

# Algorithms and Applications

Last update: Dec. 2018

Areas done in textbook:

- Sorting Algorithms
- Numerical Algorithms
- Image Processing
- Searching and Optimization

## Sorting Algorithms

- Rearranging a list of numbers into increasing (strictly non-decreasing) order.

# Potential Speedup

$O(n \log n)$  optimal for any sequential sorting algorithm without using special properties of the numbers.

Best we can expect based upon a sequential sorting algorithm using  $n$  processors is

$$\text{Optimal parallel time complexity} = \frac{O(n \log n)}{n} = O(\log n)$$

Has been obtained but the constant hidden in the order notation extremely large.

# General note

Sorting requires moving numbers from one place to another in a list.

The following algorithms concentrate upon a **message –passing model** for moving the numbers.

Sorting number is important in applications as it can make subsequent operations more efficient

# Compare-and-Exchange Sorting Algorithms

## Compare and Exchange

Form the basis of several, if not most, classical sequential sorting algorithms.

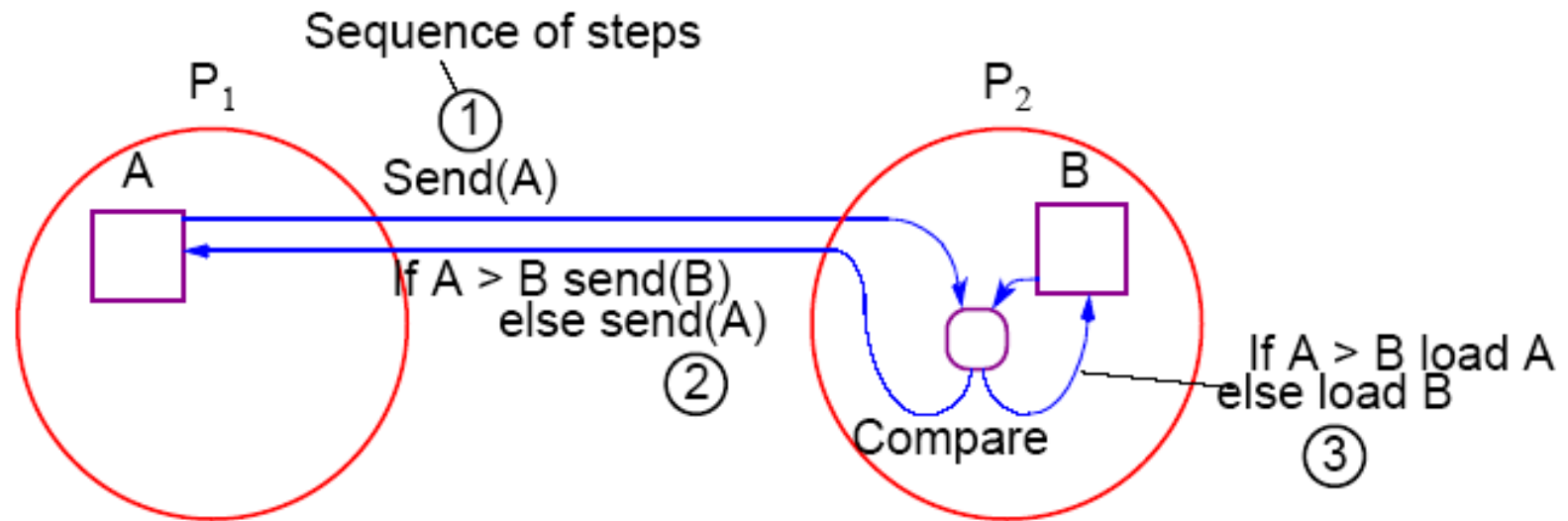
Two numbers, say  $A$  and  $B$ , are compared. If  $A > B$ ,  $A$  and  $B$  are exchanged, i.e.:

```
if (A > B) {  
    temp = A;  
    A = B;  
    B = temp;  
}
```

# Message-Passing Compare and Exchange

## Version 1

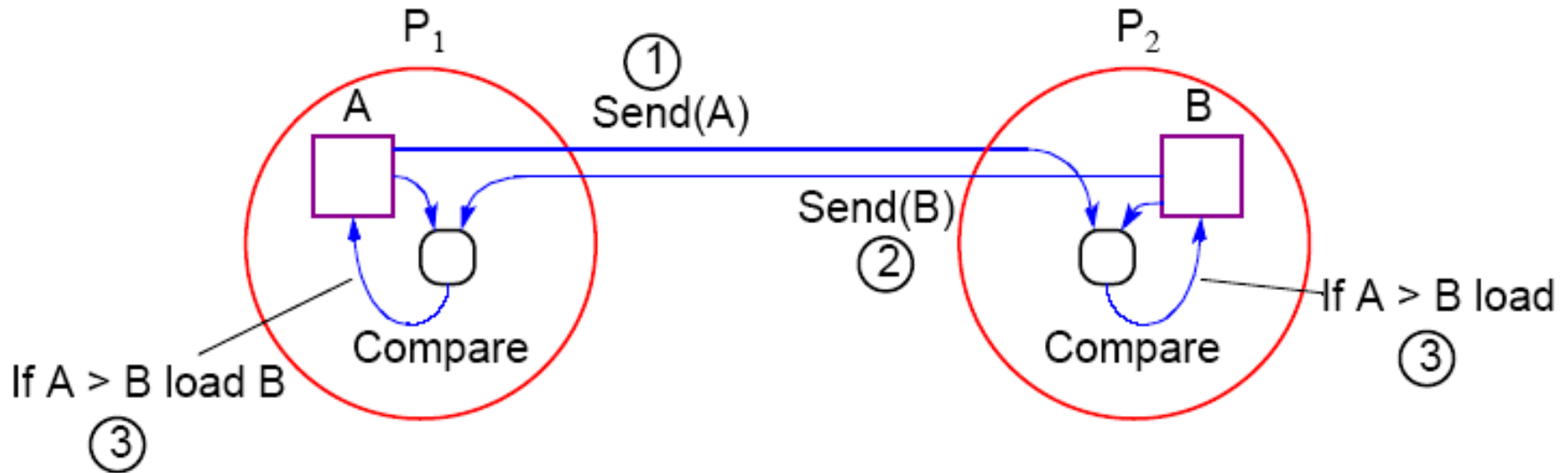
$P_1$  sends  $A$  to  $P_2$ , which compares  $A$  and  $B$  and sends back  $B$  to  $P_1$  if  $A$  is larger than  $B$  (otherwise it sends back  $A$  to  $P_1$ ):



# Alternative Message Passing Method

## Version 2

For  $P_1$  to send  $A$  to  $P_2$  and  $P_2$  to send  $B$  to  $P_1$ . Then both processes perform compare operations.  $P_1$  keeps the larger of  $A$  and  $B$  and  $P_2$  keeps the smaller of  $A$  and  $B$ :





# Note on Precision of Duplicated Computations

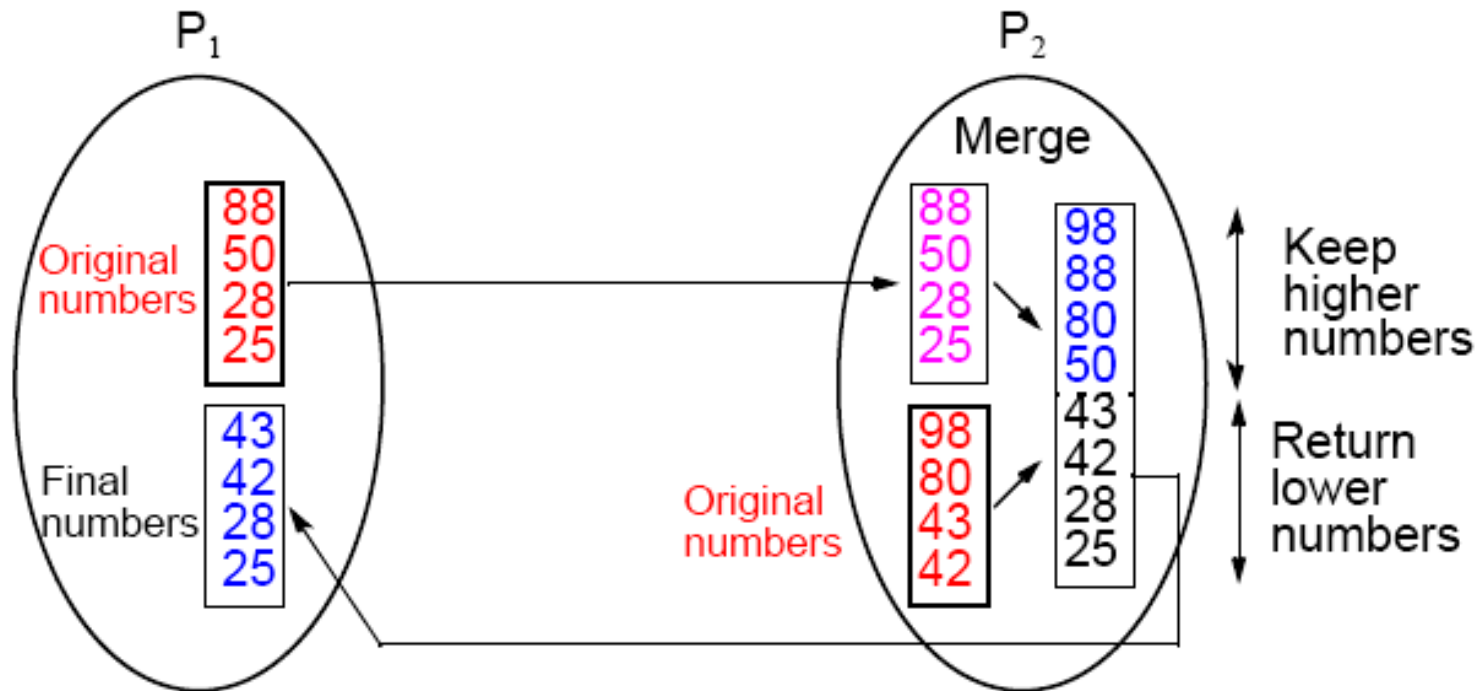
Previous code assumes that the **if** condition,  **$A > B$** , will return the same Boolean answer in both processors.

Different processors operating at different precision could conceivably produce **different answers** if real numbers are being compared.

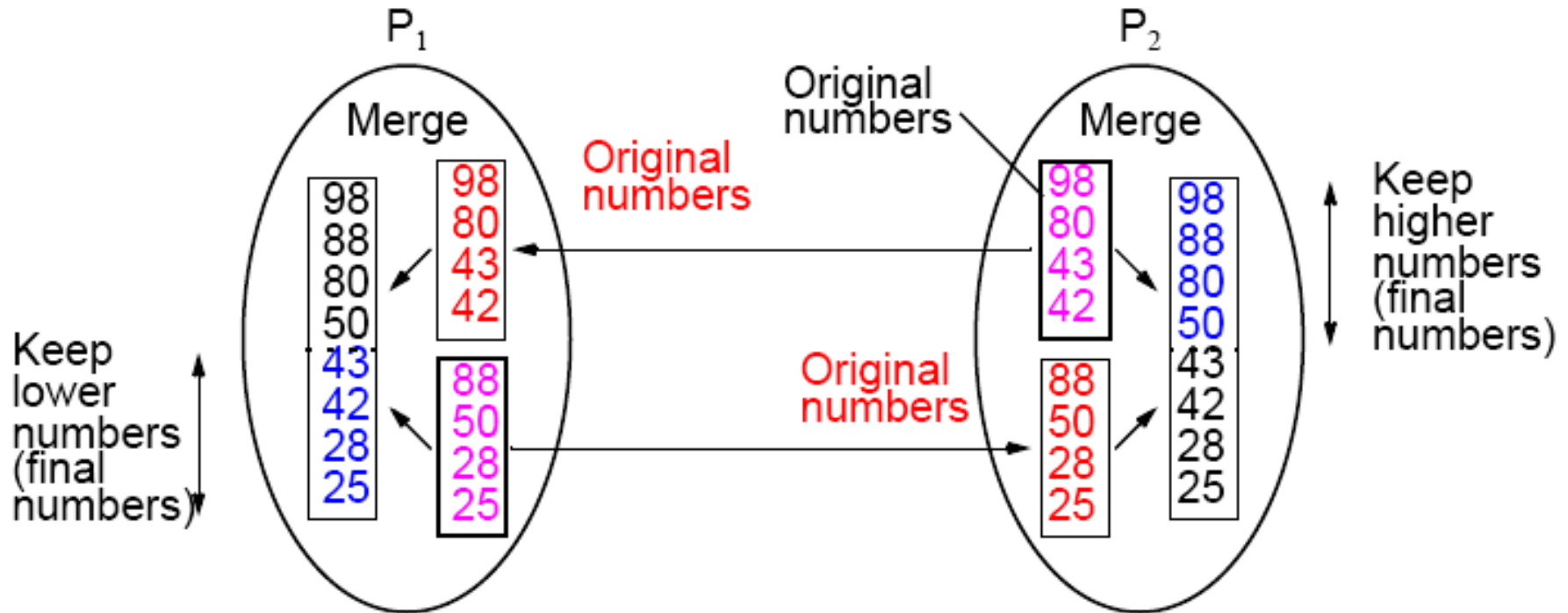
This situation applies to anywhere computations are duplicated in different processors to reduce message passing, or to make the code SPMD.

# Data Partitioning (Version 1)

$p$  processors and  $n$  numbers.  $n/p$  numbers assigned to each processor:



# Merging Two Sublists — Version 2

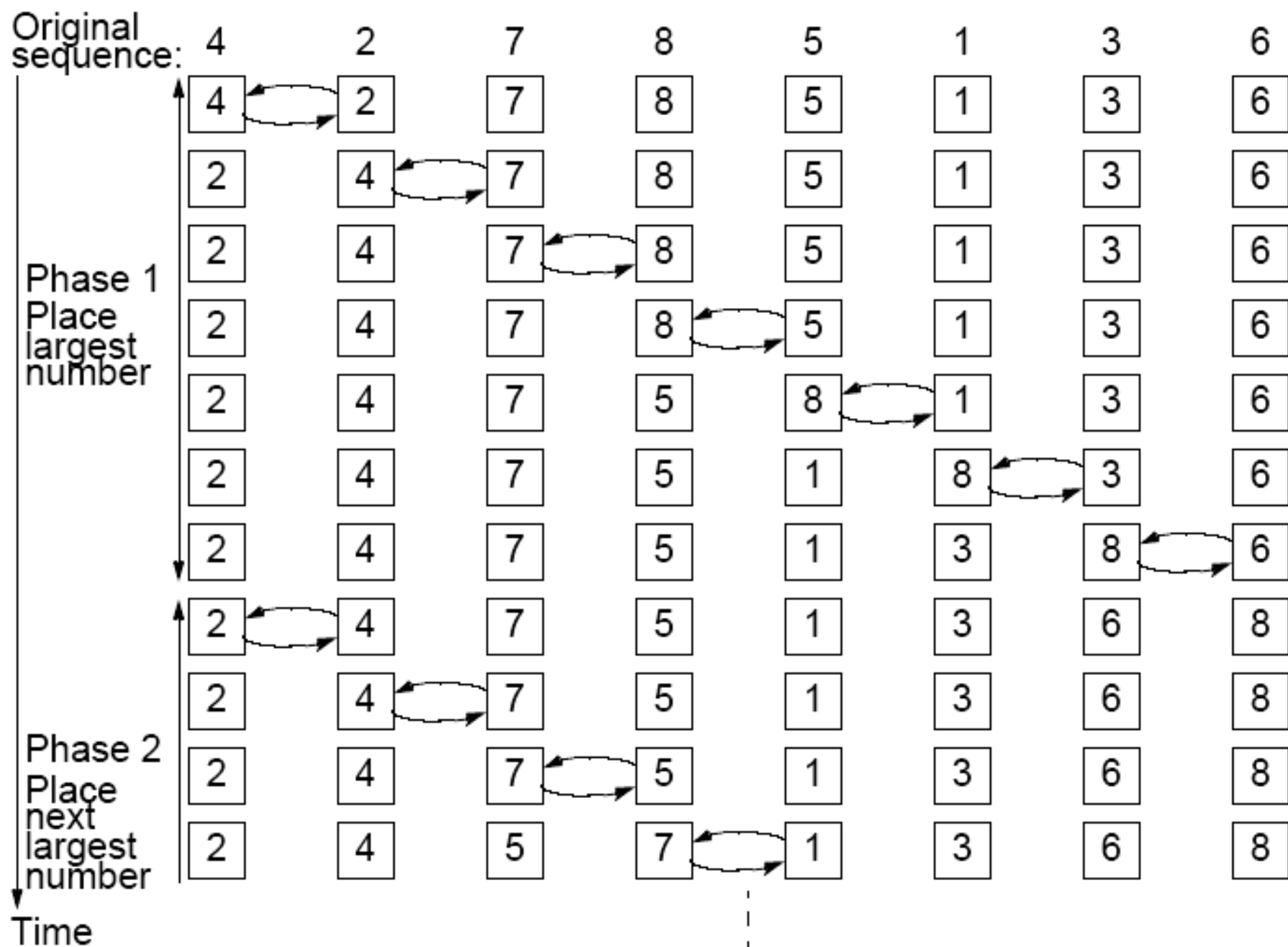


# Bubble Sort

First, largest number moved to the end of list by a series of compares and exchanges, starting at the opposite end.

Actions repeated with subsequent numbers, stopping just before the previously positioned number.

In this way, the larger numbers move (“bubble”) toward one end,



# Time Complexity

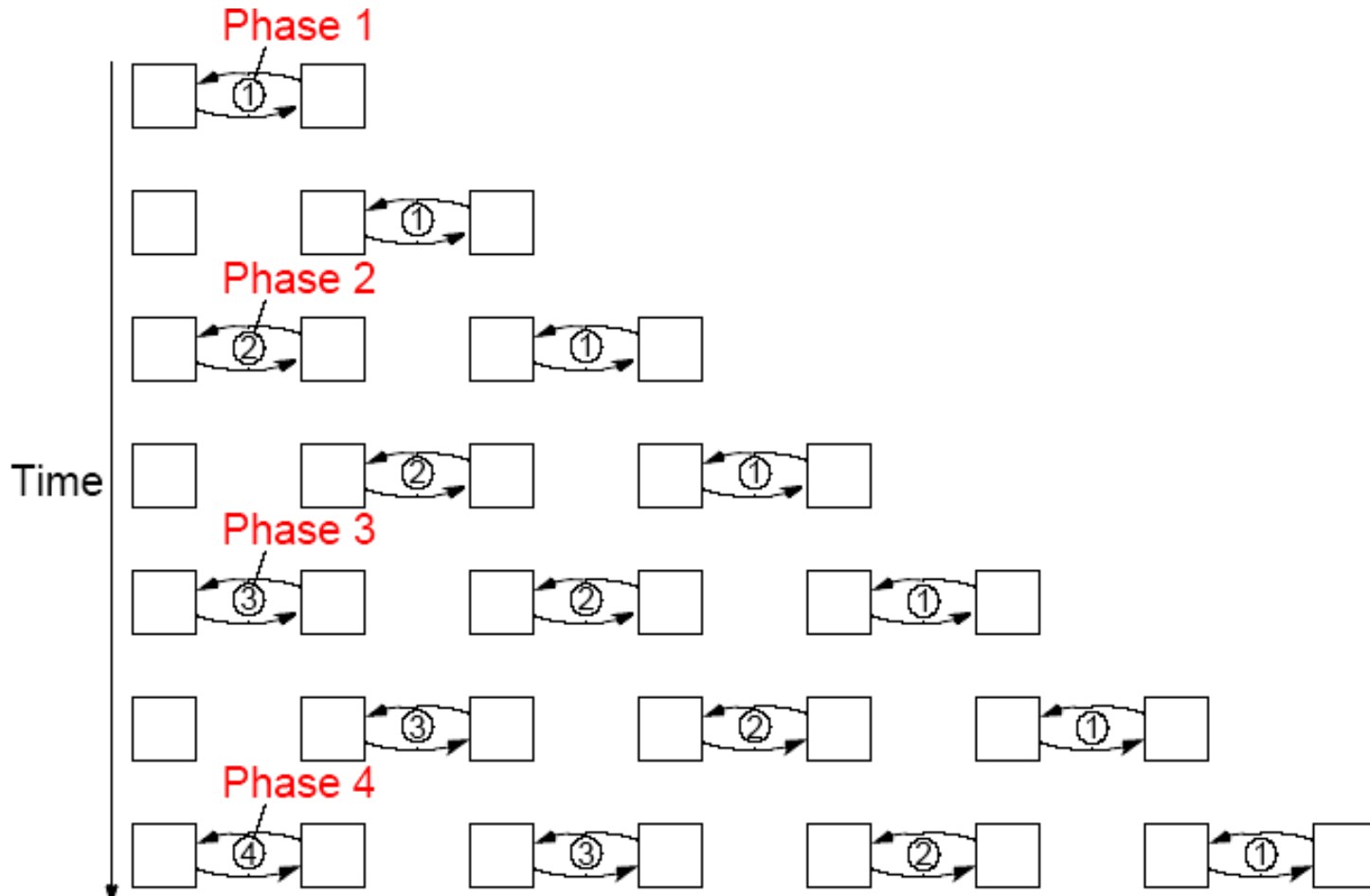
Number of compare and exchange operations

$$= \sum_{i=1}^{n-1} i = \frac{n(n-1)}{2}$$

Indicates a time complexity of  $O(n^2)$  given that a single compare-and-exchange operation has a constant complexity,  $O(1)$ .

# Parallel Bubble Sort

Iteration could start before previous iteration finished if does not overtake previous bubbling action:



# Odd-Even (Transposition) Sort

Variation of bubble sort.

Operates in two alternating phases, *even* phase and *odd* phase.

## Even phase

Even-numbered processes exchange numbers with their right neighbor.

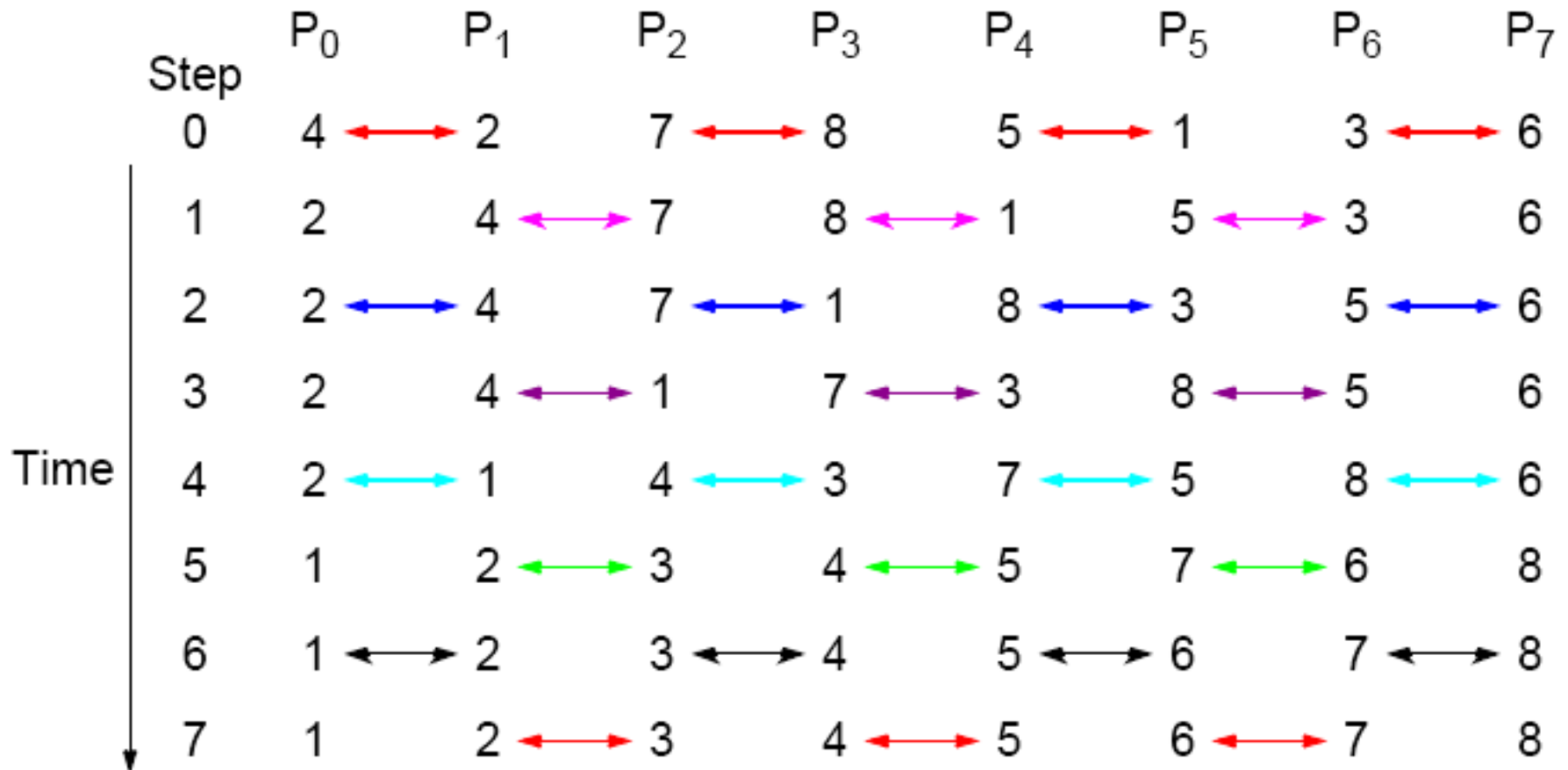
## Odd phase

Odd-numbered processes exchange numbers with their right neighbor.



# Odd-Even Transposition Sort

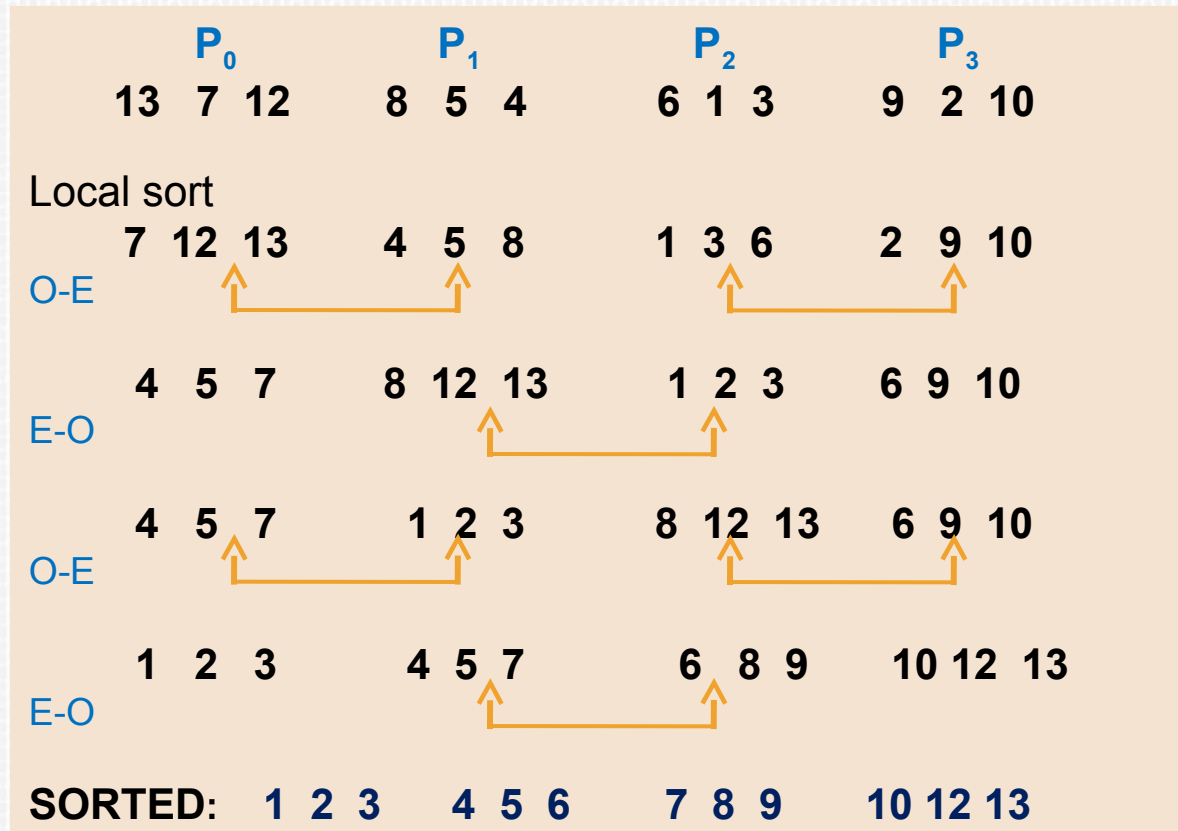
## Sorting eight numbers



# Odd-Even Transposition Sort – Example ( $N \gg$

$D \setminus$

Each PE gets  $n/p$  numbers. First, PEs sort  $n/p$  locally, then they run odd-even trans. algorithm each time doing a merge-split for  $2n/p$  numbers.



**Time complexity:**  $T_{\text{par}} = (\text{Local Sort}) + (p \text{ merge-splits}) + (p \text{ exchanges})$

$$T_{\text{par}} = (n/p)\log(n/p) + p*(n/p) + p*(n/p) = (n/p)\log(n/p) + 2n$$

# Mergesort

A classical sequential sorting algorithm using divide-and-conquer approach. Unsorted list first divided into half. Each half is again divided into two. Continued until individual numbers are obtained.

Then pairs of numbers combined (merged) into sorted list of two numbers. Pairs of these lists of four numbers are merged into sorted lists of eight numbers. This is continued until the one fully sorted list is obtained.

# John von Neumann

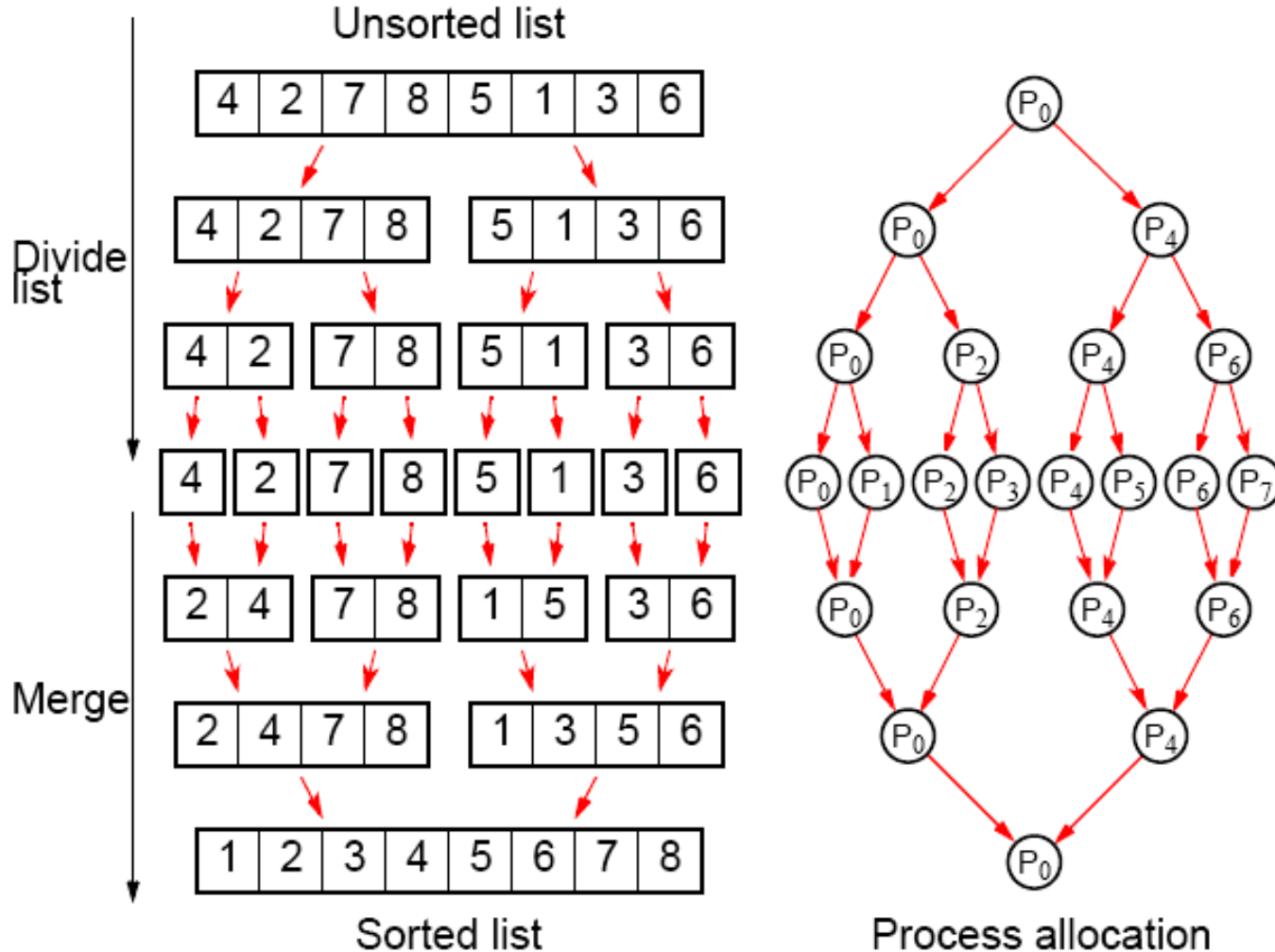


# Animated merge sort

6 5 3 1 8 7 2 4

# Parallelizing Mergesort

Using tree allocation of processes



# Analysis

## Sequential

Sequential time complexity is  $O(n \log n)$ .

## Parallel

$2 \log n$  steps in the parallel version but each step may need to perform more than one basic operation, depending upon the number of numbers being processed - see text.

# Quicksort

Very popular **sequential** sorting algorithm that performs well with average sequential time complexity of  $O(n \log n)$ .

First list divided into two sublists. All numbers in one sublist arranged to be smaller than all numbers in other sublist.

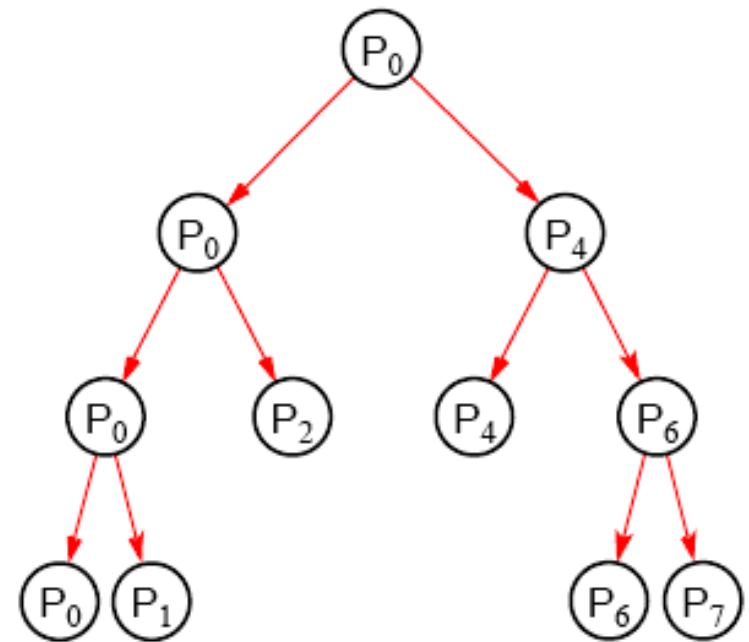
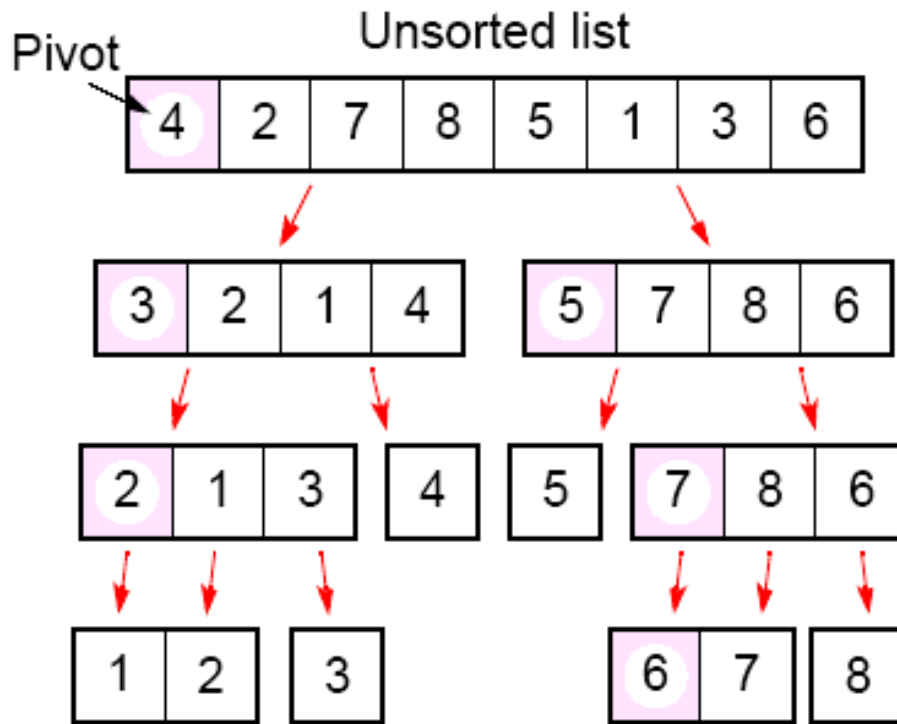
Achieved by first selecting one number, called a *pivot*, against which every other number is compared. If the number is less than the pivot, it is placed in one sublist. Otherwise, it is placed in the other sublist.

Pivot could be any number in the list, but often first number in list chosen. Pivot itself could be placed in one sublist, or the pivot could be separated and placed in its final position.



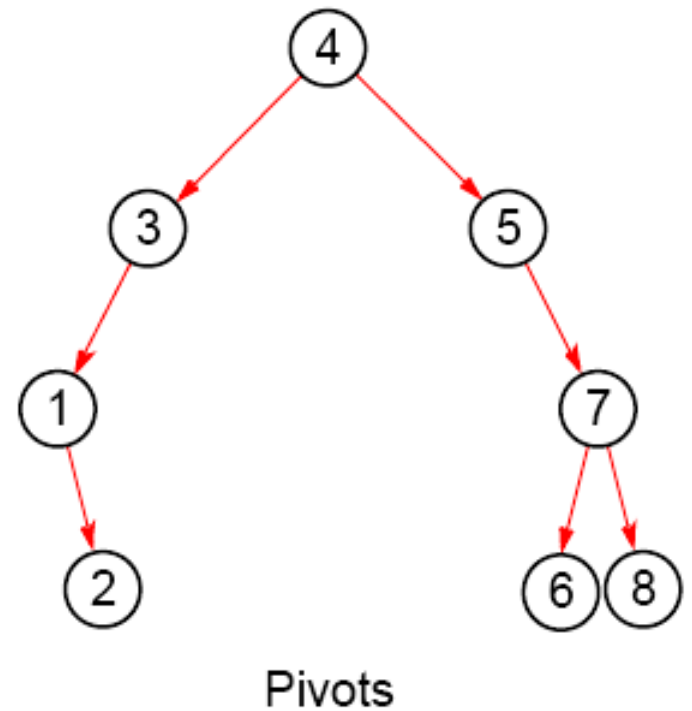
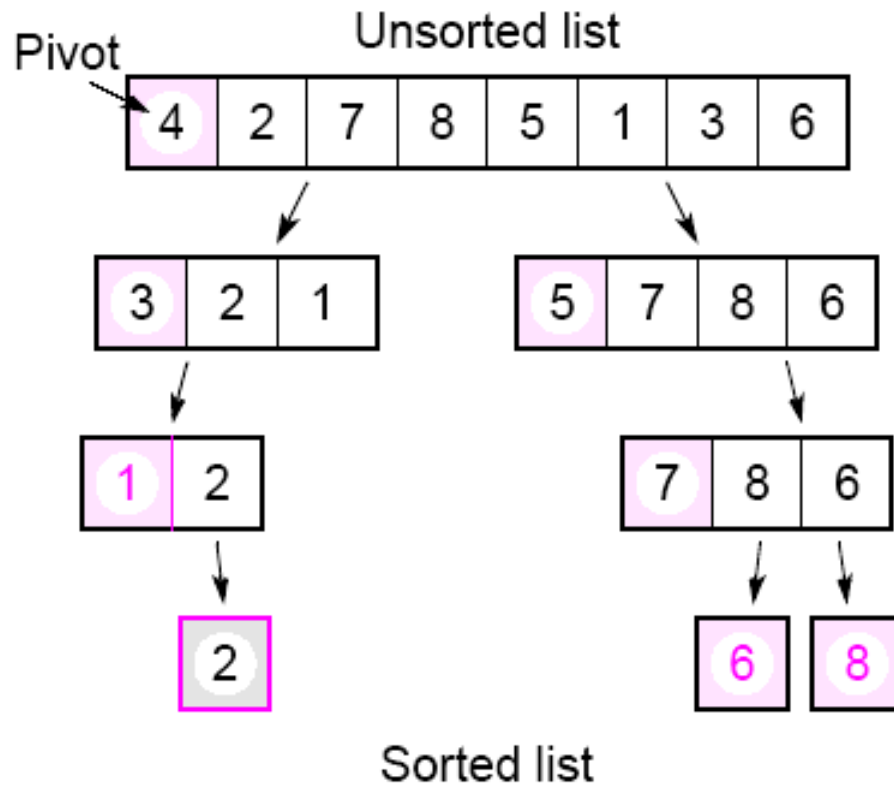
# Parallelizing Quicksort

Using tree allocation of processes



Process allocation

With the pivot being withheld in processes:



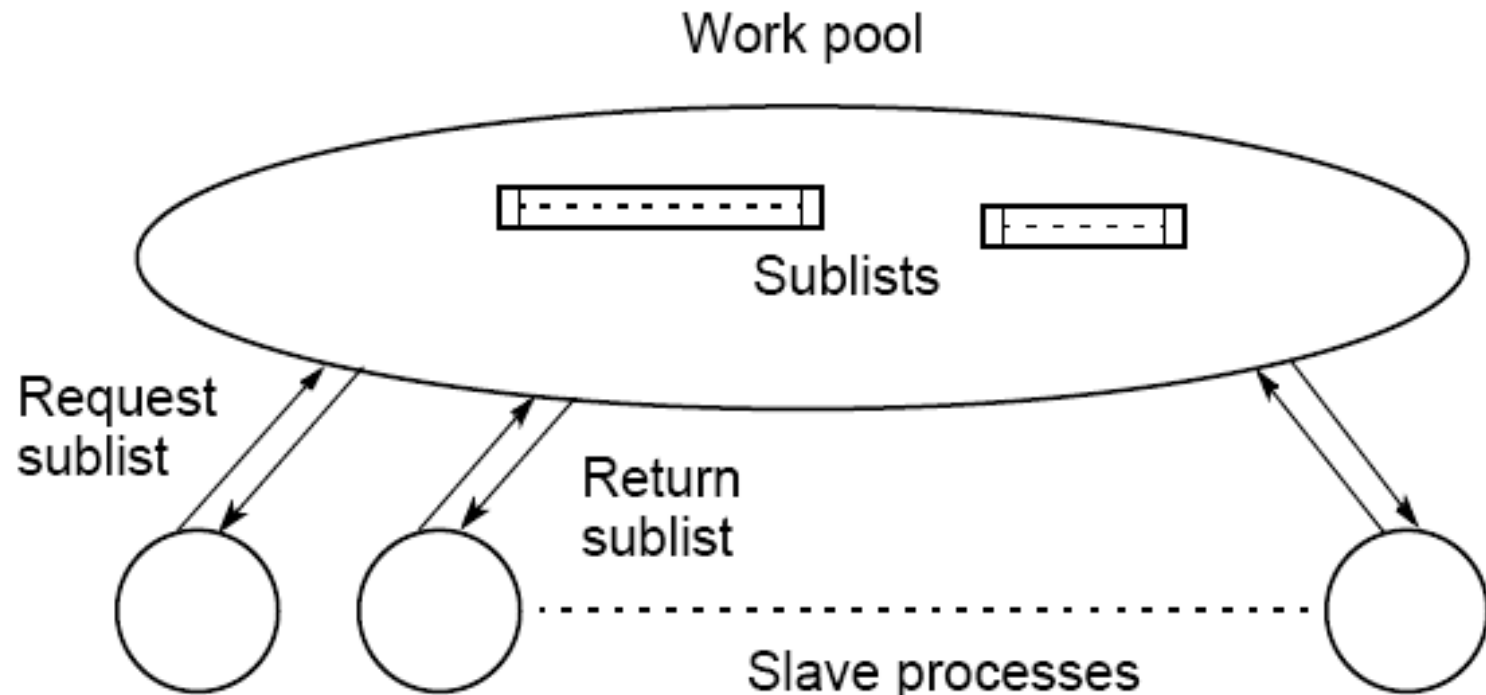
# Analysis

Fundamental problem with all tree constructions – initial division done by a single processor, which will seriously limit speed.

Tree in quicksort will not, in general, be perfectly balanced  
Pivot selection very important to make quicksort operate fast.

# Work Pool Implementation of Quicksort

First, work pool holds initial unsorted list. Given to first processor which divides list into two parts. One part returned to work pool to be given to another processor, while the other part operated upon again.



Neither Mergesort nor Quicksort parallelize very well as the processor efficiency is low (see book for analysis).

Quicksort also can be very unbalanced. Can use load balancing techniques.

# Batcher's Parallel Sorting Algorithms

- Odd-even Mergesort
- Bitonic Mergesort

Originally derived in terms of switching networks.

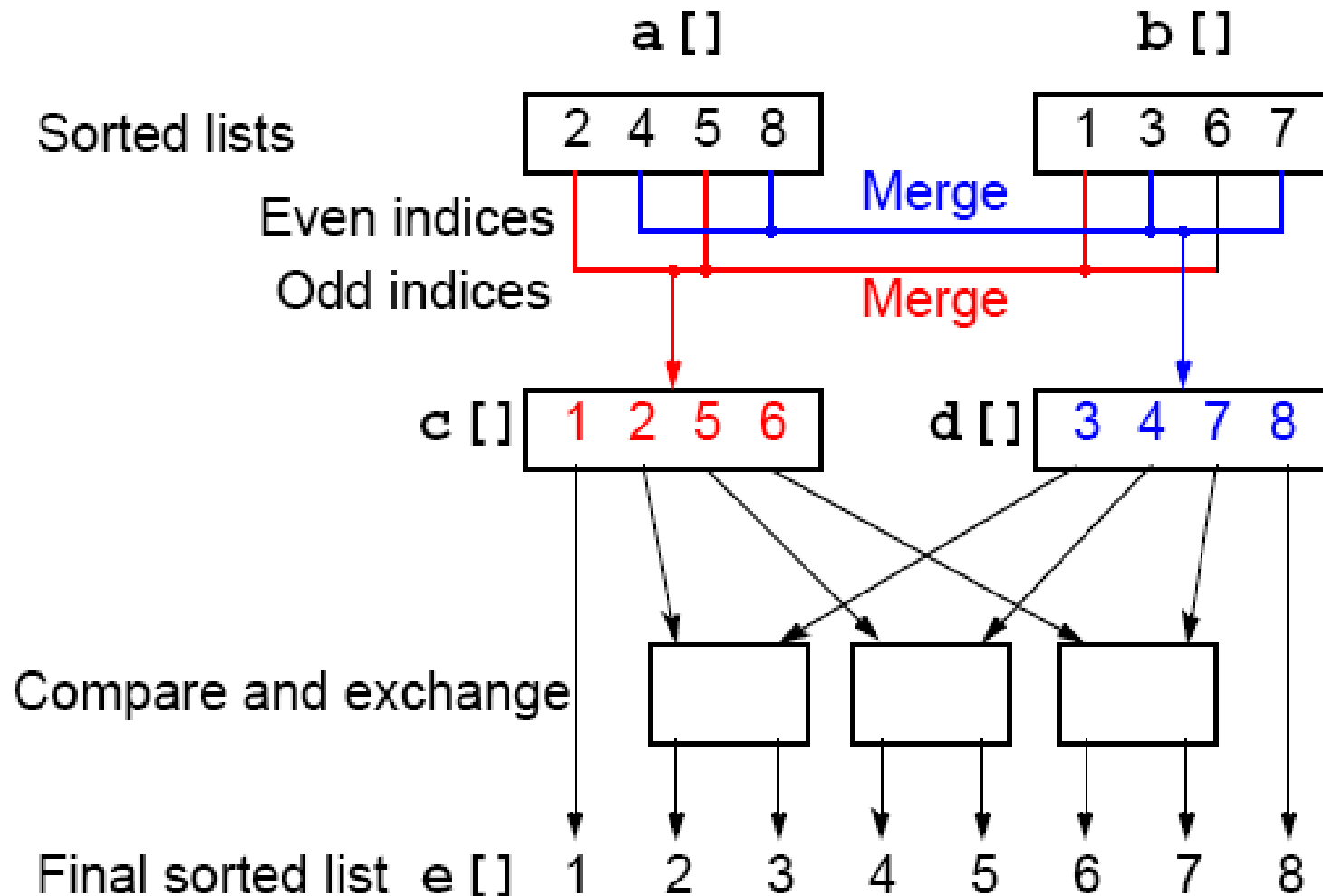
Both are well balanced and have parallel time complexity of  $O(\log^2 n)$  with  $n$  processors.

# Odd-Even Mergesort

## Odd-Even Merge Algorithm

Start with odd-even merge algorithm which will merge two *sorted* lists into one sorted list. Given two **sorted** lists  $a_1, a_2, a_3, \dots, a_n$  and  $b_1, b_2, b_3, \dots, b_n$  (where  $n$  is a power of 2).

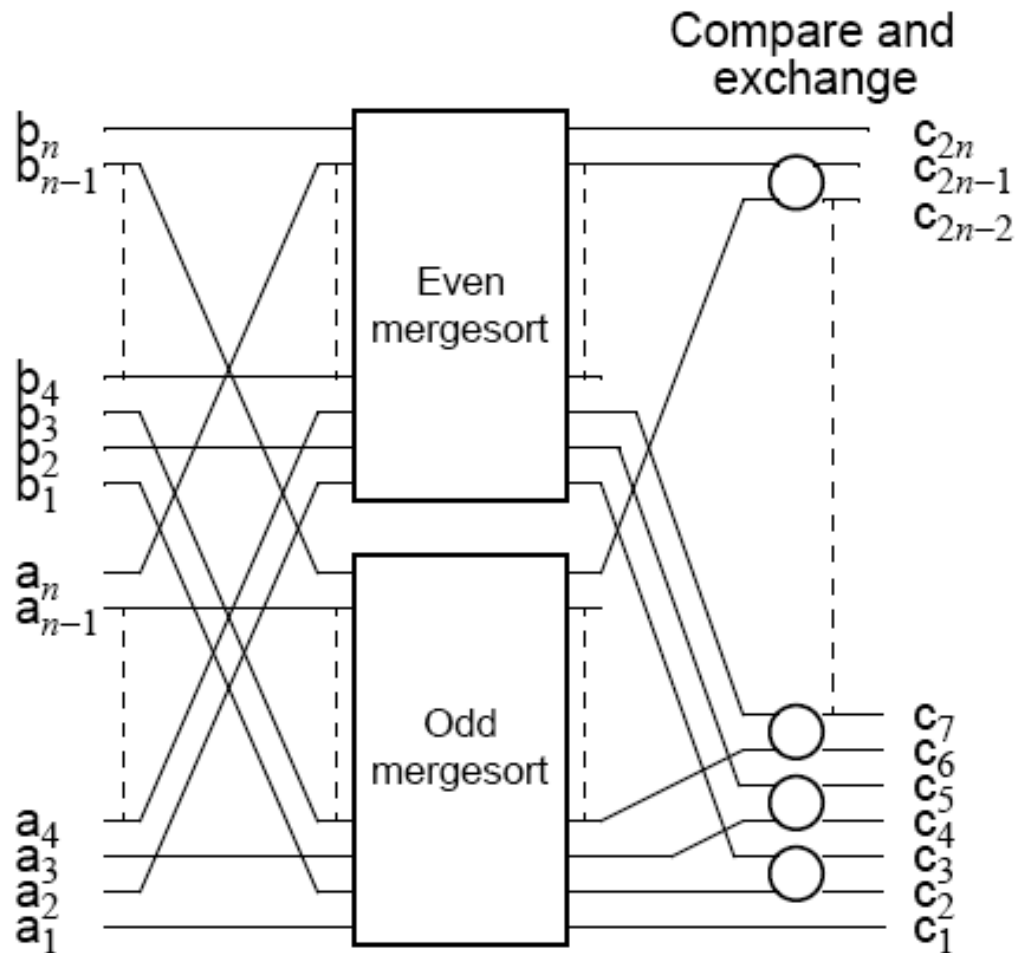
# Odd-Even Merging of Two Sorted Lists





# Odd-Even Mergesort

Apply odd-even merging recursively



# Bitonic Mergesort

## Bitonic Sequence

A *monotonic* increasing sequence is a sequence of increasing numbers.

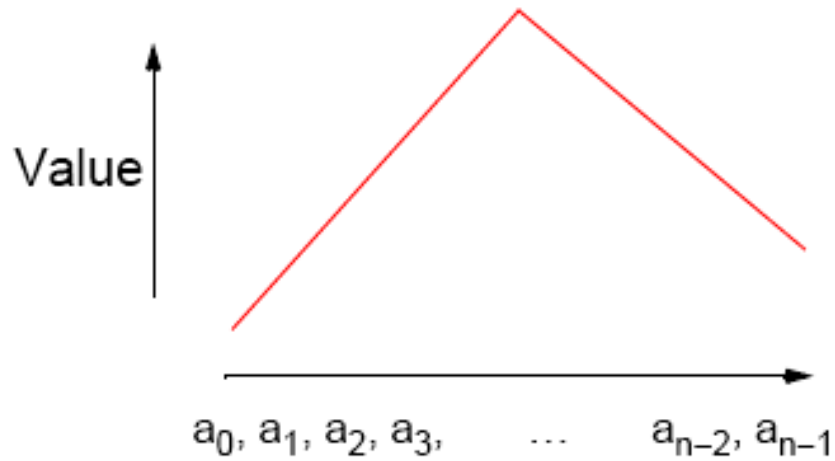
A *bitonic sequence* has two sequences, one increasing and one decreasing. e.g.

$$a_0 < a_1 < a_2, a_3, \dots, a_{i-1} < a_i > a_{i+1}, \dots, a_{n-2} > a_{n-1}$$

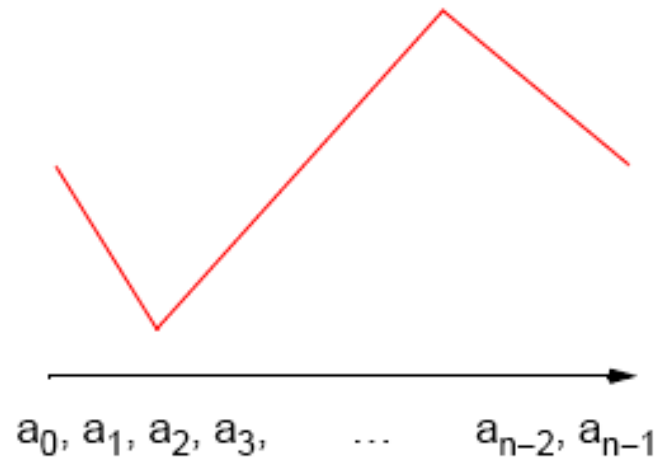
for some value of  $i$  ( $0 \leq i < n$ ).

A sequence is also bitonic if the preceding can be achieved by shifting the numbers cyclically (left or right).

# Bitonic Sequences



(a) Single maximum



(b) Single maximum and single minimum

# “Special” Characteristic of Bitonic Sequences

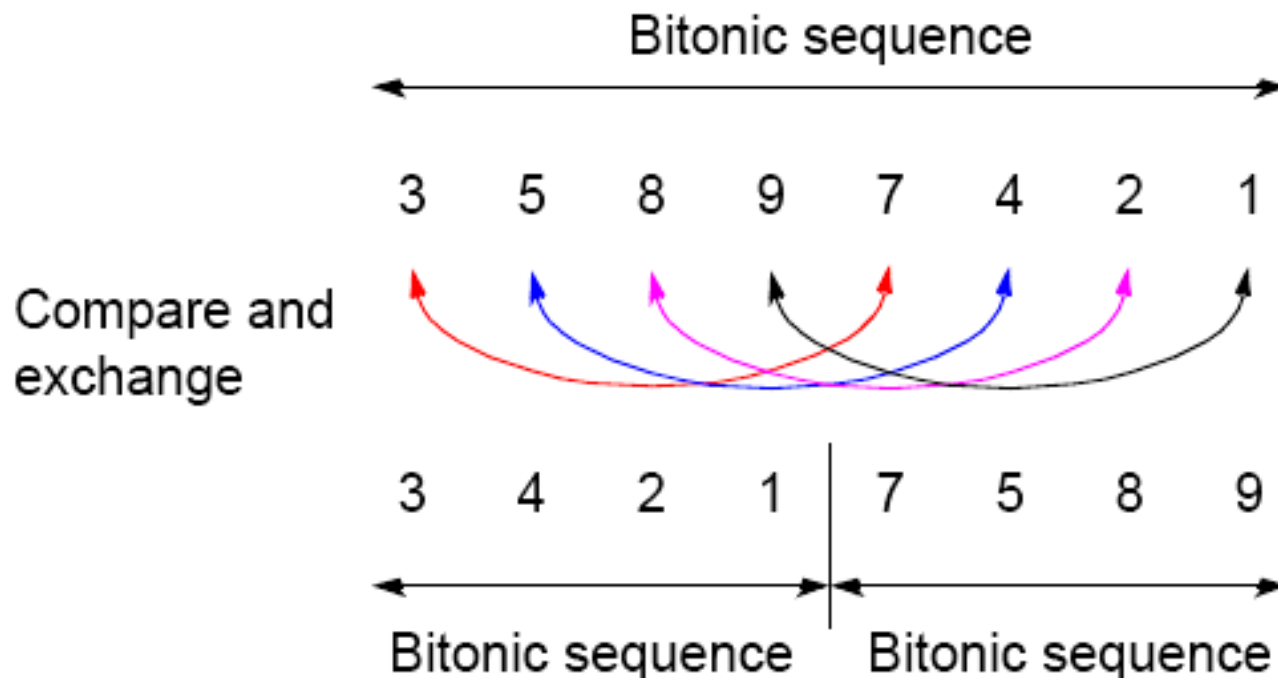
If we perform a compare-and-exchange operation on  $a_i$  with  $a_{i+n/2}$  for all  $i$ , where there are  $n$  numbers in the sequence, get **TWO** bitonic sequences, where the numbers in one sequence are **all less than the numbers in the other sequence.**

# Creating two bitonic sequences from one bitonic sequence

Starting with the bitonic sequence

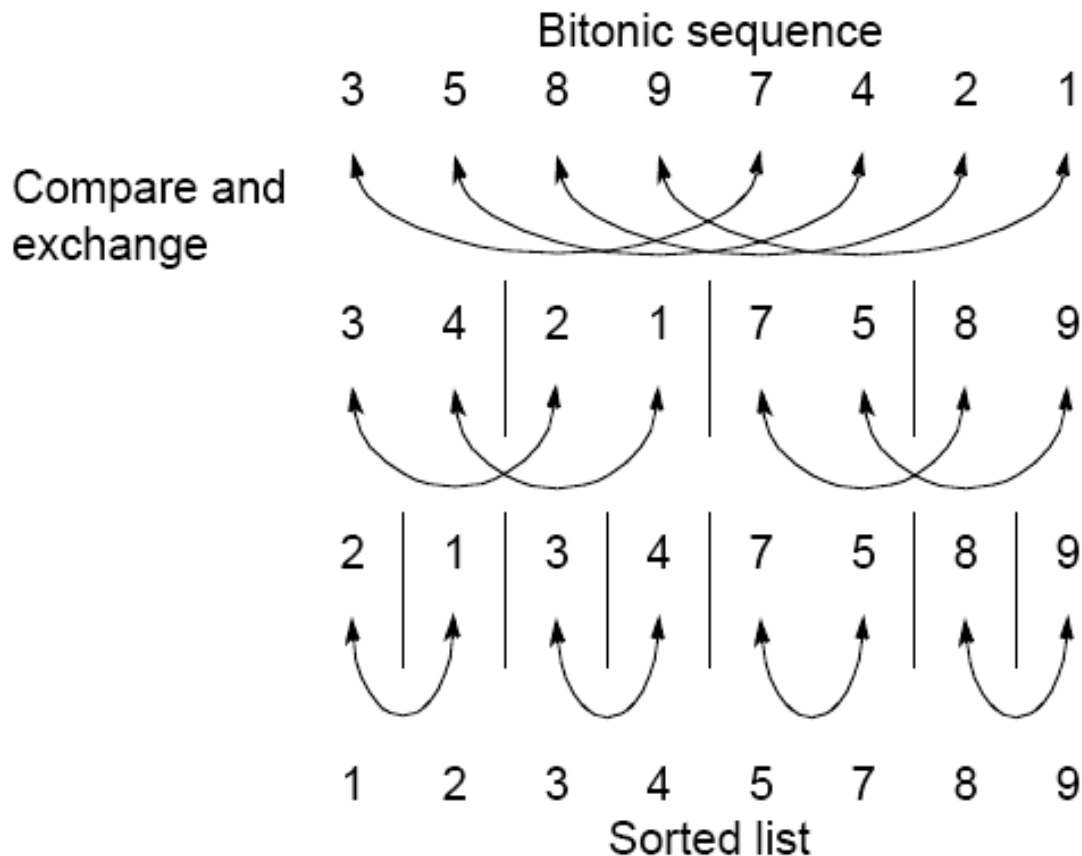
3, 5, 8, 9, 7, 4, 2, 1

we get:



# Sorting a bitonic sequence

Compare-and-exchange moves smaller numbers of each pair to left and larger numbers of pair to right. **Given a bitonic sequence**, recursively performing operations will sort the list.



# Sorting

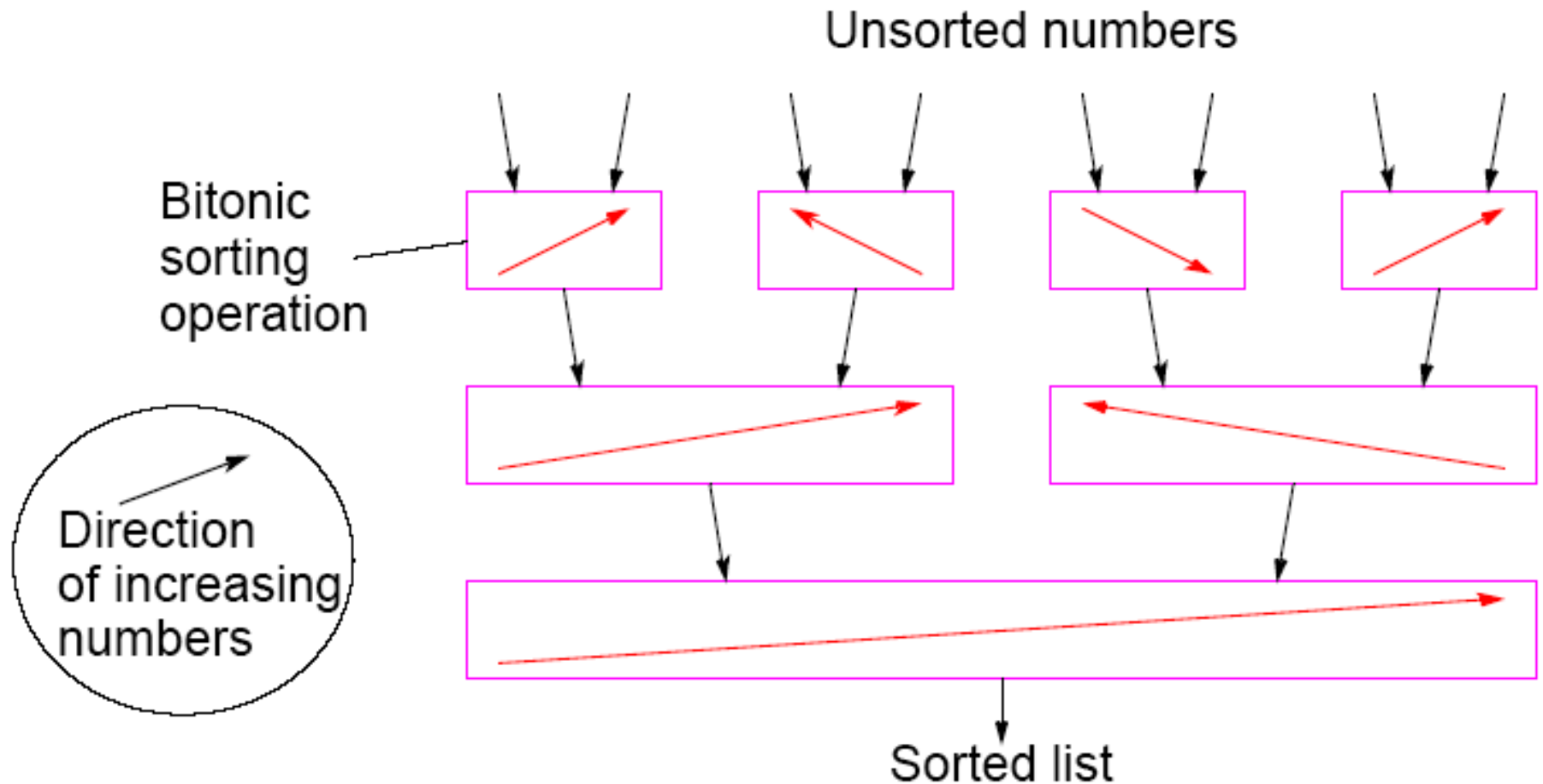
To sort an **unordered sequence**, sequences are merged into larger bitonic sequences, starting with pairs of adjacent numbers.

By a compare-and-exchange operation, pairs of adjacent numbers formed into increasing sequences and decreasing sequences. Pairs form a bitonic sequence of twice size of each original sequences.

By repeating this process, bitonic sequences of larger and larger lengths obtained.

In the final step, a single bitonic sequence sorted into a single increasing sequence.

# Bitonic Mergesort



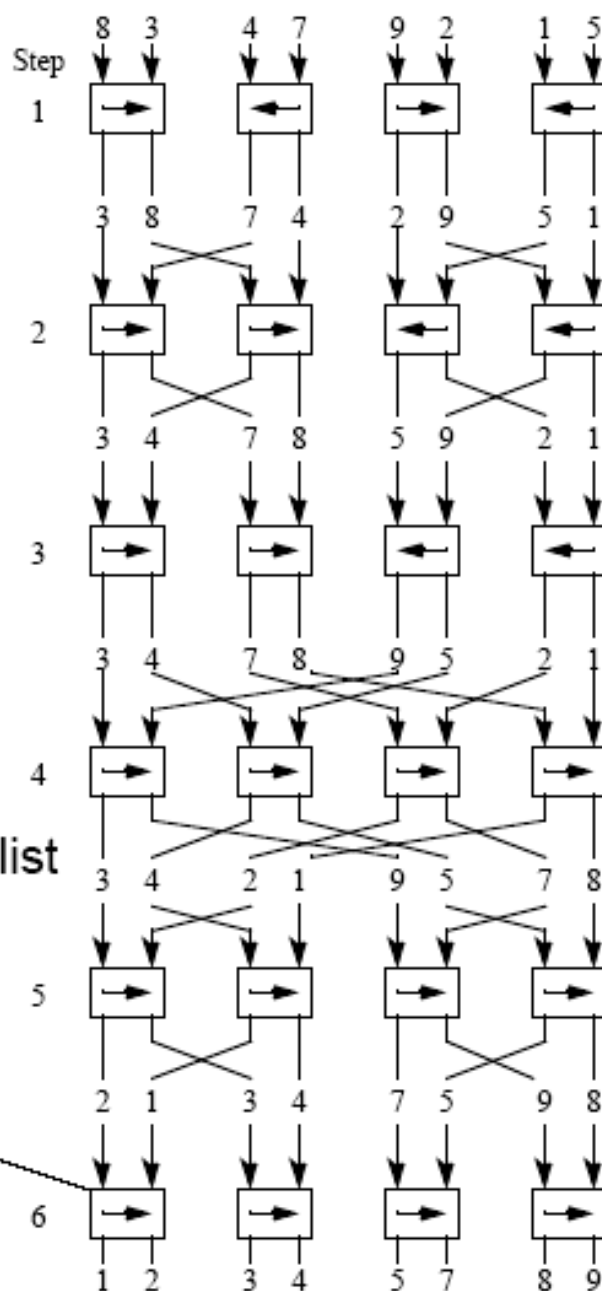
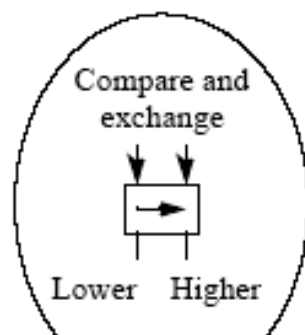


# Bitonic Mergesort on Eight Numbers

Form bitonic lists of four numbers

Form bitonic list of eight numbers

Sort bitonic list



# Phases

The six steps (for eight numbers) are divided into three phases:

Phase 1 (Step 1)      Convert pairs of numbers into increasing/decreasing sequences and into 4-bit bitonic sequences.

Phase 2 (Steps 2/3)      Split each 4-bit bitonic sequence into two 2-bit bitonic sequences, higher sequences at center.  
Sort each 4-bit bitonic sequence  
increasing/decreasing sequences and merge into 8-bit  
bitonic sequence.

Phase 3 (Steps 4/5/6) Sort 8-bit bitonic sequence.

# Number of Steps

In general, with  $n = 2^k$ , there are  $k$  phases, each of 1, 2, 3, ...,  $k$  steps. Hence the total number of steps is given by

$$\text{Steps} = \sum_{i=1}^k i = \frac{k(k+1)}{2} = \frac{\log n (\log n + 1)}{2} = O(\log^2 n)$$

# Sorting Conclusions so far

Computational time complexity using  $n$  processors

- Odd-even transposition sort -  $O(n)$
- Parallel mergesort -  $O(n)$  but unbalanced processor load and Communication
- Parallel quicksort -  $O(n)$  but unbalanced processor load, and communication can generate to  $O(n^2)$
- Odd-even Mergesort and Bitonic Mergesort  $O(\log^2 n)$   
**Bitonic mergesort has been a popular choice for a parallel sorting.**

# Sorting on Specific Networks

Algorithms can take advantage of the underlying interconnection network of the parallel computer.

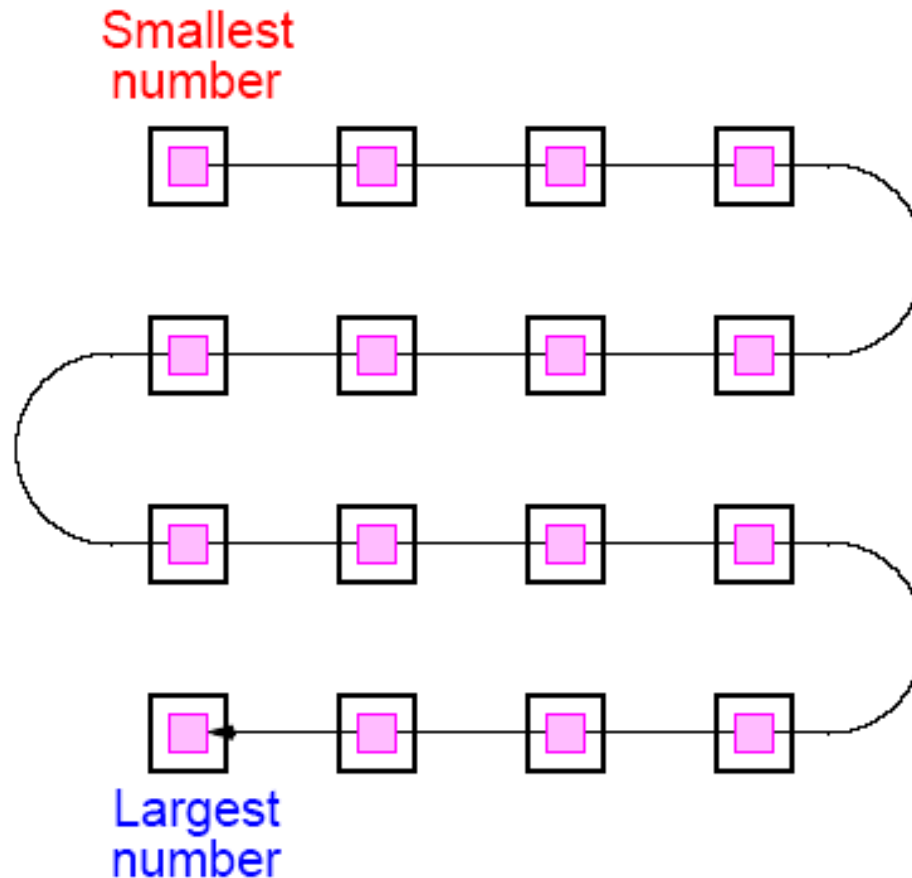
Two network structures have received specific attention: the **mesh** and **hypercube** because parallel computers have been built with these networks.

Of less interest nowadays because underlying architecture often hidden from user.

MPI features for mapping algorithms onto meshes, and one can always use a mesh or hypercube algorithm even if the underlying architecture is not the same.

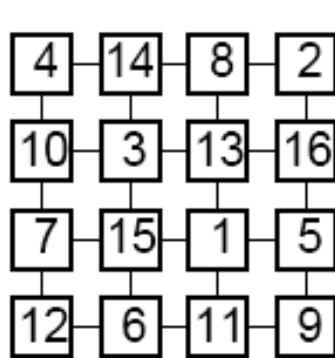
# Mesh - Two-Dimensional Sorting

The layout of a sorted sequence on a mesh could be row by row or *snakelike*. Snakelike:

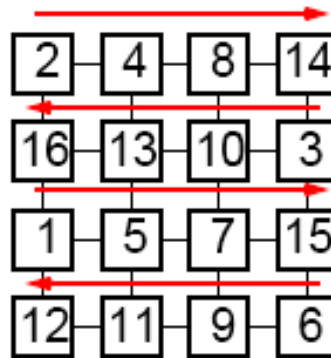


# Shearsort

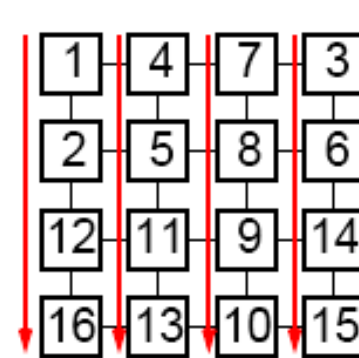
Alternate row and column sorting until list fully sorted. Row sorting alternative directions to get snake-like sorting:



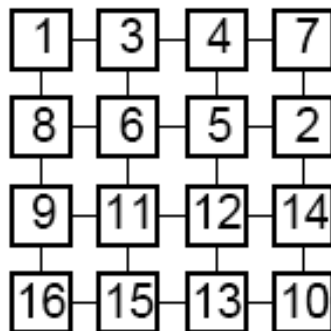
(a) Original placement of numbers



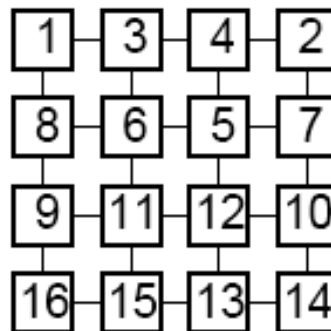
(b) Phase 1 — Row sort



(c) Phase 2 — Column sort



(d) Phase 3 — Row sort



(e) Phase 4 — Column sort



(f) Final phase — Row sort

# Shearsort

Requires  $\sqrt{n}(\log n + 1)$  steps for  $n$  numbers on a  $\sqrt{n} \times \sqrt{n}$  mesh.

הביטוי הזה של הספר לא עומד במבחן המציאות. ניקח  
את הדוגמה מהשקף הקודם.  $N=16$

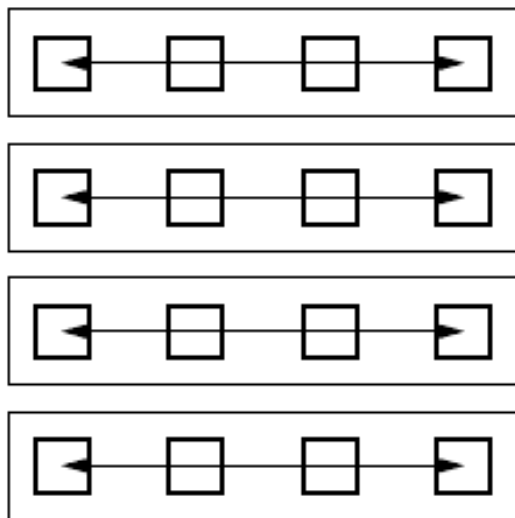
$$4*(4+1)=20$$

אנחנו ראים שצריך רק 5 שלבים  
אולי הביטוי הנכון הוא  
 $\log(n)+1=5$

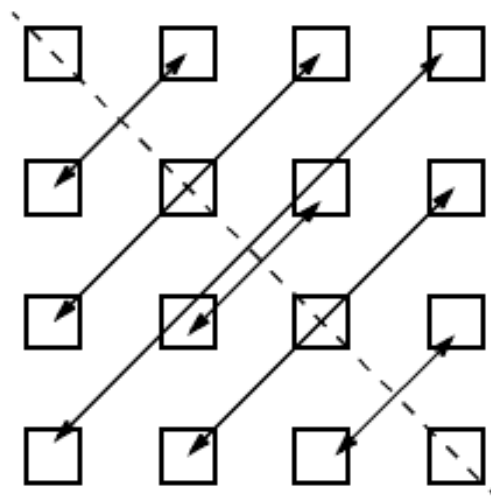


# Using Transposition

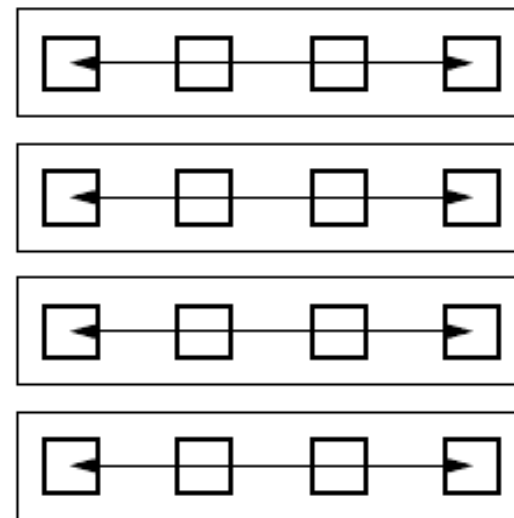
Causes the elements in each column to be in positions in a row.  
Can be placed between the row operations and column operations



(a) Operations between elements in rows



(b) Transpose operation



(c) Operations between elements in rows (originally columns)

Transposition can be achieved with  $\sqrt{n}(\sqrt{n} - 1)$  communications ( $O(n)$ ). An *all-to-all* routine could be reduce this.

# Other Sorting Algorithms

We began by giving the lower bound for the time complexity of a sequential sorting algorithm based upon comparisons as  $O(n\log n)$ .

Consequently, the time complexity of a parallel sorting algorithm based upon comparisons is  $O((\log n)/p)$  with  $p$  processors or  $O(\log n)$  with  $n$  processors.

There are sorting algorithms that can achieve better than  $O(n\log n)$  sequential time complexity and are very attractive candidates for parallelization but they often assume special properties of the numbers being sorted.

## Rank Sort as basis of a parallel sorting algorithm

Does not achieve a sequential time of  $O(n \log n)$ , but can be parallelized easily, and leads us onto linear sequential time algorithms which can be parallelized to achieve  $O(\log n)$  parallel time and are attractive algorithms for clusters.

# Rank Sort

Number of numbers that are smaller than each selected number counted. This count provides the position of selected number in sorted list; that is, its “rank.”

- First **a[0]** is read and compared with each of the other numbers, **a[1] ... a[n-1]**, recording the number of numbers less than **a[0]**.
- Suppose this number is  $x$ . This is the index of the location in the final sorted list.
- The number **a[0]** is copied into the final sorted list **b[0] ... b[n-1]**, at location **b[x]**. Actions repeated with the other numbers.

Overall sequential sorting time complexity of  $O(n^2)$  (not exactly a good sequential sorting algorithm!).

# Sequential Code

```
for (i = 0; i < n; i++) {                                /* for each number */
    x = 0;
    for (j = 0; j < n; j++) /* count number less than it */
        if (a[i] > a[j]) x++;
    b[x] = a[i];    /* copy number into correct place */
}
```

This code will fail if duplicates exist in the sequence of numbers. Easy to fix. (How?)

# Parallel Code

## Using $n$ Processors

One processor allocated to each number. Finds final index in  $O(n)$  steps. With all processors operating in parallel, parallel time complexity  $O(n)$ .

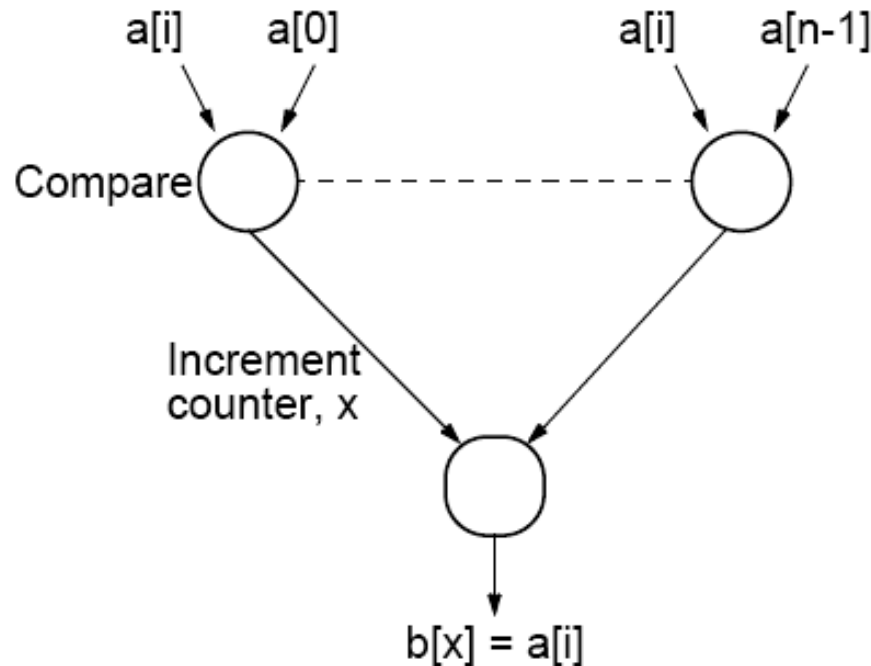
In **forall** notation, the code would look like

```
forall (i = 0; i < n; i++) {      /* for each no in parallel*/
    x = 0;
    for (j = 0; j < n; j++) /* count number less than it */
        if (a[i] > a[j]) x++;
    b[x] = a[i];                /* copy no into correct place */
}
```

Parallel time complexity,  $O(n)$ , as good as any sorting algorithm so far. Can do even better if we have more processors.

# Using $n^2$ Processors

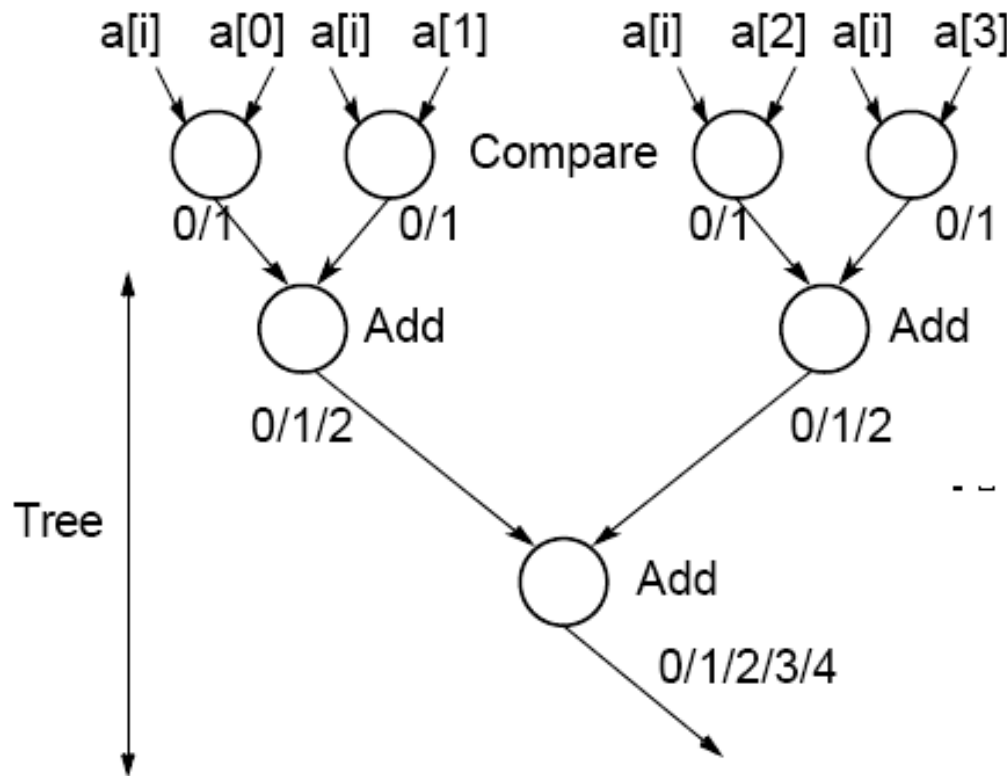
Comparing one number with the other numbers in list using multiple processors:



$n - 1$  processors used to find rank of one number. With  $n$  numbers,  $(n - 1)n$  processors or (almost)  $n^2$  processors needed. Incrementing counter done sequentially and requires maximum of  $n$  steps. Total number of steps =  $1 + n$ .

# Reduction in Number of Steps

Tree to reduce number of steps involved in incrementing counter:



$O(\log n)$  algorithm with  $n^2$  processors.  
Processor efficiency relatively low.



# Parallel Rank Sort Conclusions

Easy to do as each number can be considered in isolation.

Rank sort can sort in:

$O(n)$  with  $n$  processors

or

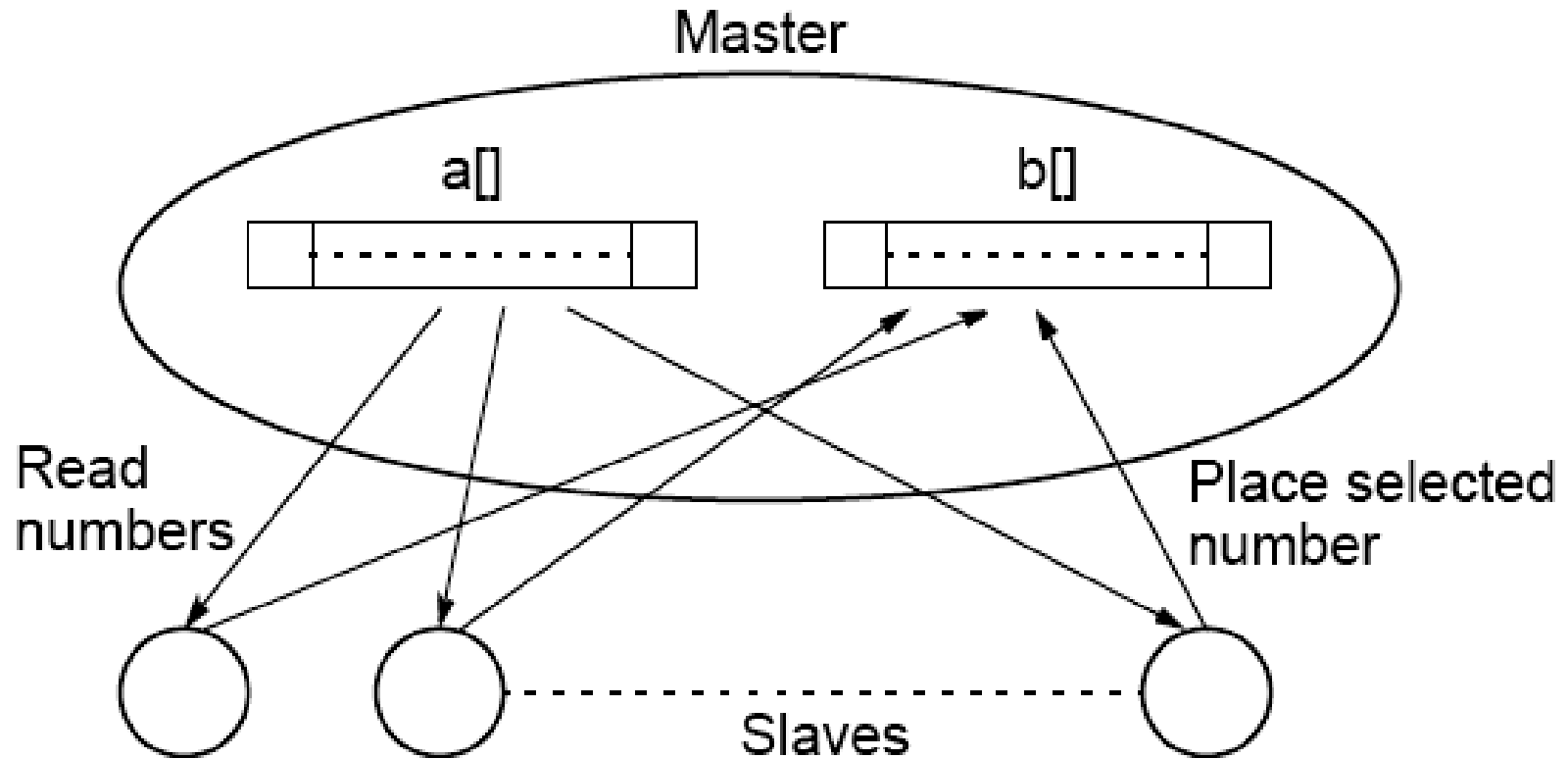
$O(\log n)$  using  $n^2$  processors.

In practical applications, using  $n^2$  processors prohibitive.

Theoretically possible to reduce time complexity to  $O(1)$  by considering all increment operations as happening in parallel since they are independent of each other.

# Message Passing Parallel Rank Sort

## Master-Slave Approach



Requires shared access to list of numbers. Master process responds to request for numbers from slaves. Algorithm better for shared memory

# Counting Sort

If the numbers to be sorted are integers, there is a way of coding the rank sort algorithm to reduce the sequential time complexity from  $O(n^2)$  to  $O(n)$ , called as *Counting Sort*.

As in the rank sort code suppose the unsorted numbers stored in an array  $a[]$  and final sorted sequence is stored in array  $b[]$ . Algorithm uses an additional array, say  $c[]$ , having one element for each possible value of the numbers. Suppose the range of integers is from 1 to  $m$ . The array has element  $c[1]$  through  $c[m]$  inclusive. Now, let us working through the algorithm in stages.

# Stable Sort Algorithms

Algorithms that will place identical numbers in the same order as in the original sequence.

Counting sort is naturally a **stable sorting algorithm**.

First, **c[ ]** will be used to hold the histogram of the sequence, that is, the number of each number. This can be computed in  $O(m)$  time with code such as:

```
for (i = 1; i <= m; i++)  
    c[i] = 0;  
for (i = 1; i <= m; i++)  
    c[a[i]]++;
```

Next stage: The number of numbers less than each number found by performing a prefix sum operation on array **c[ ]**.

In the prefix sum calculation, given a list of numbers,  $x_0, \dots, x_{n-1}$ , all the partial summations (i.e.,  $x_0$ ;  $x_0 + x_1$ ;  $x_0 + x_1 + x_2$ ;  $x_0 + x_1 + x_2 + x_3$ ;  $\dots$  ) are computed.

Here, the prefix sum is computed using the histogram originally held in **c[ ]** in  $O(m)$  time as described below:

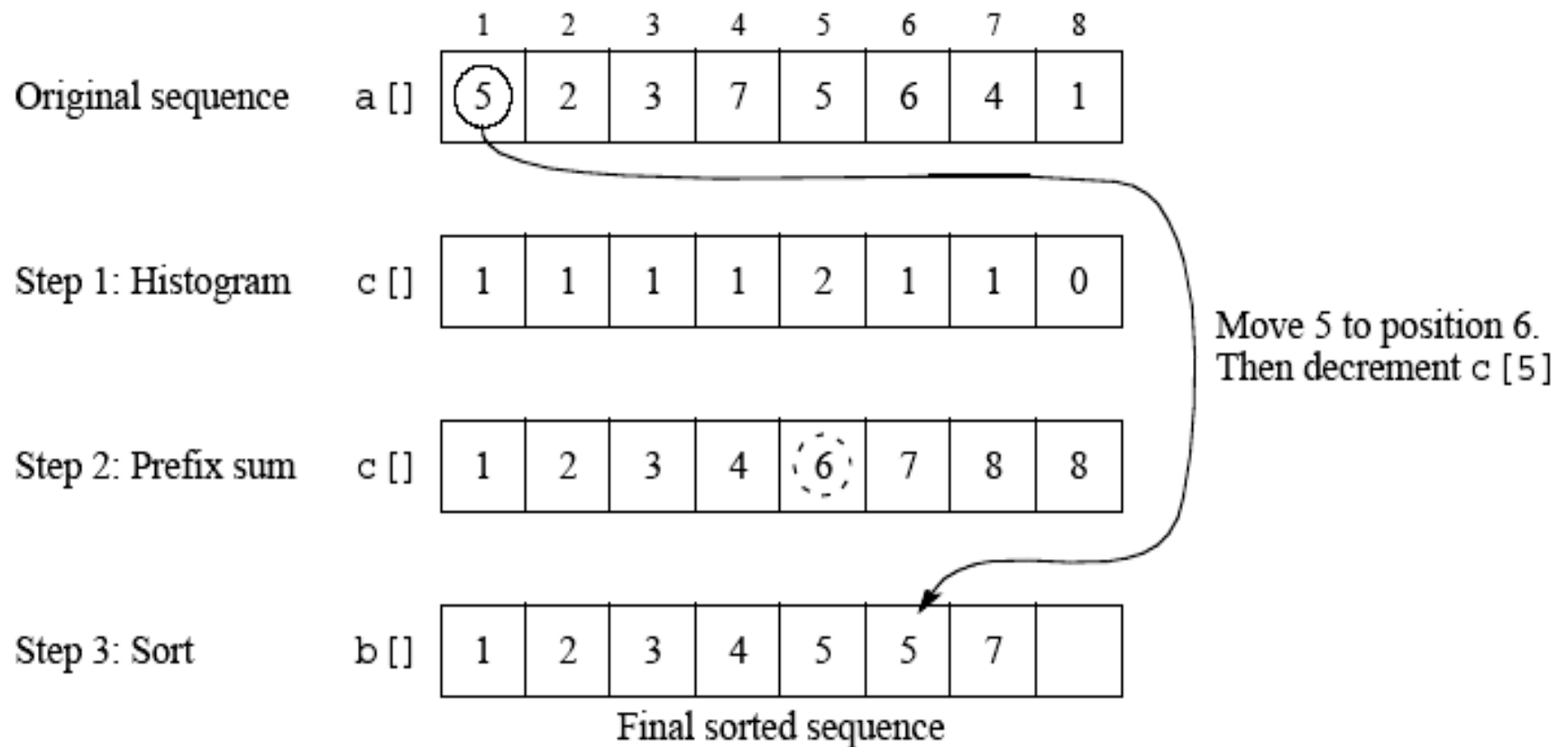
```
for (i = 2; i <= m; i++)  
    c[i] = c[i] + c[i-1];
```

Final stage: The numbers are placed in the sorted order in  $O(n)$  time as described below:

```
for (i = n; i >= 1; i--) {  
    b[c[a[i]]] = a[i]  
    c[a[i]]--;      /* ensures stable sorting */  
}
```

Complete code has  $O(n + m)$  sequential time complexity. If  $m$  is linearly related to  $n$  as it is in some applications, the code has  $O(n)$  sequential time complexity.

# Counting sort





Parallelizing counting sort can use the parallel version of the prefix sum calculation which requires  $O(\log n)$  time with  $n - 1$  processors.

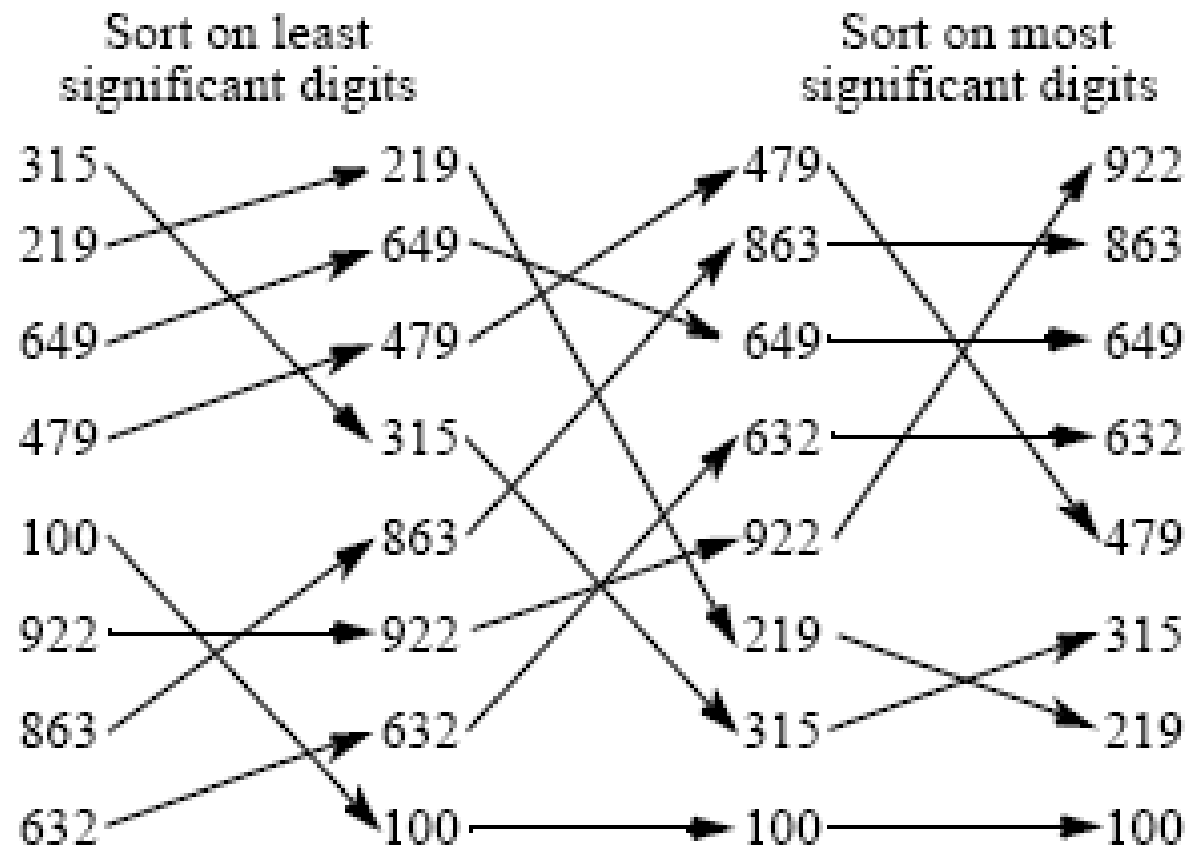
The final sorting stage can be achieved in  $O(n/p)$  time with  $p$  processors or  $O(1)$  with  $n$  processors by simply having the body of the loop done by different processors.

# Radix Sort

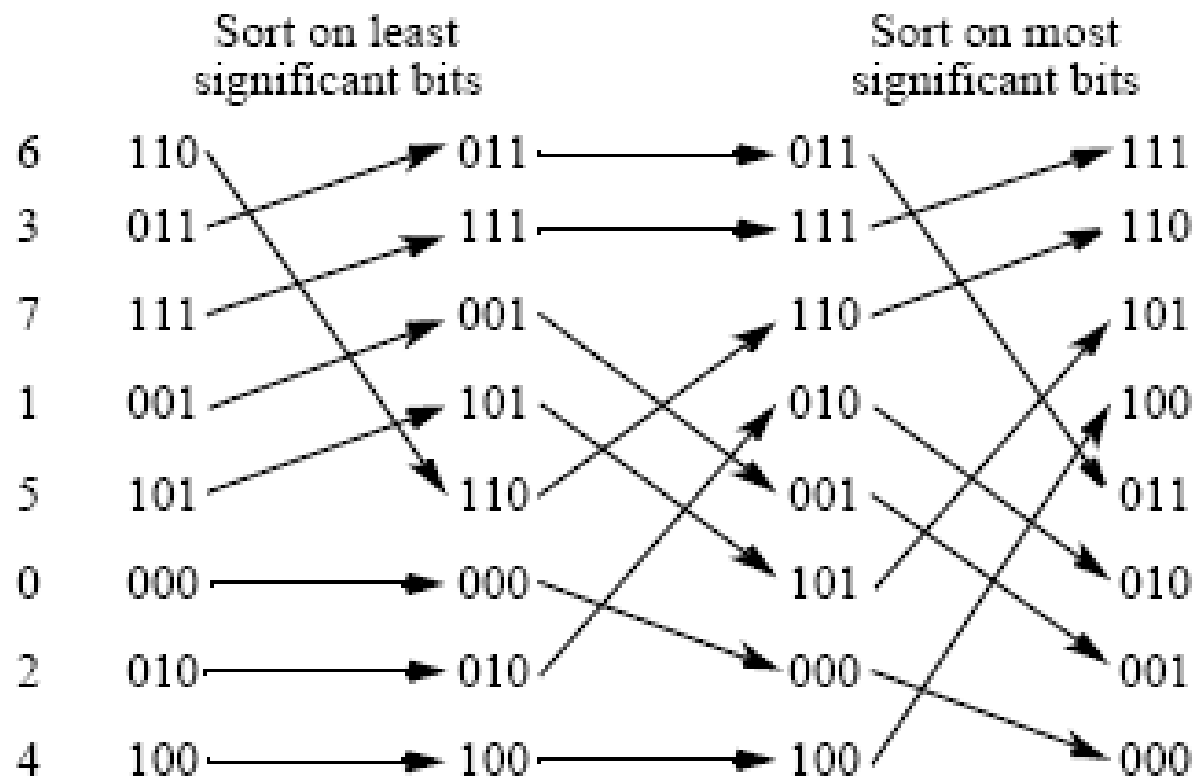
Assumes numbers to sort are represented in a positional digit representation such as binary and decimal numbers. The digits represent values and position of each digit indicates their relative weighting.

Radix sort starts at the least significant digit and sorts the numbers according to their least significant digits. The sequence is then sorted according to the next least significant digit and so on until the most significant digit, after which the sequence is sorted. For this to work, it is necessary that the order of numbers with the same digit is maintained, that is, one must use a stable sorting algorithm.

# Radix sort using decimal digits



# Radix sort using binary digits



Radix sort can be parallelized by using a parallel sorting algorithm in each phase of sorting on bits or groups of bits.

Already mentioned parallelized counting sort using prefix sum calculation, which leads to  $O(\log n)$  time with  $n - 1$  processors and constant  $b$  and  $r$ .

# Using prefix sum calculation to sort digits

When prefix sum calculation applied to a column of bits, it gives number of 1's up to each digit position because all digits can only be 0 or 1 and prefix calculation will simply add number of 1's.

The prefix calculation on the digits inverted (*diminished prefix sum*) give the number of 0's up to each digit.

When digit considered being a 0, diminished prefix sum calculation provides new position for number.

When digit considered being a 1, result of prefix sum calculation plus largest diminished prefix calculation gives final position for number.

# Sample Sort

An old idea (pre1970) as are many basic sorting ideas.

Has been discussed in the context of quicksort and bucket sort.

*In context of quicksort*

Sample sort takes a sample of  $s$  numbers from the sequence of  $n$  numbers.

Median of this sample used as the first pivot to divide sequence into two parts as required as the first step by the quicksort algorithm rather than the usual first number in the list.

## *In context of bucket sort*

Sample sort can divide ranges so that each bucket will have approximately the same number of numbers.

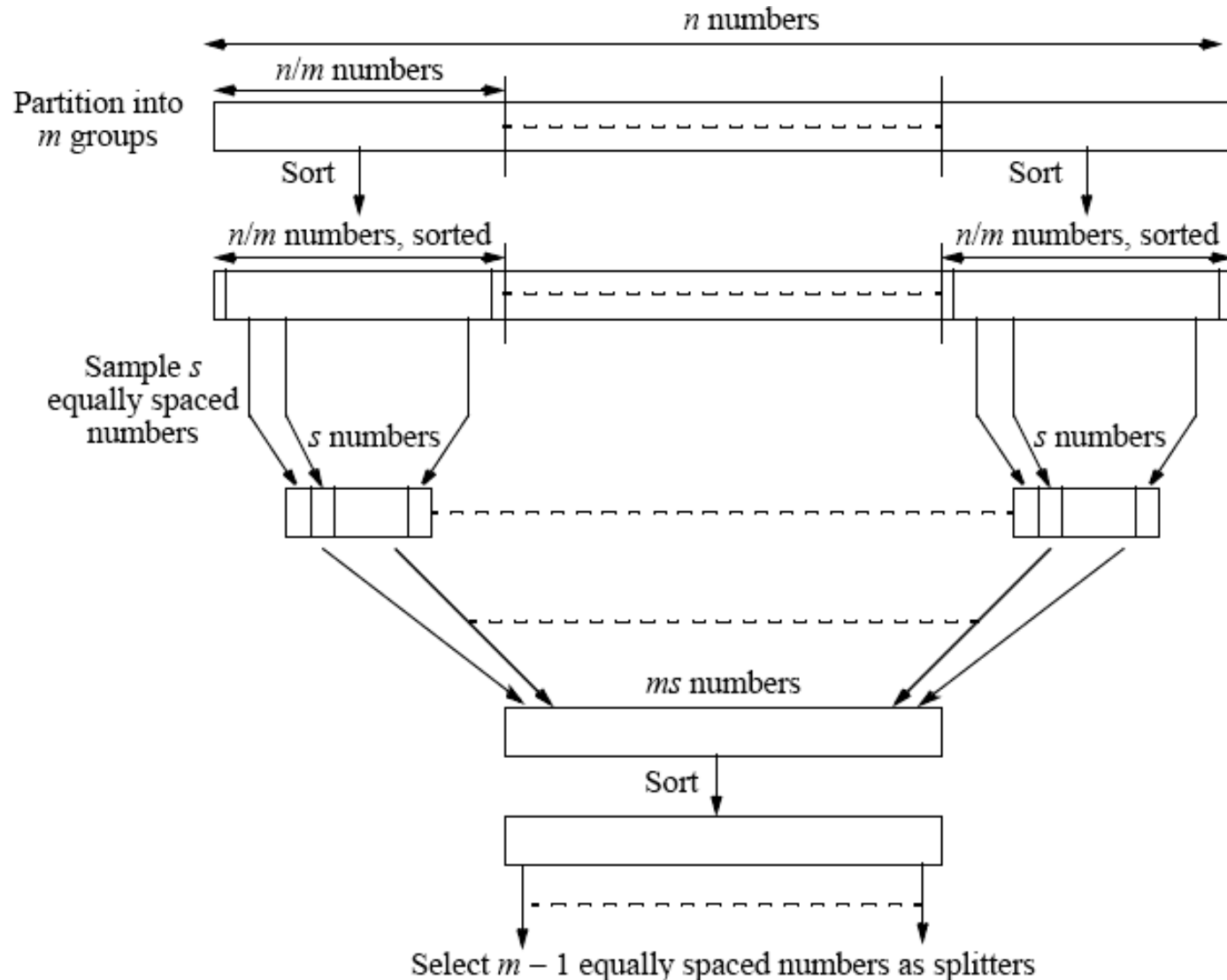
Picks out numbers from sequence of  $n$  numbers as splitters which define the range of numbers for each bucket. If there are  $m$  buckets,  $m - 1$  splitters are needed.

- Numbers are first divided into  $n/m$  groups.
- Each group sorted and a sample of  $s$  equally spaced numbers chosen from each group.
- Creates  $ms$  samples in total which are then sorted and  $m - 1$  equally spaced numbers selected as splitters.



# Selecting splitters

## - Sample sort version of bucket sort



# Sorting Algorithms on Clusters

Factors for efficient implementation on clusters:

**Using collective operations** such broadcast, gather, scatter, and reduce provided in message-passing software such as MPI rather than non-uniform communication patterns that require point-to-point communication, because collective operations expected to be implemented efficiently.

**Using local operation** - Distributed memory of a cluster does not favor algorithms requiring access to widely separately numbers.

Algorithms that require only local operations better, although all sorting algorithms finally have to move numbers in the worst case from one end of sequence to other somehow.

**Cache memory** -- better to have an algorithm that operate upon a block of numbers that can be placed in the cache. Will need to know the size and organization of the cache, and this has to become part of the algorithm as parameters.

**Clusters of shared memory processors** -- algorithms need to take into account that the groups of processors may operate in the shared memory mode. Groups may intercommunicate in a message-passing mode.

To take this into account requires parameters such as number of processors within each shared memory system and size of the memory in each system.

עד כאן מצגת זו

# Algorithms and Applications

Last update: Dec. 2018

ITCS4145/5145, Parallel Programming B. Wilkinson Spring 2009.

slides10-1

Areas done in textbook:

- Sorting Algorithms
- Numerical Algorithms
- Image Processing
- Searching and Optimization

# Sorting Algorithms

- Rearranging a list of numbers into increasing (strictly non-decreasing) order.

## Potential Speedup

$O(n \log n)$  optimal for any sequential sorting algorithm without using special properties of the numbers.

Best we can expect based upon a sequential sorting algorithm using  $n$  processors is

$$\text{Optimal parallel time complexity} = \frac{O(n \log n)}{n} = O(\log n)$$

Has been obtained but the constant hidden in the order notation extremely large.



## General note

Sorting requires moving numbers from one place to another in a list.

The following algorithms concentrate upon a **message –passing model** for moving the numbers.

Sorting number is important in applications as it can make subsequent operations more efficient

## Compare-and-Exchange Sorting Algorithms

### Compare and Exchange

Form the basis of several, if not most, classical sequential sorting algorithms.

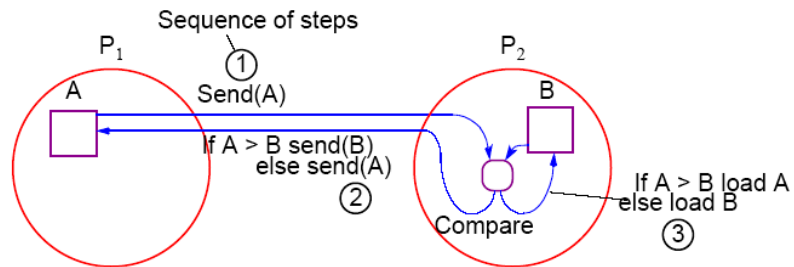
Two numbers, say  $A$  and  $B$ , are compared. If  $A > B$ ,  $A$  and  $B$  are exchanged, i.e.:

```
if (A > B) {  
    temp = A;  
    A = B;  
    B = temp;  
}
```

# Message-Passing Compare and Exchange

## Version 1

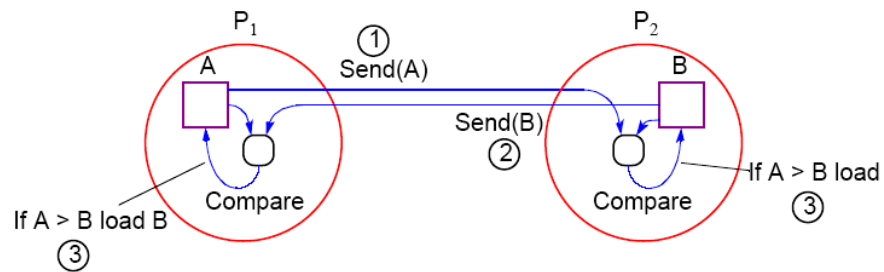
$P_1$  sends  $A$  to  $P_2$ , which compares  $A$  and  $B$  and sends back  $B$  to  $P_1$  if  $A$  is larger than  $B$  (otherwise it sends back  $A$  to  $P_1$ ):



# Alternative Message Passing Method

## Version 2

For  $P_1$  to send  $A$  to  $P_2$  and  $P_2$  to send  $B$  to  $P_1$ . Then both processes perform compare operations.  $P_1$  keeps the larger of  $A$  and  $B$  and  $P_2$  keeps the smaller of  $A$  and  $B$ :



## Note on Precision of Duplicated Computations

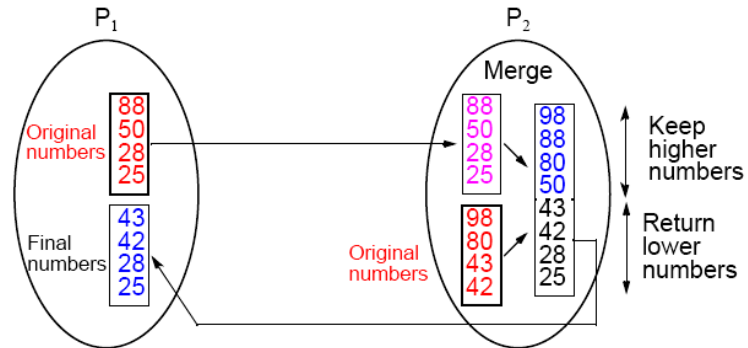
Previous code assumes that the **if** condition, **A > B**, will return the same Boolean answer in both processors.

Different processors operating at different precision could conceivably produce **different answers** if real numbers are being compared.

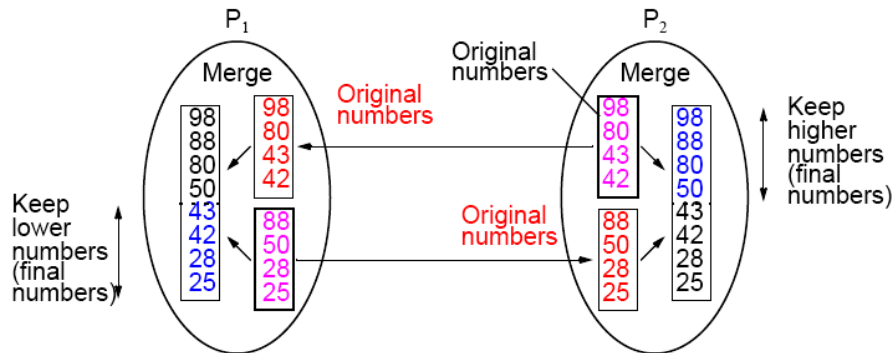
This situation applies to anywhere computations are duplicated in different processors to reduce message passing, or to make the code SPMD.

# Data Partitioning (Version 1)

$p$  processors and  $n$  numbers.  $n/p$  numbers assigned to each processor:



## Merging Two Sublists — Version 2



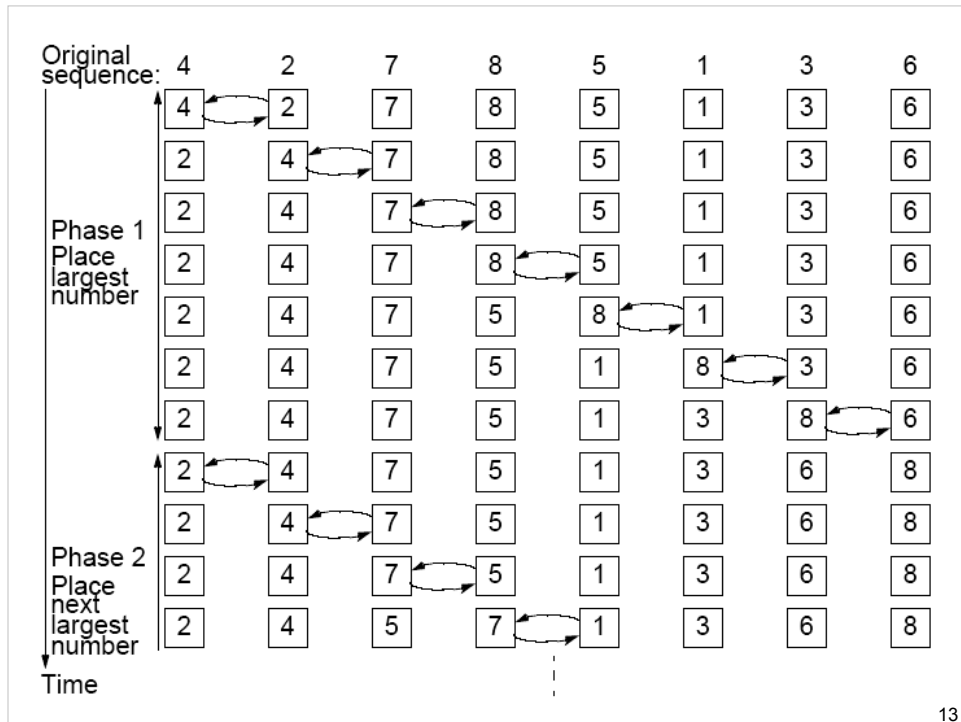
## **Bubble Sort**

First, largest number moved to the end of list by a series of compares and exchanges, starting at the opposite end.

Actions repeated with subsequent numbers, stopping just before the previously positioned number.

In this way, the larger numbers move (“bubble”) toward one end,





# Time Complexity

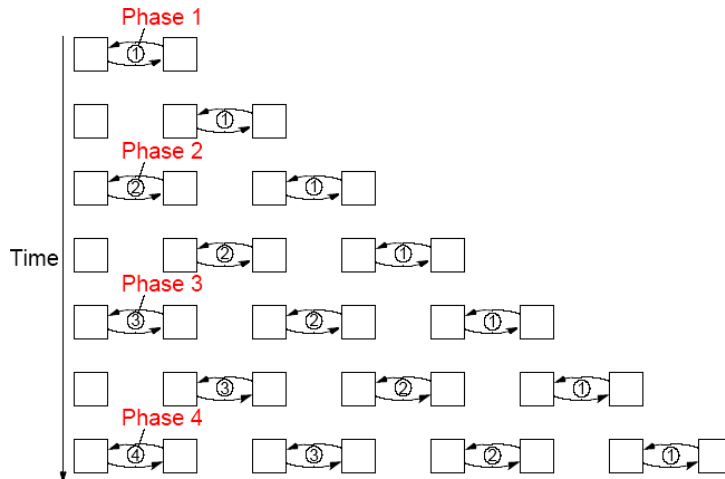
Number of compare and exchange operations

$$= \sum_{i=1}^{n-1} i = \frac{n(n-1)}{2}$$

Indicates a time complexity of  $O(n^2)$  given that a single compare-and-exchange operation has a constant complexity,  $O(1)$ .

# Parallel Bubble Sort

Iteration could start before previous iteration finished if does not overtake previous bubbling action:



# Odd-Even (Transposition) Sort

Variation of bubble sort.

Operates in two alternating phases, *even* phase and *odd* phase.

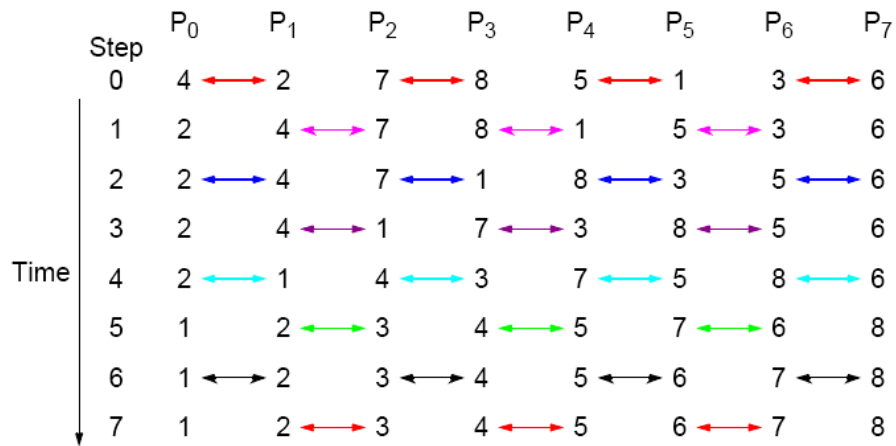
## **Even phase**

Even-numbered processes exchange numbers with their right neighbor.

## **Odd phase**

Odd-numbered processes exchange numbers with their right neighbor.

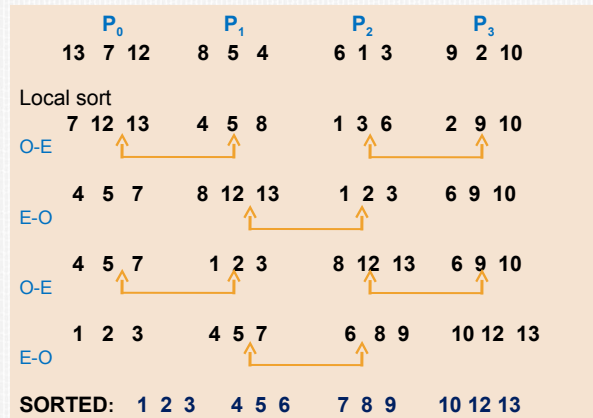
## Odd-Even Transposition Sort Sorting eight numbers



## Odd-Even Transposition Sort – Example ( $N \gg p$ )

$p=4$

Each PE gets  $n/p$  numbers. First, PEs sort  $n/p$  locally, then they run odd-even trans. algorithm each time doing a merge-split for  $2n/p$  numbers.



**Time complexity:**  $T_{\text{par}} = (\text{Local Sort}) + (p \text{ merge-splits}) + (p \text{ exchanges})$

$$T_{\text{par}} = (n/p)\log(n/p) + p^*(n/p) + p^*(n/p) = (n/p)\log(n/p) + 2n$$

# Mergesort

A classical sequential sorting algorithm using divide-and-conquer approach. Unsorted list first divided into half. Each half is again divided into two. Continued until individual numbers are obtained.

Then pairs of numbers combined (merged) into sorted list of two numbers. Pairs of these lists of four numbers are merged into sorted lists of eight numbers. This is continued until the one fully sorted list is obtained.

# John von Neumann



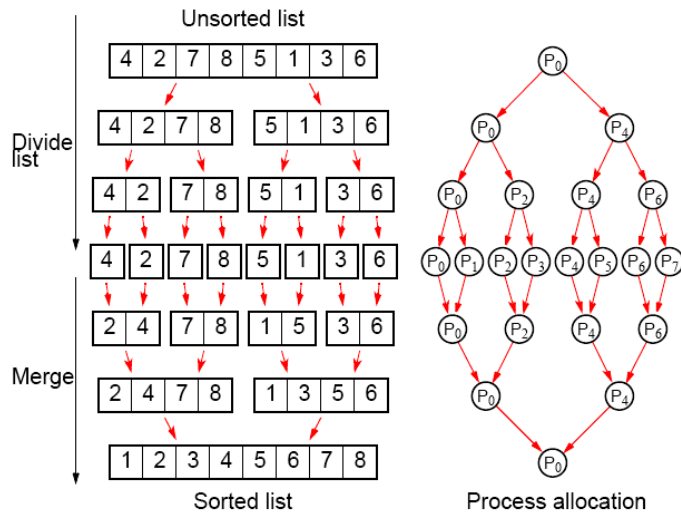


# Animated merge sort

6 5 3 1 8 7 2 4

# Parallelizing Mergesort

Using tree allocation of processes



# **Analysis**

## **Sequential**

Sequential time complexity is  $O(n \log n)$ .

## **Parallel**

$2 \log n$  steps in the parallel version but each step may need to perform more than one basic operation, depending upon the number of numbers being processed - see text.

# Quicksort

Very popular **sequential** sorting algorithm that performs well with average sequential time complexity of  $O(n \log n)$ .

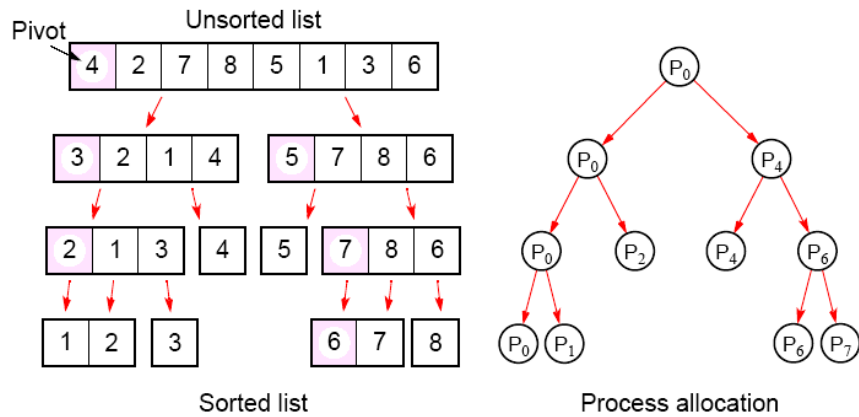
First list divided into two sublists. All numbers in one sublist arranged to be smaller than all numbers in other sublist.

Achieved by first selecting one number, called a *pivot*, against which every other number is compared. If the number is less than the pivot, it is placed in one sublist. Otherwise, it is placed in the other sublist.

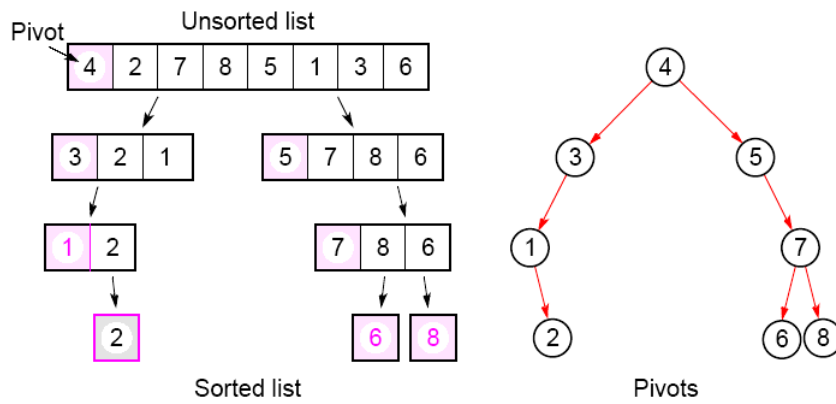
Pivot could be any number in the list, but often first number in list chosen. Pivot itself could be placed in one sublist, or the pivot could be separated and placed in its final position.

# Parallelizing Quicksort

Using tree allocation of processes



With the pivot being withheld in processes:



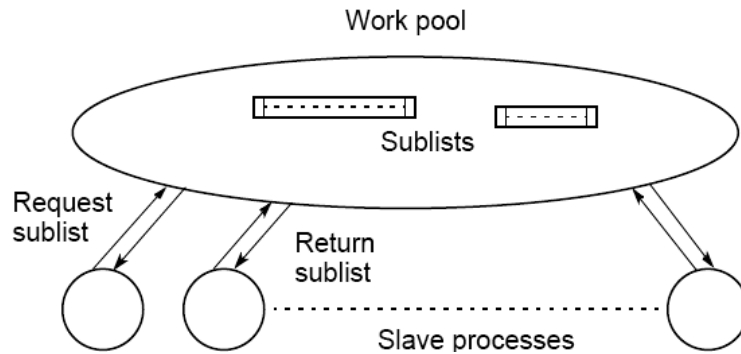
## **Analysis**

Fundamental problem with all tree constructions – initial division done by a single processor, which will seriously limit speed.

Tree in quicksort will not, in general, be perfectly balanced  
Pivot selection very important to make quicksort operate fast.

## Work Pool Implementation of Quicksort

First, work pool holds initial unsorted list. Given to first processor which divides list into two parts. One part returned to work pool to be given to another processor, while the other part operated upon again.





Neither Mergesort nor Quicksort parallelize very well as the processor efficiency is low (see book for analysis).

Quicksort also can be very unbalanced. Can use load balancing techniques.

# Batcher's Parallel Sorting Algorithms

- Odd-even Mergesort
- Bitonic Mergesort

Originally derived in terms of switching networks.

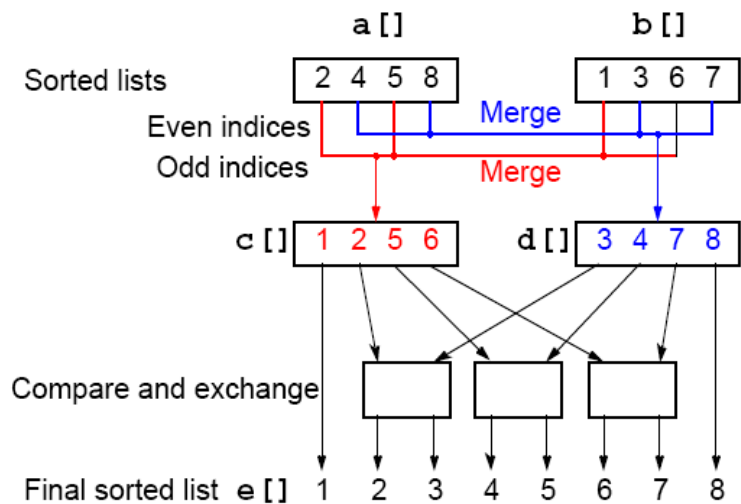
Both are well balanced and have parallel time complexity of  $O(\log^2 n)$  with  $n$  processors.

# Odd-Even Mergesort

## Odd-Even Merge Algorithm

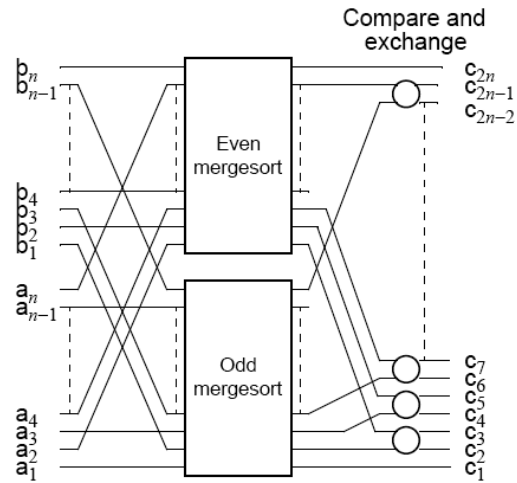
Start with odd-even merge algorithm which will merge two *sorted* lists into one sorted list. Given two **sorted** lists  $a_1, a_2, a_3, \dots, a_n$  and  $b_1, b_2, b_3, \dots, b_n$  (where  $n$  is a power of 2).

# Odd-Even Merging of Two Sorted Lists



# Odd-Even Mergesort

Apply odd-even merging recursively



# Bitonic Mergesort

## Bitonic Sequence

A *monotonic* increasing sequence is a sequence of increasing numbers.

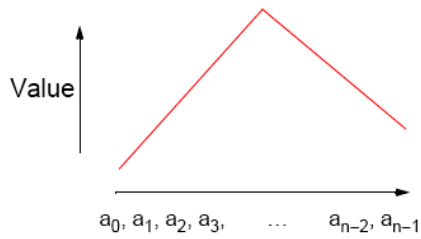
A *bitonic sequence* has two sequences, one increasing and one decreasing. e.g.

$$a_0 < a_1 < a_2, a_3, \dots, a_{i-1} < a_i > a_{i+1}, \dots, a_{n-2} > a_{n-1}$$

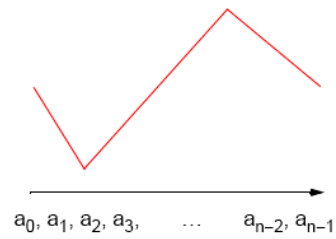
for some value of  $i$  ( $0 \leq i < n$ ).

A sequence is also bitonic if the preceding can be achieved by shifting the numbers cyclically (left or right).

# Bitonic Sequences



(a) Single maximum



(b) Single maximum and single minimum

## “Special” Characteristic of Bitonic Sequences

If we perform a compare-and-exchange operation on  $a_i$  with  $a_{i+n/2}$  for all  $i$ , where there are  $n$  numbers in the sequence, get **TWO** bitonic sequences, where the numbers in one sequence are **all less than the numbers in the other sequence**.

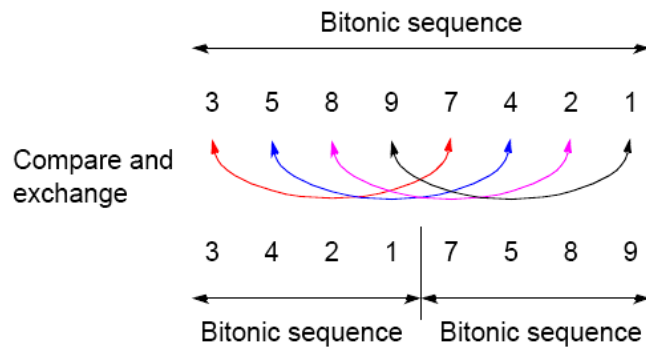


## Creating two bitonic sequences from one bitonic sequence

Starting with the bitonic sequence

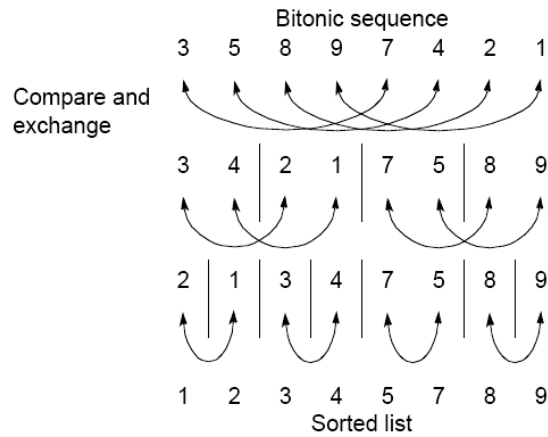
3, 5, 8, 9, 7, 4, 2, 1

we get:



# Sorting a bitonic sequence

Compare-and-exchange moves smaller numbers of each pair to left and larger numbers of pair to right. **Given a bitonic sequence**, recursively performing operations will sort the list.



## Sorting

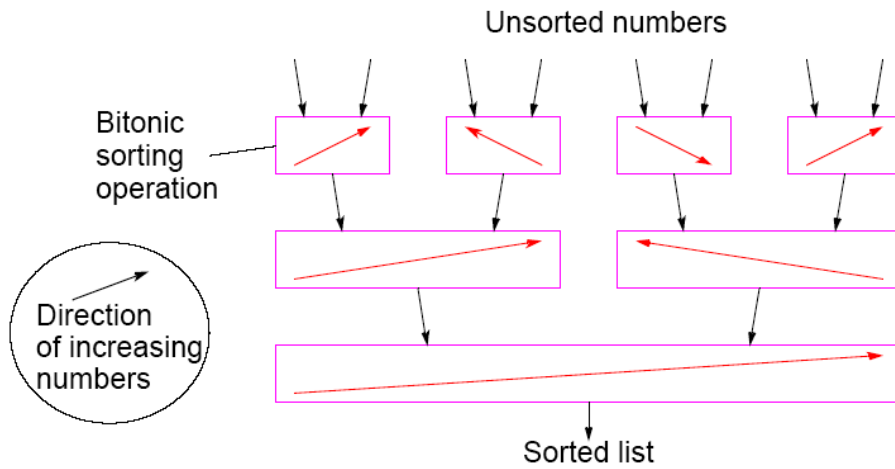
To sort an **unordered sequence**, sequences are merged into larger bitonic sequences, starting with pairs of adjacent numbers.

By a compare-and-exchange operation, pairs of adjacent numbers formed into increasing sequences and decreasing sequences. Pairs form a bitonic sequence of twice size of each original sequences.

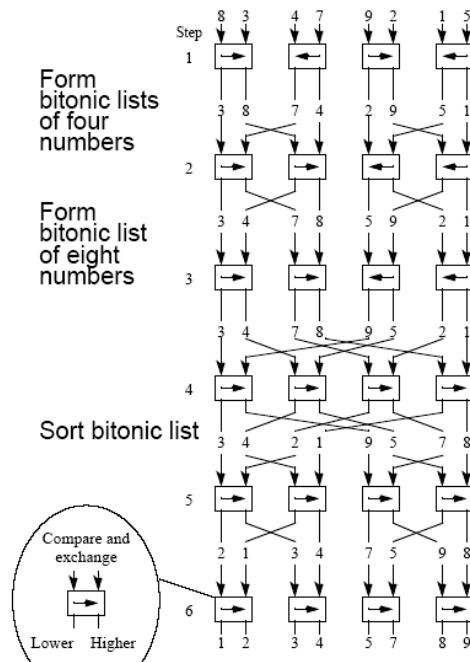
By repeating this process, bitonic sequences of larger and larger lengths obtained.

In the final step, a single bitonic sequence sorted into a single increasing sequence.

# Bitonic Mergesort



# Bitonic Mergesort on Eight Numbers



# Phases

The six steps (for eight numbers) are divided into three phases:

Phase 1 (Step 1)     Convert pairs of numbers into increasing/  
decreasing sequences and into 4-bit  
bitonic sequences.

Phase 2 (Steps 2/3) Split each 4-bit bitonic sequence into two  
2-bit bitonic sequences, higher  
sequences at center.

increasing/  
bit     Sort each 4-bit bitonic sequence  
decreasing sequences and merge into 8-  
bitonic sequence.

Phase 3 (Steps 4/5/6) Sort 8-bit bitonic sequence.

## Number of Steps

In general, with  $n = 2^k$ , there are  $k$  phases, each of 1, 2, 3, ...,  $k$  steps. Hence the total number of steps is given by

$$\text{Steps} = \sum_{i=1}^k i = \frac{k(k+1)}{2} = \frac{\log n (\log n + 1)}{2} = O(\log^2 n)$$

## Sorting Conclusions so far

Computational time complexity using  $n$  processors

- Odd-even transposition sort -  $O(n)$
- Parallel mergesort -  $O(n)$  but unbalanced processor load and Communication
- Parallel quicksort -  $O(n)$  but unbalanced processor load, and communication can generate to  $O(n^2)$
- Odd-even Mergesort and Bitonic Mergesort  $O(\log^2 n)$   
**Bitonic mergesort has been a popular choice for a parallel sorting.**



## Sorting on Specific Networks

Algorithms can take advantage of the underlying interconnection network of the parallel computer.

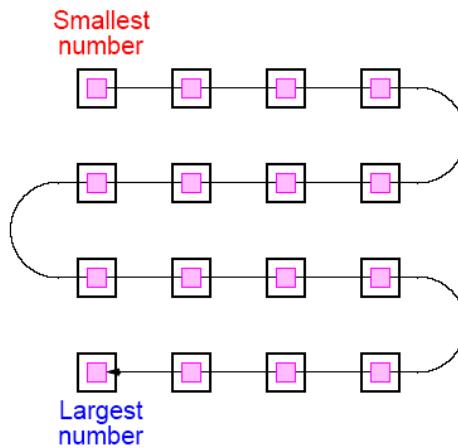
Two network structures have received specific attention: the **mesh** and **hypercube** because parallel computers have been built with these networks.

Of less interest nowadays because underlying architecture often hidden from user.

MPI features for mapping algorithms onto meshes, and one can always use a mesh or hypercube algorithm even if the underlying architecture is not the same.

## Mesh - Two-Dimensional Sorting

The layout of a sorted sequence on a mesh could be row by row or *snakelike*. Snakelike:



# Shearsort

Alternate row and column sorting until list fully sorted. Row sorting alternative directions to get snake-like sorting:

4	14	8	2
10	3	13	16
7	15	1	5
12	6	11	9

(a) Original placement of numbers

2	4	8	14
16	13	10	3
1	5	7	15
12	11	9	6

(b) Phase 1 — Row sort

1	4	7	3
2	5	8	6
12	11	9	14
16	13	10	15

(c) Phase 2 — Column sort

1	3	4	7
8	6	5	2
9	11	12	14
16	15	13	10

(d) Phase 3 — Row sort

1	3	4	2
8	6	5	7
9	11	12	10
16	15	13	14

(e) Phase 4 — Column sort

1	2	3	4
8	7	6	5
9	10	11	12
16	15	14	13

(f) Final phase — Row sort

## Shearsort

Requires  $\sqrt{n}(\log n + 1)$  steps for  $n$  numbers on a  $\sqrt{n} \times \sqrt{n}$  mesh.

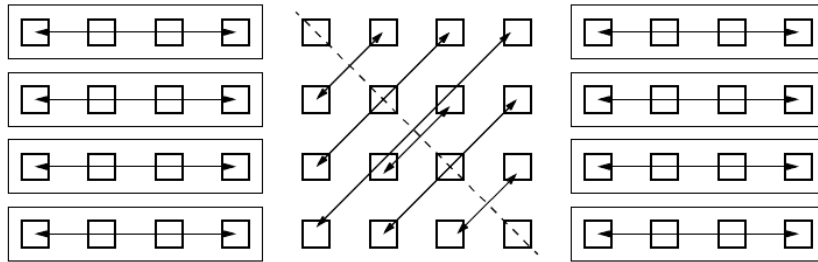
הביטוי הזה של הספר לא עומד במבחן המציאות. ניקח  
את הדוגמה מהשקף הקודם.  $N=16$

$$4 \cdot (4+1) = 20$$

אנחנו ראים שצריך רק 5 שלבים  
אולי הביטוי הנכון הוא  
 $\log(n)+1=5$

# Using Transposition

Causes the elements in each column to be in positions in a row.  
Can be placed between the row operations and column operations



(a) Operations between elements in rows

(b) Transpose operation

(c) Operations between elements in rows (originally columns)

Transposition can be achieved with  $\sqrt{n}(\sqrt{n} - 1)$  communications ( $O(n)$ ). An *all-to-all* routine could be reduce this.

## Other Sorting Algorithms

We began by giving the lower bound for the time complexity of a sequential sorting algorithm based upon comparisons as  $O(n \log n)$ .

Consequently, the time complexity of a parallel sorting algorithm based upon comparisons is  $O((\log n)/p)$  with  $p$  processors or  $O(\log n)$  with  $n$  processors.

There are sorting algorithms that can achieve better than  $O(n \log n)$  sequential time complexity and are very attractive candidates for parallelization but they often assume special properties of the numbers being sorted.

## Rank Sort as basis of a parallel sorting algorithm

Does not achieve a sequential time of  $O(n \log n)$ , but can be parallelized easily, and leads us onto linear sequential time algorithms which can be parallelized to achieve  $O(\log n)$  parallel time and are attractive algorithms for clusters.

## Rank Sort

Number of numbers that are smaller than each selected number counted. This count provides the position of selected number in sorted list; that is, its “rank.”

- First **a[0]** is read and compared with each of the other numbers, **a[1] ... a[n-1]**, recording the number of numbers less than **a[0]**.
- Suppose this number is  $x$ . This is the index of the location in the final sorted list.
- The number **a[0]** is copied into the final sorted list **b[0] ... b[n-1]**, at location **b[x]**. Actions repeated with the other numbers.

Overall sequential sorting time complexity of  $O(n^2)$  (not exactly a good sequential sorting algorithm!).



## Sequential Code

```
for (i = 0; i < n; i++) {           /* for each number */
    x = 0;
    for (j = 0; j < n; j++) /* count number less than it */
        if (a[i] > a[j]) x++;
    b[x] = a[i]; /* copy number into correct place */
}
```

This code will fail if duplicates exist in the sequence of numbers. Easy to fix. (How?)

## Parallel Code Using $n$ Processors

One processor allocated to each number. Finds final index in  $O(n)$  steps. With all processors operating in parallel, parallel time complexity  $O(n)$ .

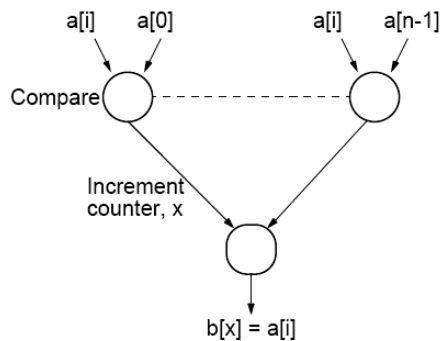
In **forall** notation, the code would look like

```
forall (i = 0; i < n; i++) {      /* for each no in parallel*/
    x = 0;
    for (j = 0; j < n; j++) /* count number less than it */
        if (a[i] > a[j]) x++;
    b[x] = a[i];                /* copy no into correct place */
}
```

Parallel time complexity,  $O(n)$ , as good as any sorting algorithm so far. Can do even better if we have more processors.

## Using $n^2$ Processors

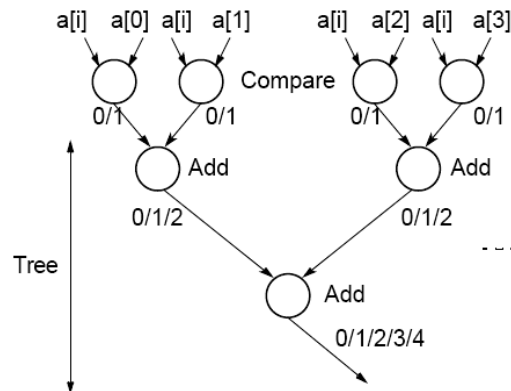
Comparing one number with the other numbers in list using multiple processors:



$n - 1$  processors used to find rank of one number. With  $n$  numbers,  $(n - 1)n$  processors or (almost)  $n^2$  processors needed. Incrementing counter done sequentially and requires maximum of  $n$  steps. Total number of steps =  $1 + n$ .

## Reduction in Number of Steps

Tree to reduce number of steps involved in incrementing counter:



$O(\log n)$  algorithm with  $n^2$  processors.  
Processor efficiency relatively low.

## Parallel Rank Sort Conclusions

Easy to do as each number can be considered in isolation.

Rank sort can sort in:

$O(n)$  with  $n$  processors

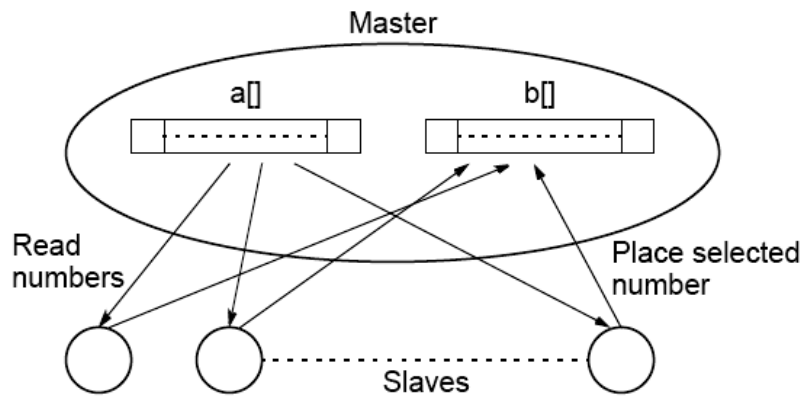
or

$O(\log n)$  using  $n^2$  processors.

In practical applications, using  $n^2$  processors prohibitive.

Theoretically possible to reduce time complexity to  $O(1)$  by considering all increment operations as happening in parallel since they are independent of each other.

## Message Passing Parallel Rank Sort Master-Slave Approach



Requires shared access to list of numbers. Master process responds to request for numbers from slaves. Algorithm better for shared memory

## Counting Sort

If the numbers to be sorted are integers, there is a way of coding the rank sort algorithm to reduce the sequential time complexity from  $O(n^2)$  to  $O(n)$ , called as *Counting Sort*.

As in the rank sort code suppose the unsorted numbers stored in an array `a[]` and final sorted sequence is stored in array `b[]`. Algorithm uses an additional array, say `c[]`, having one element for each possible value of the numbers. Suppose the range of integers is from 1 to  $m$ . The array has element `c[1]` through `c[m]` inclusive. Now, let us working through the algorithm in stages.

## Stable Sort Algorithms

Algorithms that will place identical numbers in the same order as in the original sequence.

Counting sort is naturally a **stable sorting algorithm**.



First, **c[ ]** will be used to hold the histogram of the sequence, that is, the number of each number. This can be computed in  $O(m)$  time with code such as:

```
for (i = 1; i <= m; i++)  
    c[i] = 0;  
for (i = 1; i <= m; i++)  
    c[a[i]]++;
```

Next stage: The number of numbers less than each number found by performing a prefix sum operation on array **c[ ]**.

In the prefix sum calculation, given a list of numbers,  $x_0, \dots, x_{n-1}$ , all the partial summations (i.e.,  $x_0$ ;  $x_0 + x_1$ ;  $x_0 + x_1 + x_2$ ;  $x_0 + x_1 + x_2 + x_3$ ; ... ) are computed.

Here, the prefix sum is computed using the histogram originally held in **c[ ]** in  $O(m)$  time as described below:

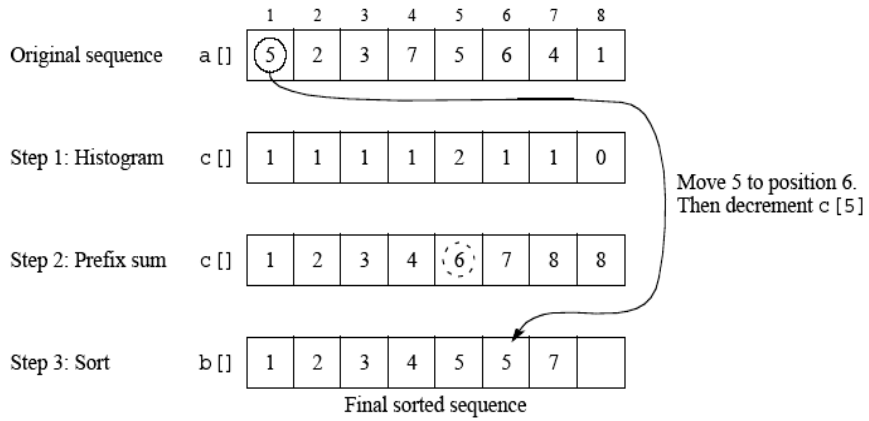
```
for (i = 2; i <= m; i++)  
    c[i] = c[i] + c[i-1];
```

Final stage: The numbers are placed in the sorted order in  $O(n)$  time as described below:

```
for (i = n; i >= 1; i--) {  
    b[c[a[i]]] = a[i]  
    c[a[i]]--;    /* ensures stable sorting */  
}
```

Complete code has  $O(n + m)$  sequential time complexity. If  $m$  is linearly related to  $n$  as it is in some applications, the code has  $O(n)$  sequential time complexity.

# Counting sort



Parallelizing counting sort can use the parallel version of the prefix sum calculation which requires  $O(\log n)$  time with  $n - 1$  processors.

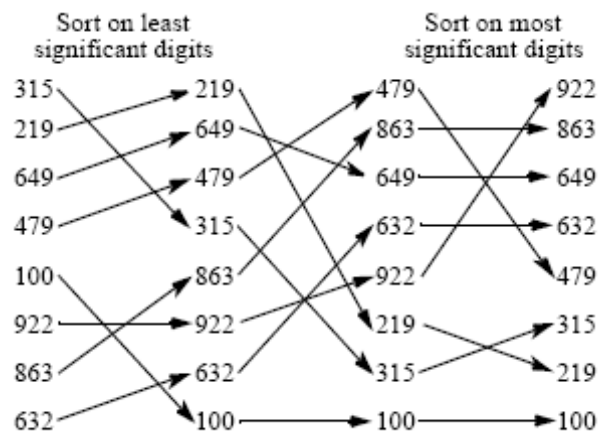
The final sorting stage can be achieved in  $O(n/p)$  time with  $p$  processors or  $O(1)$  with  $n$  processors by simply having the body of the loop done by different processors.

## Radix Sort

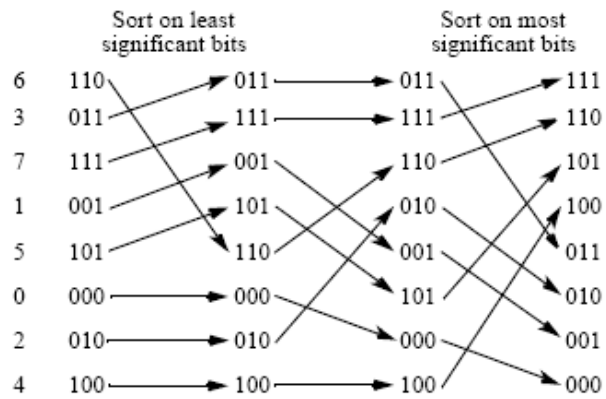
Assumes numbers to sort are represented in a positional digit representation such as binary and decimal numbers. The digits represent values and position of each digit indicates their relative weighting.

Radix sort starts at the least significant digit and sorts the numbers according to their least significant digits. The sequence is then sorted according to the next least significant digit and so on until the most significant digit, after which the sequence is sorted. For this to work, it is necessary that the order of numbers with the same digit is maintained, that is, one must use a stable sorting algorithm.

## Radix sort using decimal digits



## Radix sort using binary digits





Radix sort can be parallelized by using a parallel sorting algorithm in each phase of sorting on bits or groups of bits.

Already mentioned parallelized counting sort using prefix sum calculation, which leads to  $O(\log n)$  time with  $n - 1$  processors and constant  $b$  and  $r$ .

## Using prefix sum calculation to sort digits

When prefix sum calculation applied to a column of bits, it gives number of 1's up to each digit position because all digits can only be 0 or 1 and prefix calculation will simply add number of 1's.

The prefix calculation on the digits inverted (*diminished prefix sum*) give the number of 0's up to each digit.

When digit considered being a 0, diminished prefix sum calculation provides new position for number.

When digit considered being a 1, result of prefix sum calculation plus largest diminished prefix calculation gives final position for number.

## Sample Sort

An old idea (pre1970) as are many basic sorting ideas.

Has been discussed in the context of quicksort and bucket sort.

### *In context of quicksort*

Sample sort takes a sample of  $s$  numbers from the sequence of  $n$  numbers.

Median of this sample used as the first pivot to divide sequence into two parts as required as the first step by the quicksort algorithm rather than the usual first number in the list.

### *In context of bucket sort*

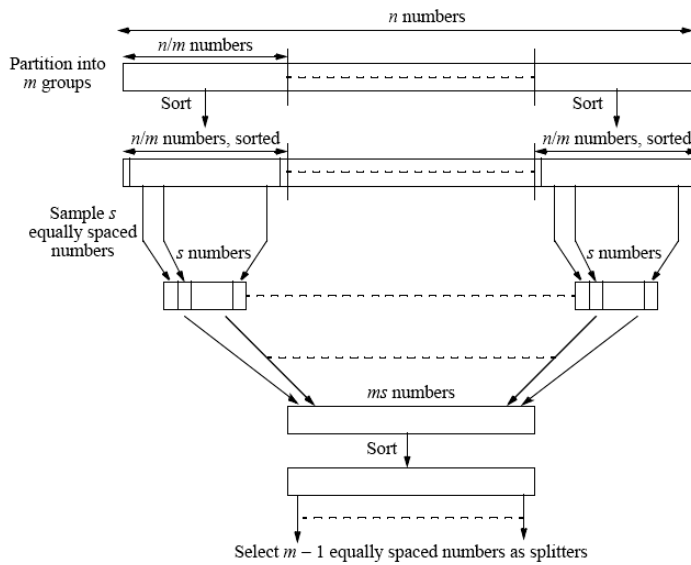
Sample sort can divide ranges so that each bucket will have approximately the same number of numbers.

Picks out numbers from sequence of  $n$  numbers as splitters which define the range of numbers for each bucket. If there are  $m$  buckets,  $m - 1$  splitters are needed.

- Numbers are first divided into  $n/m$  groups.
- Each group sorted and a sample of  $s$  equally spaced numbers chosen from each group.
- Creates  $ms$  samples in total which are then sorted and  $m - 1$  equally spaced numbers selected as splitters.

# Selecting splitters

## - Sample sort version of bucket sort



# Sorting Algorithms on Clusters

Factors for efficient implementation on clusters:

**Using collective operations** such broadcast, gather, scatter, and reduce provided in message-passing software such as MPI rather than non-uniform communication patterns that require point-to-point communication, because collective operations expected to be implemented efficiently.

**Using local operation** - Distributed memory of a cluster does not favor algorithms requiring access to widely separately numbers.

Algorithms that require only local operations better, although all sorting algorithms finally have to move numbers in the worst case from one end of sequence to other somehow.

**Cache memory** -- better to have an algorithm that operate upon a block of numbers that can be placed in the cache. Will need to know the size and organization of the cache, and this has to become part of the algorithm as parameters.

**Clusters of shared memory processors** -- algorithms need to take into account that the groups of processors may operate in the shared memory mode. Groups may intercommunicate in a message-passing mode.

To take this into account requires parameters such as number of processors within each shared memory system and size of the memory in each system.

עד כאן מצגת זו