (at most i even elements $< x$) $\rightarrow$ **OK**
  $a_{2(i-1)} < x$ (at least $i - 1$ even elements $< x$) $\rightarrow$ **OK**

$\Rightarrow$ **only the left even-indexed neighbour of x can be out of place**

OddEvenMerge – A First Try

```
OddEvenMerge_1 (A:Array[1..n]) {
// merge the sorted arrays A[1..n/2] and A[n/2+1..n]
// into A (sorted); n is a power of 2

    OddEvenSplit(A,Odd,Even);

    Sort(Odd); Sort(Even);

    OddEvenJoin(A,Odd,Even);

    for i from 1 to n/2-1 do {
        if A[2i] > A[2i+1]
        then exchange A[2i] and A[2i+1]
    }
}
```

OddEvenSplit and OddEvenJoin (in parallel!)

```
OddEvenSplit (A:Array[1..n],
              Odd:Array[1..n/2], Even:Array[1..n/2]) {
    for i from 1 to n/2 do in parallel {
        Odd[i]  := A[2i-1];
        Even[i] := A[2i];
    }
}

OddEvenJoin (A:Array[1..n],
             Odd:Array[1..n/2], Even:Array[1..n/2]) {
    for i from 1 to n/2 do in parallel {
        A[2i-1] := Odd[i]  ;
        A[2i]   := Even[i];
    }
}
```

Towards a Better Implementation of OddEvenMerge

After OddEvenSplit:

- Odd consists of two halves that are already sorted
- Even consists of two halves that are already sorted

$\Rightarrow$ Odd and Even can be sorted using OddEvenMerge

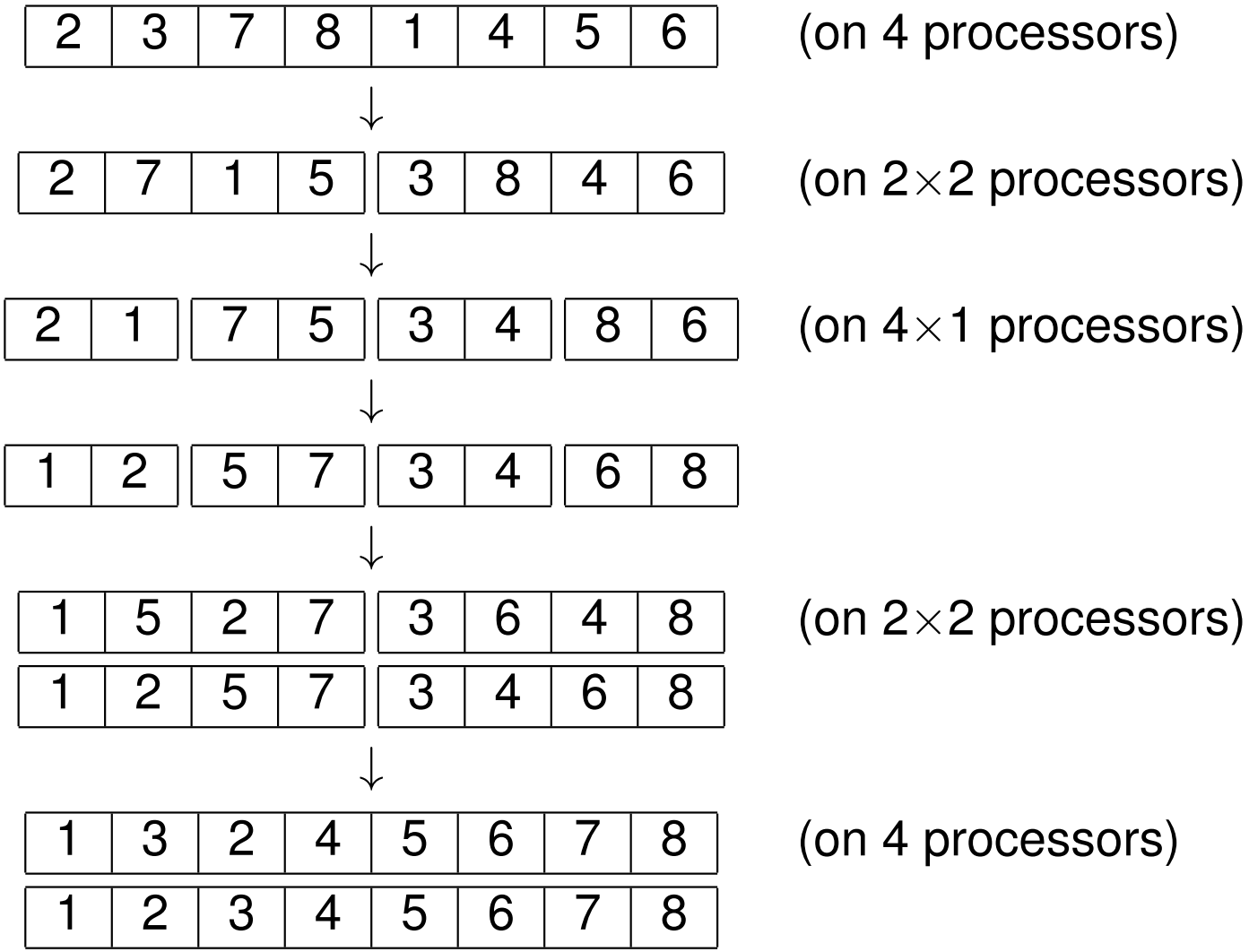
OddEvenMerge in Parallel:

- OddEvenSplit and OddEvenJoin are already parallel
- calls to OddEvenMerge can be executed in parallel (recursive calls will again issues parallel calls)
- final exchange loop can be parallelised

Parallel OddEvenMerge

```
OddEvenMerge (A:Array[1..n]) {
    ! add stopping criterion:
    if n<=2 then { SortTwo(A); return; };

    OddEvenSplit(A,Odd,Even);

    do in parallel{ OddEvenMerge(Odd);
                    OddEvenMerge(Even); }

    OddEvenJoin(A,Odd,Even);

    for i from 1 to n/2-1 do in parallel {
        if A[2i] > A[2i+1]
        then exchange A[2i] and A[2i+1]
    }
}
```

Parallelism in OddEvenMerge



OddEvenMergeSort (in Parallel)

```
OddEvenMergeSort(A:Array[1..n]) {
  if n >= 2 then {

    do in parallel {
      OddEvenMergeSort(A[1..n/2]);
    |
      OddEvenMergeSort(A[n/2+1..n]);
    };

    OddEvenMerge(A);
  }
}
```

Complexity of Odd-Even MergeSort

Complexity of OddEvenMerge:

- $\Theta(\log n)$ subsequent steps
- each step executed on $\frac{n}{2}$ processors
- total work: $\Theta(n \log n)$

Complexity of Odd-Even MergeSort:

- requires executions of OddEvenMerge on subarrays of lengths $k = 2, 4, \ldots, n$
- each OddEvenMerge step requires $\Theta(\log k)$ steps
- number of subsequent steps:

$$\log 2 + \log 4 + \cdots + \log n = \Theta\left((\log n)^2\right)$$

- total work: $\Theta\left(n(\log n)^2\right)$