Sorting (Chapter 9)

Alexandre David
B2-206

Sorting

Problem

Arrange an unordered collection of elements into monotonically increasing (or decreasing) order.

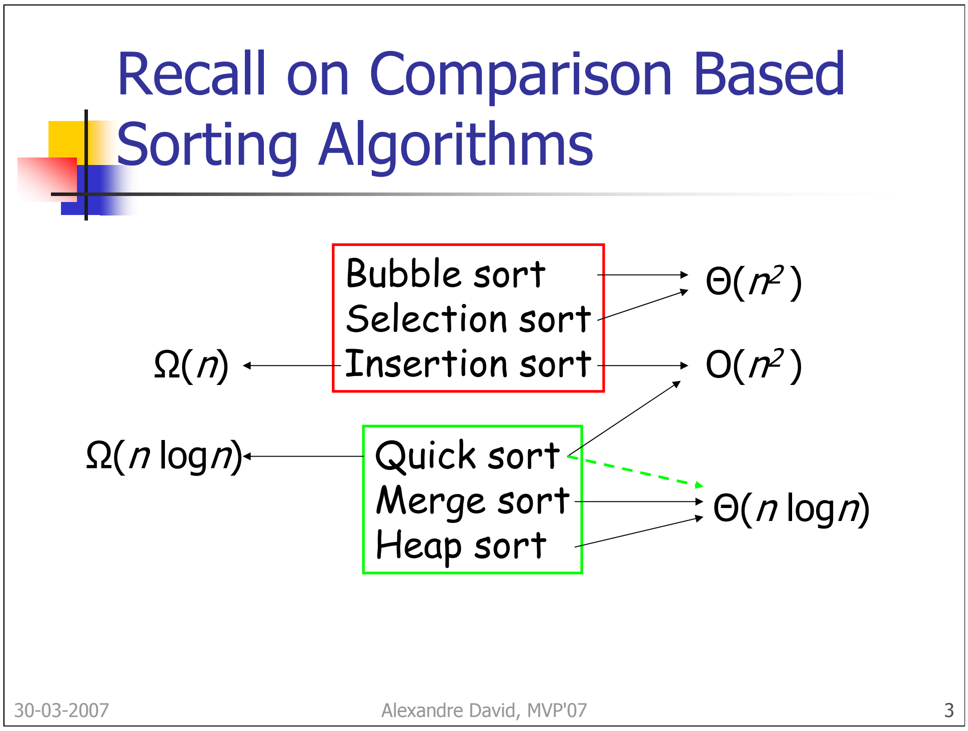
Let $S = \langle a_1, a_2, \ldots, a_n \rangle$.

Sort S into $S' = \langle a_1', a_2', \ldots, a_n' \rangle$ such that

$a_i' \le a_j'$ for $1 \le i \le j \le n$

and S' is a permutation of S.

The elements to sort (actually used for comparisons) are also called the keys.



You should know these complexities from a previous course on algorithms.

Characteristics of Sorting Algorithms

- In-place sorting: No need for additional memory (or only constant size).
- Stable sorting: Ordered elements keep their original relative position.
- Internal sorting: Elements fit in process memory.
- External sorting: Elements are on auxiliary storage.

We assume internal sorting is possible.

Fundamental Distinction

- **Comparison based** sorting:
  - *Compare-exchange* of pairs of elements.
  - Lower bound is $\Omega(n \log n)$ (proof based on decision trees).
  - Merge & heap-sort are optimal.
- **Non-comparison based** sorting:
  - Use information on the element to sort.
  - Lower bound is $\Omega(n)$.
  - Counting & radix-sort are optimal.

We assume comparison based sorting is used.

Issues in Parallel Sorting

- Where to store input & output?
  - One process or distributed?
  - Enumeration of processes used to distribute output.
- How to compare?
  - How many elements per process?
  - As many processes as element $\Rightarrow$ poor performance because of inter-process communication.

Parallel Compare-Exchange

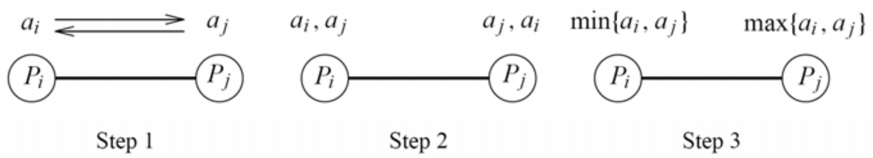
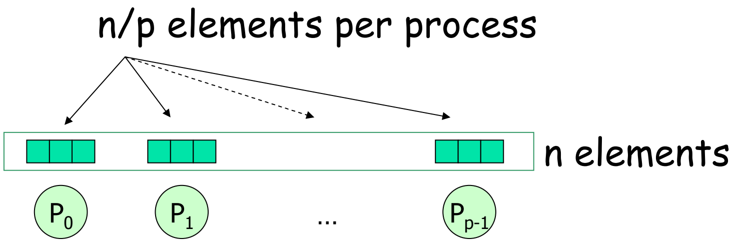


Figure 9.1 A parallel compare-exchange operation. Processes P_i and P_j send their elements to each other. Process P_i keeps $\min\{a_i, a_j\}$, and P_j keeps $\max\{a_i, a_j\}$.

Communication cost: $t_s + t_w$.
Comparison cost much cheaper $\Rightarrow$ communication time dominates.

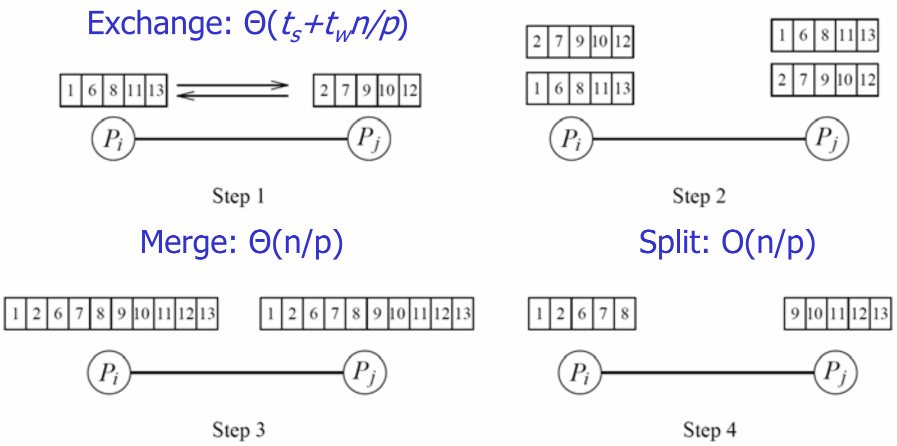
Blocks of Elements Per Process



Blocks: $A_0 \leq A_1 \leq \ldots \leq A_{p-1}$

Compare-Split

For large blocks: $\Theta(n/p)$



Sorting Networks

- Mostly of theoretical interest.
- Key idea: Perform many comparisons in parallel.
- Key elements:
  - Comparators: 2 inputs, 2 outputs.
  - Network architecture: Comparators arranged in columns, each performing a permutation.
  - Speed proportional to the depth.

Comparators

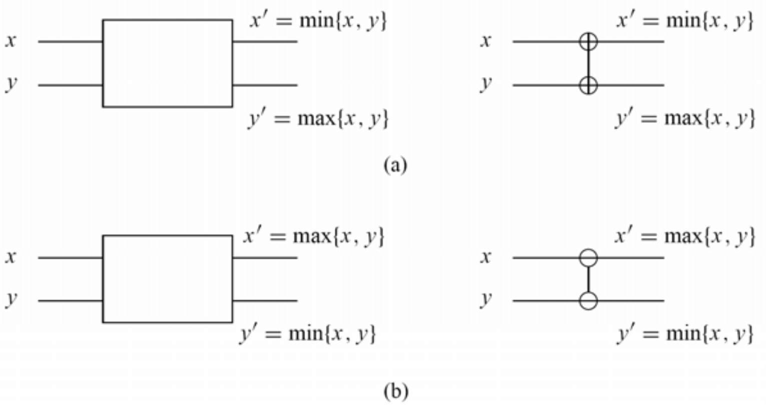


Figure 9.3 A schematic representation of comparators: (a) an increasing comparator, and (b) a decreasing comparator.

Sorting Networks

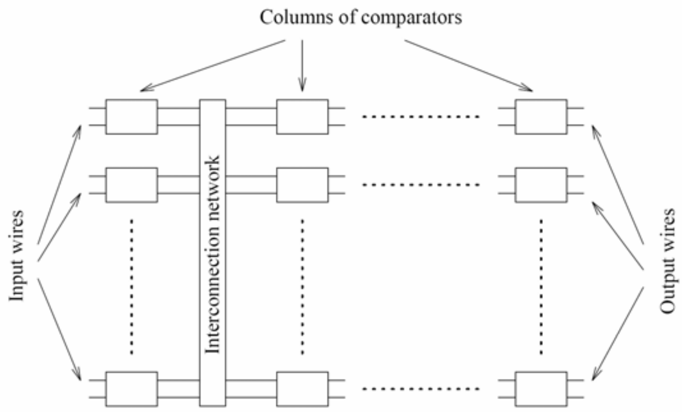
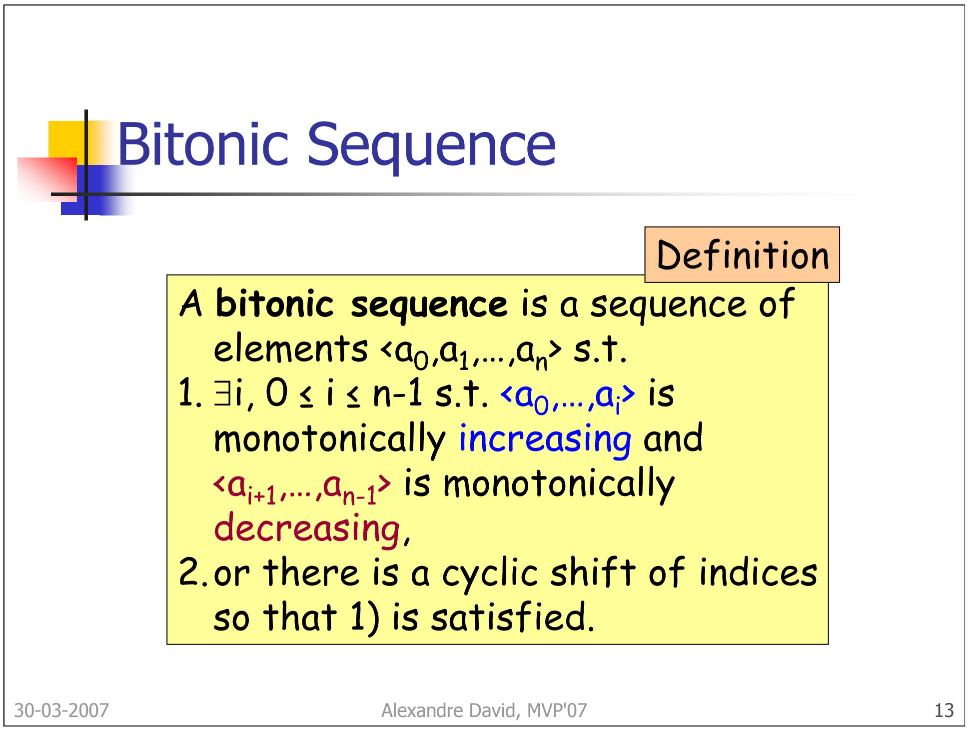


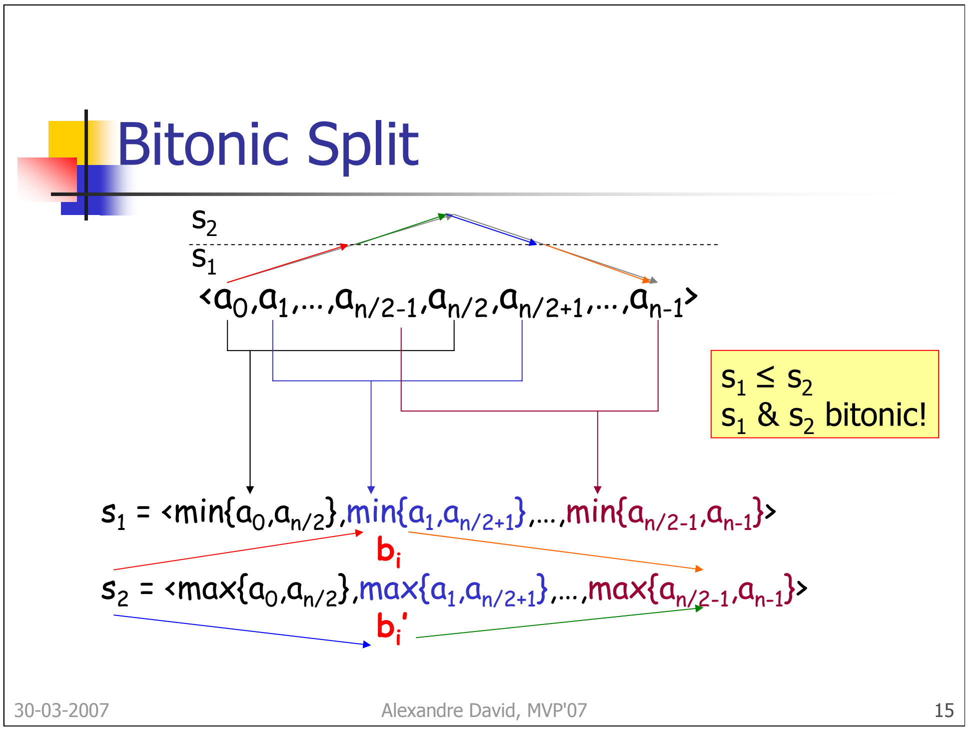
Figure 9.4 A typical sorting network. Every sorting network is made up of a series of columns, and each column contains a number of comparators connected in parallel.



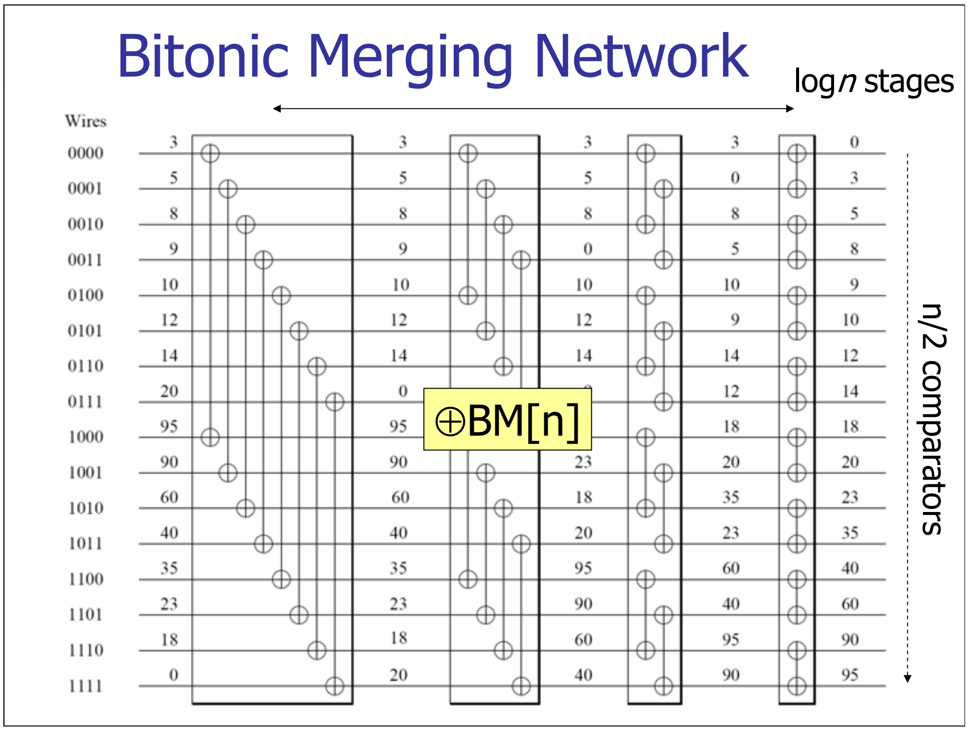
Example: <1,2,4,7,6,0> & <8,9,2,1,0,4> are bitonic sequences.

Bitonic Sort

- Rearrange a bitonic sequence to be sorted.
- Divide & conquer type of algorithm (similar to quicksort) using **bitonic splits**.
  - Sorting a bitonic sequence using bitonic splits = bitonic merge.
  - But we need a bitonic sequence…



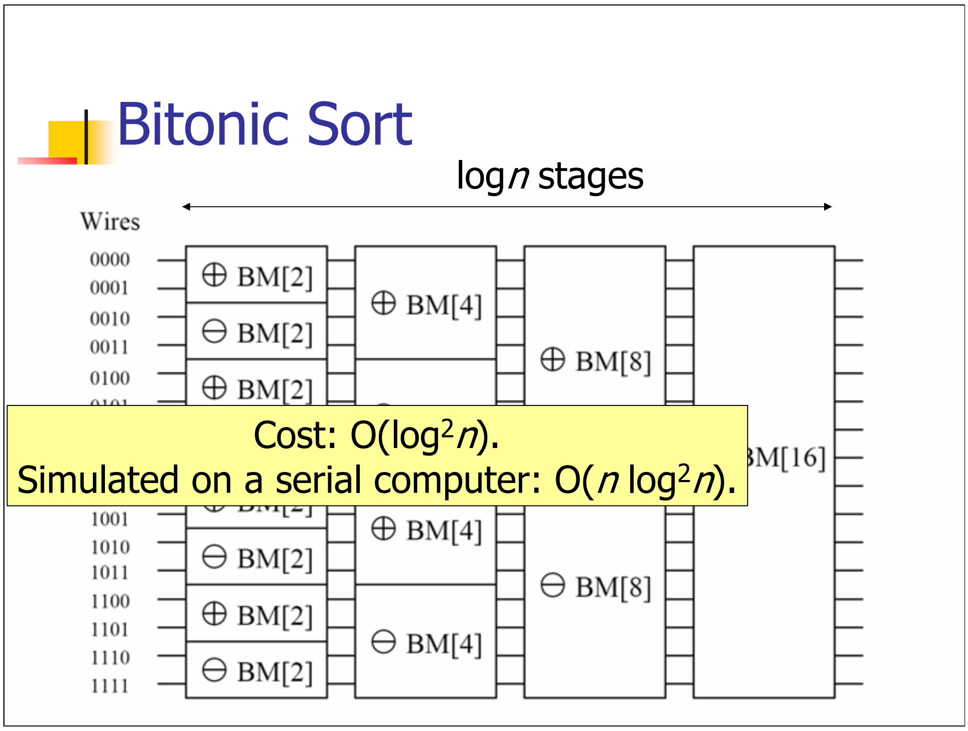
And in fact the procedure works even if the original sequence needs a cyclic shift to look like this particular case.



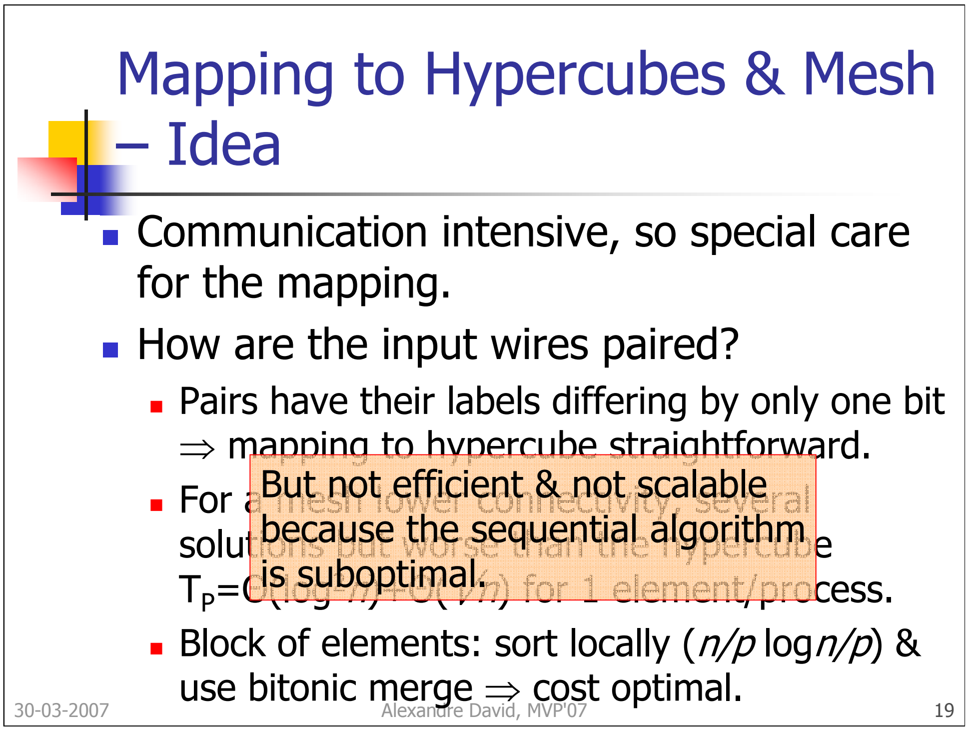
Cost: Θ(log*n*) obviously.

Bitonic Sort

- Use the bitonic network to merge bitonic sequences of increasing length… starting from 2, etc.
- Bitonic network is a component.



Not cost optimal compared to the optimal serial algorithm.



Hypercube: Neighbors differ with each other by one bit.

Bubble Sort

```
procedure BUBBLE_SORT(n)
begin
   for i := n-1 downto 1 do
      for j := 1 to i do
         compare_exchange(a_j,a_j+1);
end
```

$\Theta(n^2)$

- Difficult to parallelize as it is because it is inherently sequential.

It is difficult to sort *n* elements in time log*n* using *n* processes (cost optimal w.r.t. the best serial algorithm in *n* log*n*) but it is easy to parallelize other (less efficient) algorithms.

Odd-Even Transposition Sort

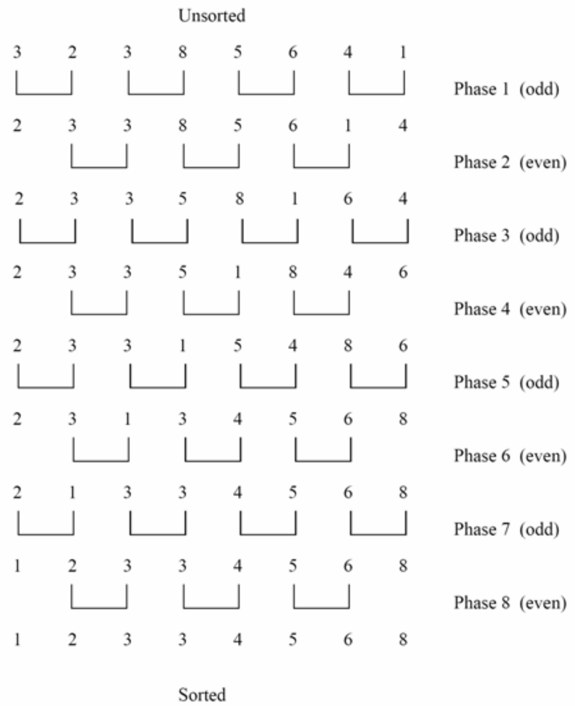
procedure ODD-EVEN(n)
begin
    for $i := 1$ **to** n **do**
    begin
        if i is odd **then**
            for $j := 0$ **to** $n/2 - 1$ **do**
                compare-exchange(a_{2j+1}, a_{2j+2});
        if i is even **then**
            for $j := 1$ **to** $n/2 - 1$ **do**
                compare-exchange(a_{2j}, a_{2j+1});
    end for
end ODD-EVEN

$\Theta(n^2)$

$(a_1,a_2),(a_3,a_4)$...

$(a_2,a_3),(a_4,a_5)$...

Algorithm 9.3 Sequential odd-even transposition sort algorithm.


Unsorted
3 2 3 8 5 6 4 1
Phase 1 (odd)
2 3 3 8 5 6 1 4
Phase 2 (even)
2 3 3 5 8 1 6 4
Phase 3 (odd)
2 3 3 5 1 8 4 6
Phase 4 (even)
2 3 3 1 5 4 8 6
Phase 5 (odd)
2 3 1 3 4 5 6 8
Phase 6 (even)
2 1 3 3 4 5 6 8
Phase 7 (odd)
1 2 3 3 4 5 6 8
Phase 8 (even)
1 2 3 3 4 5 6 8
Sorted

Odd-Even Transposition Sort

- Easy to parallelize!
  - Θ(n) if 1 process/element.
  - Not cost optimal but use fewer processes, an optimal local sort, and compare-splits:

$$T_P = \Theta\left(\frac{n}{p}\log\frac{n}{p}\right) + \Theta(n) + \Theta(n)$$

local sor … tion

Cost optimal for p = O(log$n$) but not scalable (few processes).

30-03-2007 Alexandre David, MVP'07 23

Write speedup & efficiency to find the bound on p but you can also see it with T_P.

Odd-Even Transposition Sort

- Parallel formulation cost-optimal for $p=O(\log n)$.
- Isoefficiency function: $W=\Theta(p2^p)$. Exponential(p) $\Rightarrow$ poorly scalable.

Improvement: Shellsort

- 2 phases:
  - Move elements on longer distances.
  - Odd-even transposition but stop when no change.
- Idea: Put quickly elements near their final position to reduce the number of iterations of odd-even transposition.

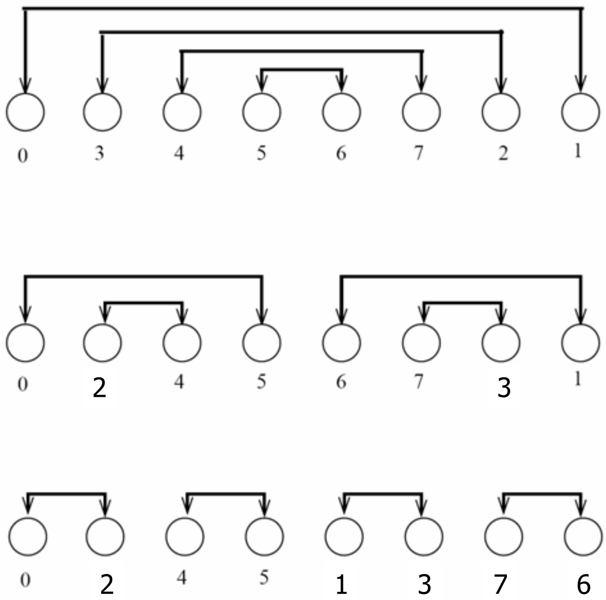
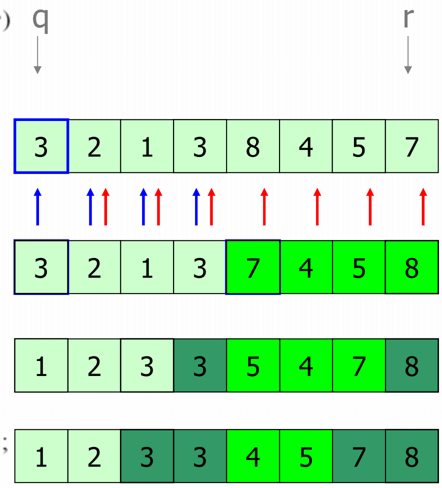


Figure 9.14 An example of the first phase of parallel shellsort on an eight-process array.

Quicksort

- Average complexity: O(n log$n$).
  - But very efficient in practice.
  - Average "robust".
  - Low overhead and very simple.
- Divide & conquer algorithm:
  - Partition A[q..r] into A[q..s] ≤ A[s+1..r].
  - Recursively sort sub-arrays.
  - Subtlety: How to partition?

```
procedure QUICKSORT (A, q, r)
begin
    if q < r then
    begin
        x := A[q];
        s := q;
        for i := q + 1 to r do
            if A[i] ≤ x then
            begin
                s := s + 1;
                swap(A[s], A[i]);
            end if
        swap(A[q], A[s]);
        QUICKSORT (A, q, s);
        QUICKSORT (A, s + 1, r);
    end if
end QUICKSORT
```



Algorithm 9.5 The sequential quicksort algorithm.

30-03-2007 Alexandre David, MVP'07 28

Hoare partitioning is better. Check in your algorithm course.

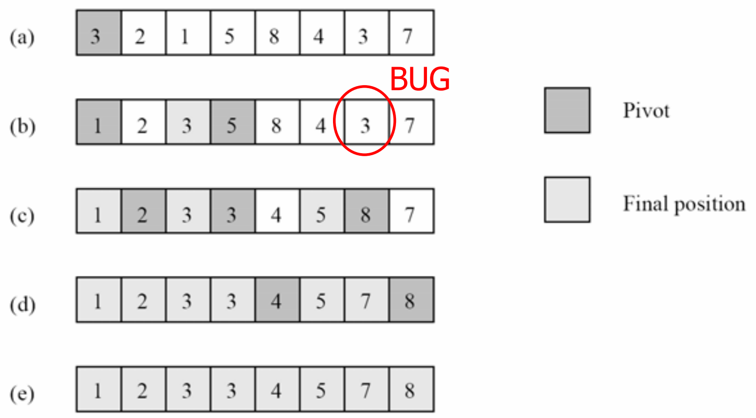


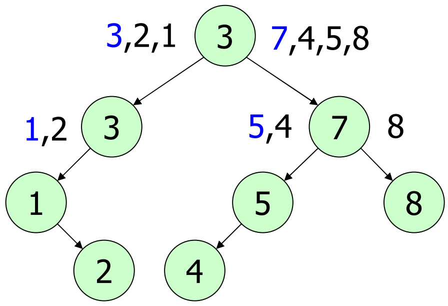
Figure 9.15 Example of the quicksort algorithm sorting a sequence of size $n = 8$.

Parallel Quicksort

- Simple version:
  - Recursive decomposition with one process per recursive call.
  - Not cost optimal: Lower bound = n (initial partitioning).
  - Best we can do: Use O(log$n$) processes.
  - Need to parallelize the partitioning step.

Parallel Quicksort for CRCW PRAM

- See execution of quicksort as constructing a binary tree.



Text & algorithm 9.5:
A[p..s] ≤ x < A[s+1..q].
Figures & algorithm 9.6:
A[p..s] < x ≤ A[s+1..q].

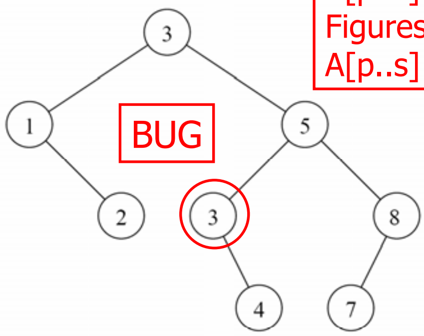
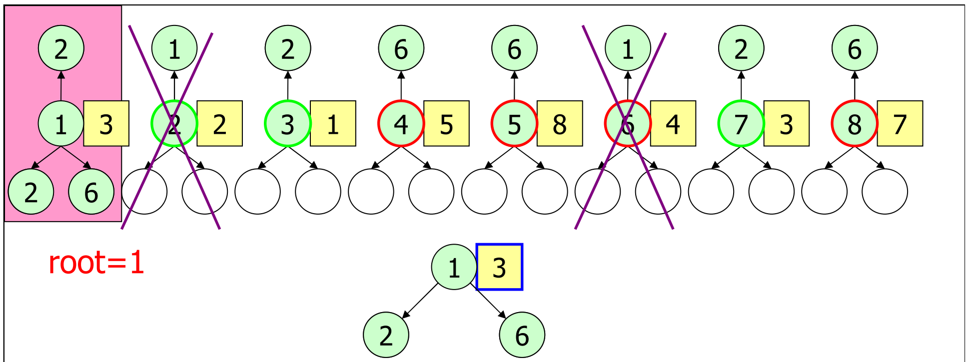


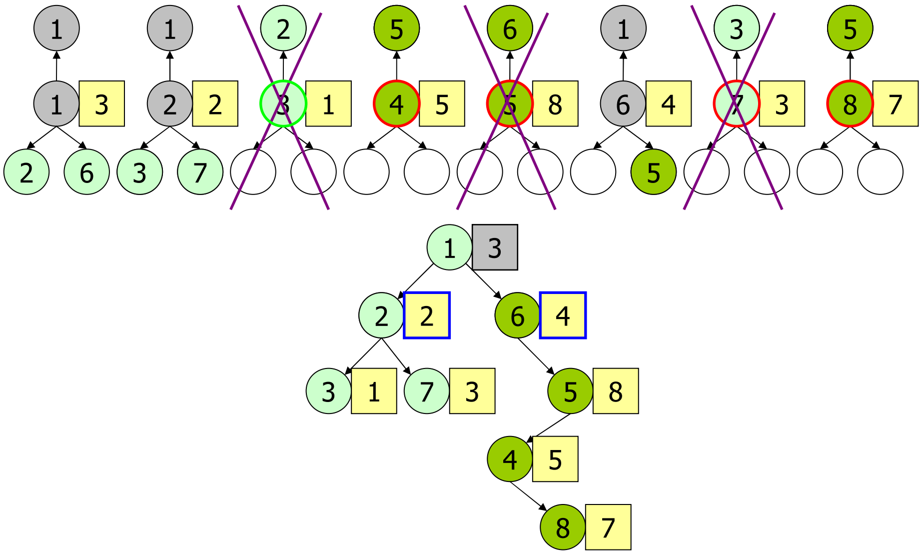
Figure 9.16 A binary tree generated by the execution of the quicksort algorithm. Each level of the tree represents a different array-partitioning iteration. If pivot selection is optimal, then the height of the tree is $\Theta(\log n)$, which is also the number of iterations.

```
procedure BUILD_TREE (A[1...n])
begin
    for each process i do
    begin
        root := i;    only one succeeds
        parent_i := root;
        leftchild[i] := rightchild[i] := n + 1;
    end for
    repeat for each process i ≠ root do
    begin
        if (A[i   A[i]≤A[parent_i]
            (A[                     ent_i) then
        begin
            leftchild[parent_i] := i;
            if i = leftchild[parent_i] then exit
            else parent_i := leftchild[parent_i];
        end for
        else
        begin
            rightchild[parent_i] := i;
            if i = rightchild[parent_i] then exit
            else parent_i := rightchild[parent_i];
        end else
    end repeat
end BUILD_TREE
```

33

This algorithm does not correspond exactly to the serial version. Time for partitioning: O(1).

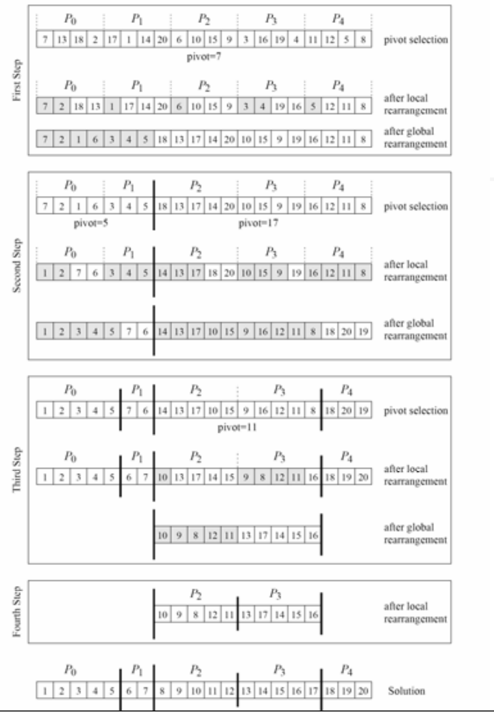

2
1
3
2
6
1
2
2
2
3
1
6
4
5
6
5
8
1
6
4
2
7
3
6
8
7
root=1
1
3
2
6

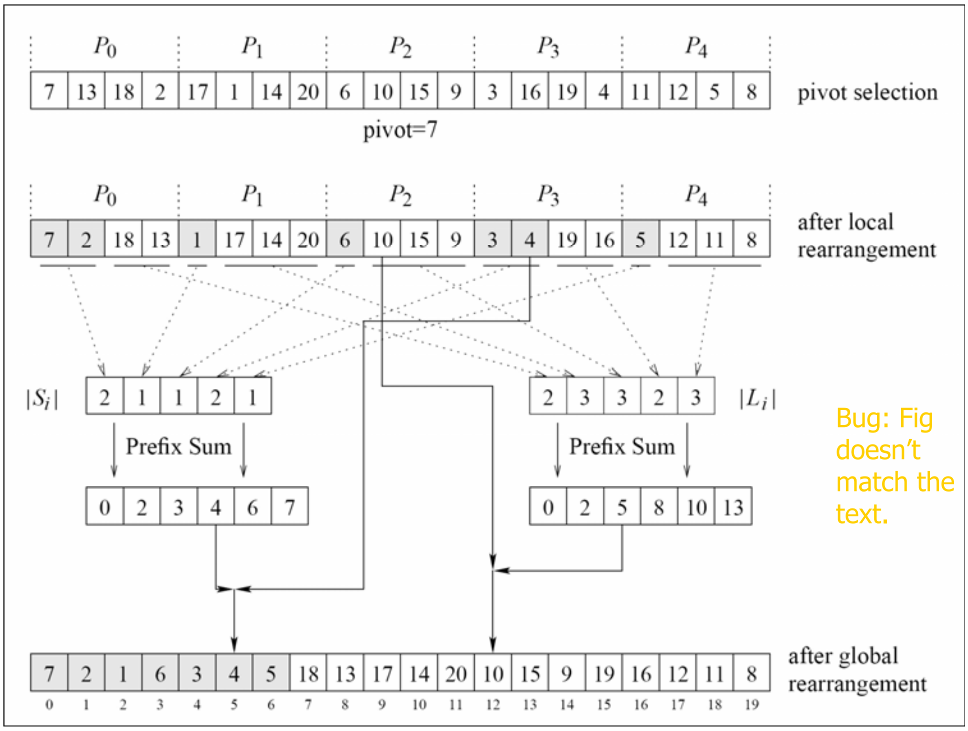


Each step: $\Theta(1)$. Average height: $\Theta(\log n)$.
This is cost-optimal – but it is only a model.

Parallel Quicksort – Shared Address (Realistic)

- Same idea but remove contention:
  - Choose the pivot & broadcast it.
  - Each process rearranges its block of elements *locally*.
  - *Global* rearrangement of the blocks.
  - When the blocks reach a certain size, local sort is used.


First Step
Second Step
Third Step
Fourth Step
P_0 P_1 P_2 P_3 P_4
pivot selection
after local rearrangement
after global rearrangement
pivot=7
pivot=5
pivot=17
pivot=11
Solution



Bug: Fig doesn't match the text.

Cost

- Scalability determined by time to broadcast the pivot & compute the prefix-sums.
- Cost optimal.

$$T_P = \overbrace{\Theta\left(\frac{n}{p}\log\frac{n}{p}\right)}^{\text{local sort}} + \overbrace{\Theta\left(\frac{n}{p}\log p\right) + \Theta(\log^2 p)}^{\text{array splits}}.$$

MPI Formulation of Quicksort

- Arrays must be explicitly distributed.
- Two phases:
  - Local partition smaller/larger than pivot.
  - Determine who will sort the sub-arrays.
    - And send the sub-arrays to the right process.

Final Word

- Pivot selection is very important.
- Affects performance.
- Bad pivot means idle processes.