# Parallel Sorting Algorithms

(cf. Grama et al.)

## Sorting: Overview

- Definitely one of the most important and used topics in computing
- Sorting can be **comparison-based** or **non-comparison-based**
- The fundamental operation of comparison-based sorting is the **exchange-comparison** (compare-exchange)
- The **lower limit** of any sequential sorting algorithm for **n** numbers is $\Theta(n \log n)$
- In the parallel computing field, the **most** we can expect with n processors is $O(\log n)$! In fact it is obtained, but with very large constants ....

# Example: Sorting of n numbers

**Bubble Sort**

**Input:** sequence of n numbers $<a_1, a_2, a_3, \dots a_n>$

**Output:** permutation $<a'_1, a'_2, a'_3, \dots a'_n>$ of the elements so that $a'_1 \leq a'_2, \leq a'_3 \leq \dots \leq a'_n$

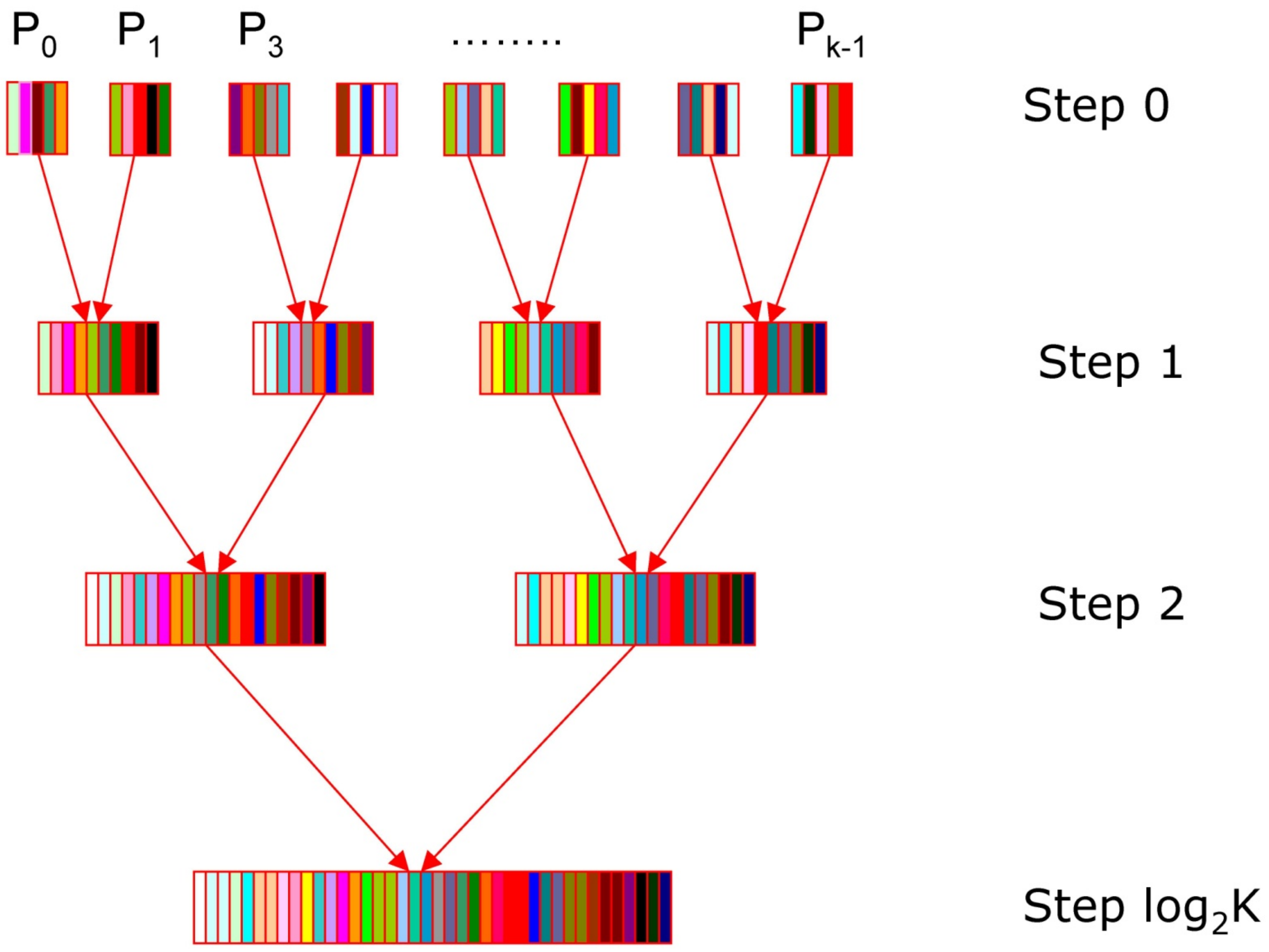
```
/* Bubble sort for integers */
#define SWAP(a,b) { int t; t=a; a=b; b=t; }
void SORT( int a[], int n )
/* Pre-condition: a contains n items to be sorted */
    {
    int i, j;
    /* Make n passes through the array */
    for(i=0;i<n;i++)
        {
        /* From the first element to the end of the unsorted section */
        for(j=1;j<(n-i);j++)
            {
            /* If adjacent items are out of order, swap them */
            if( a[j-1]>a[j] ) SWAP(a[j-1],a[j]);
            }
        }
    }
```

# Example: Parallel Sorting

Idea !

```
SORT(a[0 : n/2-1])
SORT(a[n/2 : n-1])
MERGE(a[0 : n/2-1], a[n/2 : n-1])
```

# Example: Parallel Sorting



# Formally

- Let $S = \langle a_1, a_2, \ldots, a_n \rangle$ be a sequence $n$ elements
- **The ordering of** S consists in a sequence $S' = \langle a'_1, a'_2, \ldots, a'_n \rangle$ such that $a'_i \leq a'_j$ for $1 \leq i \leq j \leq n$ and $S'$ is a permutation of $S$

## Compare exchange

- Base of each **comparison based** algorithm

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

**compare-exchange operation**

# Sorting: Basics

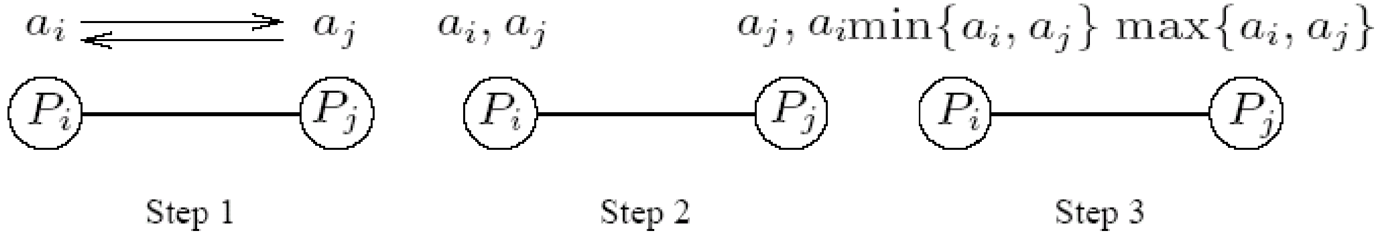
What is a **ordered parallel sequence**? **Where** are the input and output lists stored?

The main assumption is that the input and output lists are **distributed**

- By convention, the sorted list is **partitioned** with the property that each **sub-list is sorted** and **each element** of the list of processor $P_i$ is smaller than those in $P_j$ if $i < j$
- **The previous fact is at the basis of every parallel sorting algorithm!**

# Sorting: Parallel Compare Exchange Operation

In **sequential sorting** algorithms, the input and any ordered sequences (including the output) are stored in the memory of the **same process**. In the parallel case, the data can, of course, be stored in other processes, making the algorithm further complicated!



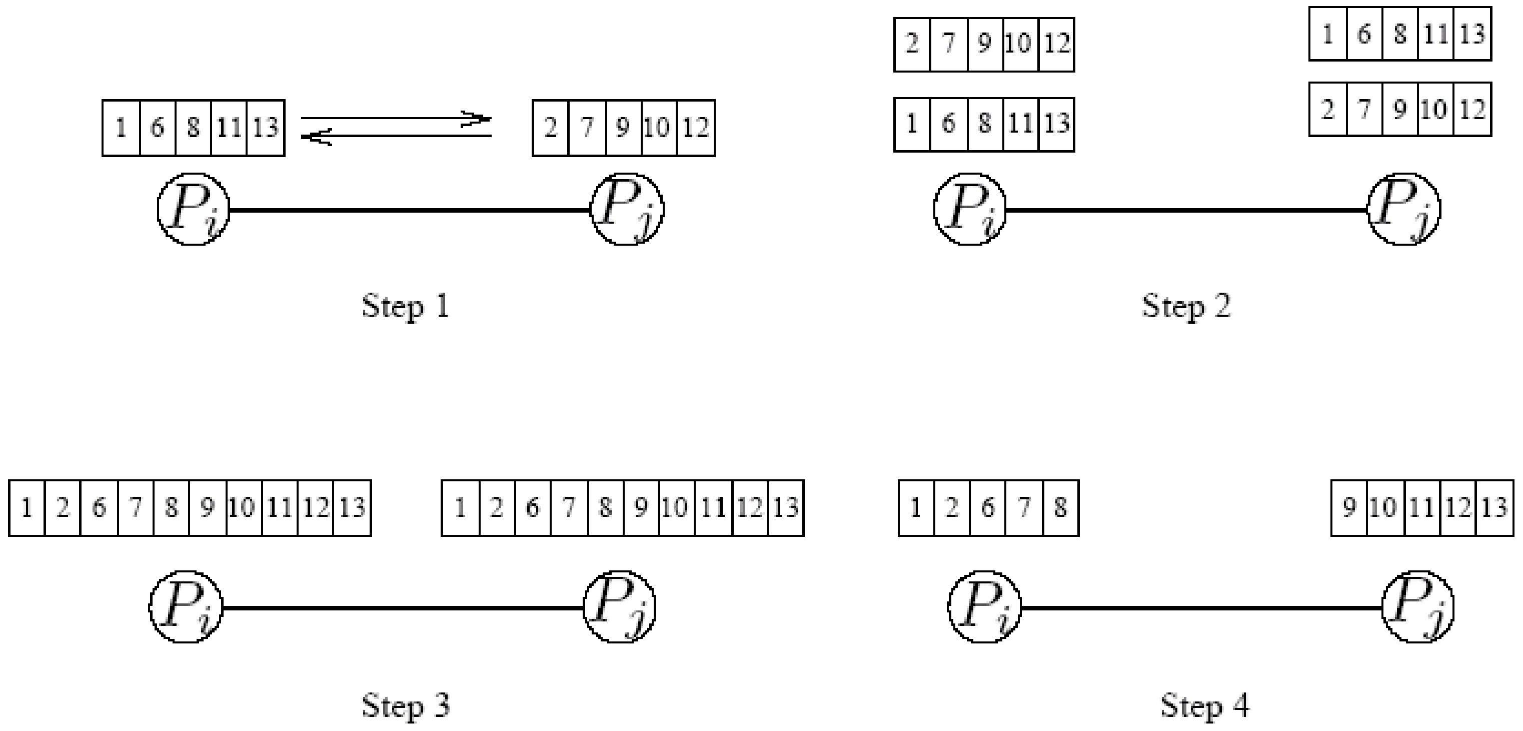
A compare-exchange operation (one element per processor). The processes *Pi* and *Pj* send their elements to each other. The process *Pi* keeps $\min\{a_i,a_j\}$, and *Pj* keeps $\max\{a_i,a_j\}$

# Sorting: Basics

What is the counterpart of a parallel sequence comparison?

- If each processor has an element, the **compare-exchange** operation stores the smallest element in the processor with **smaller id**. This is done in time $t_s + t_w$
- If you have more than one element per processor, this operation is called **compare split** . Suppose that each of the two processors have n / p elements.
- **After the compare-split operation, the n/p smallest elements are in the processor *Pi* while the greater n / p are in *Pj*, where i <j**
- The time for a compare-split operation is $(t_s + t_w n/p)$, assuming that the two "partial" lists were initially ordered

# Sorting: Compare Split Parallel Operation



A compare-split operation. Each process sends its block of size n / p to another process. Each process "merges" the block that receives with its own and maintains only the appropriate half. In this example, the process $P_i$ maintains the minor elements and the process $P_j$ maintains the bigger.

## Sorting Networks

- They are "**comparator**" networks, designed for sorting
- A **comparator is a device** with two inputs x and y and two outputs x 'and y'. For an increasing comparator, we have x '= min {x, y} and y' = max {x, y}; and vice-versa.
- We denote an increasing comparator with ⊕ and decreasing comparator with ⊖
- The "**speed**" of the network is proportional to its depth

# Sorting Networks: Comparators

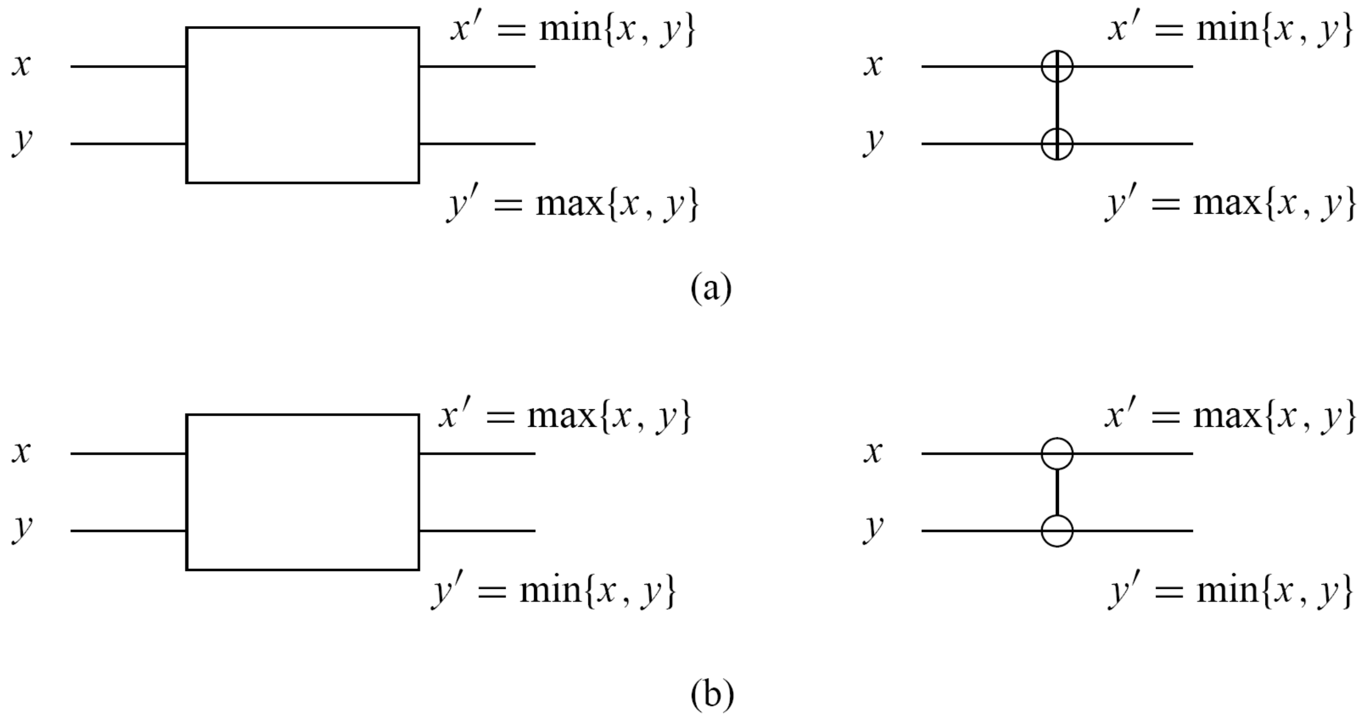


**Figure 9.3** A schematic representation of comparators: (a) an increasing comparator, and (b) a decreasing comparator.

# Ordering Networks

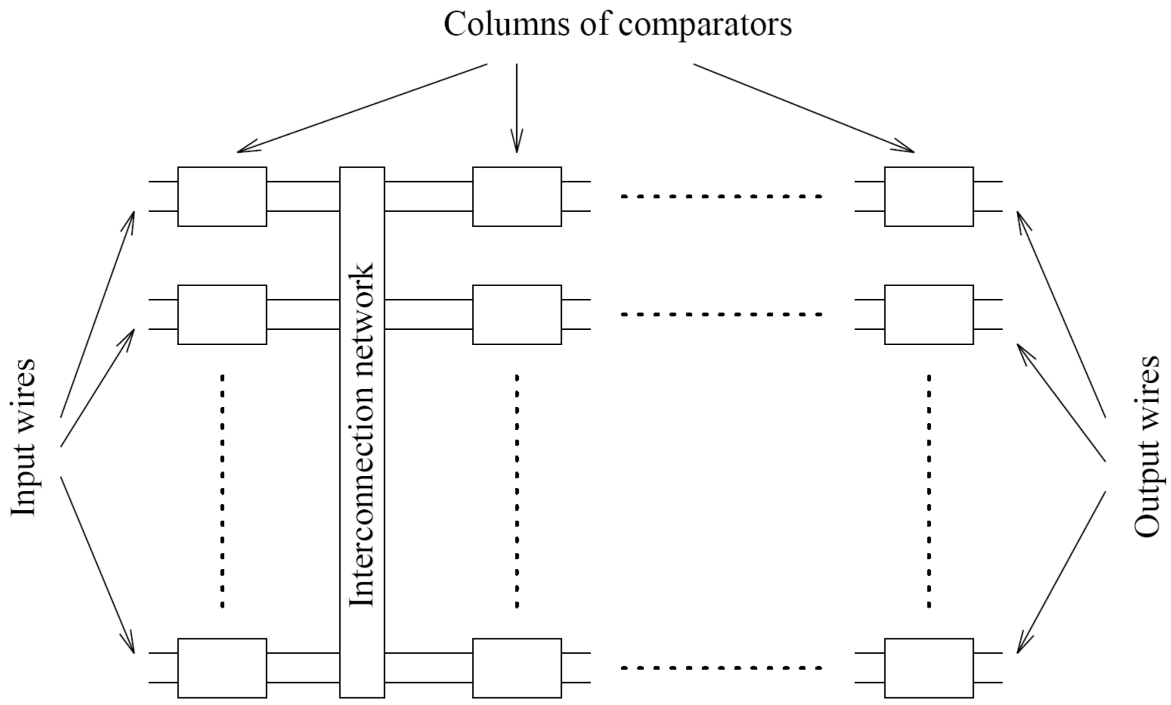


**Figure 9.4** A typical sorting network. Every sorting network is made up of a series of columns, and each column contains a number of comparators connected in parallel.

## Sorting Networks : Bitonic Sort

- A **bitonic** sorting network sorts n elements in time $\Theta(\log^2 n)$
- A **bitonic sequence** has two “**tonalities**" - ascending and descending, or vice versa. Any cyclic rotation of such networks is also considered **bitonic**
- For example $\langle 1,2,4,7,6,0 \rangle$ is a bitonic sequence, because before it increases and then decreases. $\langle 8,9,2,1,0,4 \rangle$ is another bitonic sequence, because it is a cyclic shift of $\langle 0,4,8,9,2,1 \rangle$
- **The heart of the network is the reordering of the bitonic sequence into an ordered sequence**

# Sorting Networks : Bitonic Sort

- Let $s = \langle a_0, a_1, \ldots, a_{n-1} \rangle$ be a bitonic sequence **such that** $a_0 \leq a_1 \leq \cdots \leq a_{n/2-1}$ e $a_{n/2} \geq a_{n/2+1} \geq \cdots \geq a_{n-1}$.

- Let's consider the following sub-sequences of $s$:

$$s_1 = \langle \min\{a_0, a_{n/2}\}, \min\{a_1, a_{n/2+1}\}, \ldots, \min\{a_{n/2-1}, a_{n-1}\} \rangle$$

$$s_2 = \langle \max\{a_0, a_{n/2}\}, \max\{a_1, a_{n/2+1}\}, \ldots, \max\{a_{n/2-1}, a_{n-1}\} \rangle$$

- Note that $s_1$ and $s_2$ are both bitonic and that each element of $s_1$ is smaller than each element of $s_2$.

- We can re-apply the procedure recursively on $s_1$ and $s_2$ to obtain subsequences of size 1, i.e. the **ordered sequence (bitonic merge)**

# Sorting Networks : Bitonic Sort

| | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Original sequence | 3 | 5 | 8 | 9 | 10 | 12 | 14 | 20 | 95 | 90 | 60 | 40 | 35 | 23 | 18 | 0 |
| 1st Split | 3 | 5 | 8 | 9 | 10 | 12 | 14 | 0 | 95 | 90 | 60 | 40 | 35 | 23 | 18 | 20 |
| 2nd Split | 3 | 5 | 8 | 0 | 10 | 12 | 14 | 9 | 35 | 23 | 18 | 20 | 95 | 90 | 60 | 40 |
| 3rd Split | 3 | 0 | 8 | 5 | 10 | 9 | 14 | 12 | 18 | 20 | 35 | 23 | 60 | 40 | 95 | 90 |
| 4th Split | 0 | 3 | 5 | 8 | 9 | 10 | 12 | 14 | 18 | 20 | 23 | 35 | 40 | 60 | 90 | 95 |

**Figure 9.5** Merging a 16-element bitonic sequence through a series of $\log 16$ bitonic splits.

The sorting of a bitonic sequence occurs in log n steps!

# Sorting Networks : Bitonic Sort

- We can now build a **sorting** network to implement the **bitonic merge algorithm.**
- This network is called ***bitonic merging network***.
- This network contains **log n columns**. Each column contains n / 2 **comparators** and performs one step of the bitonic merge.
- We denote by **⊕BM[n]** a **bitonic merging network** with n inputs
- **Replacing** the comparators ⊕ with the Ɵ comparators we get a **descending output sequence** : This network is called ƟBM [n]
- ⊕BM[n] is also known as **Co-Ex-Low**; ƟBM [n] is also known as **Co-Ex-High**

# Sorting Networks : Bitonic Sort

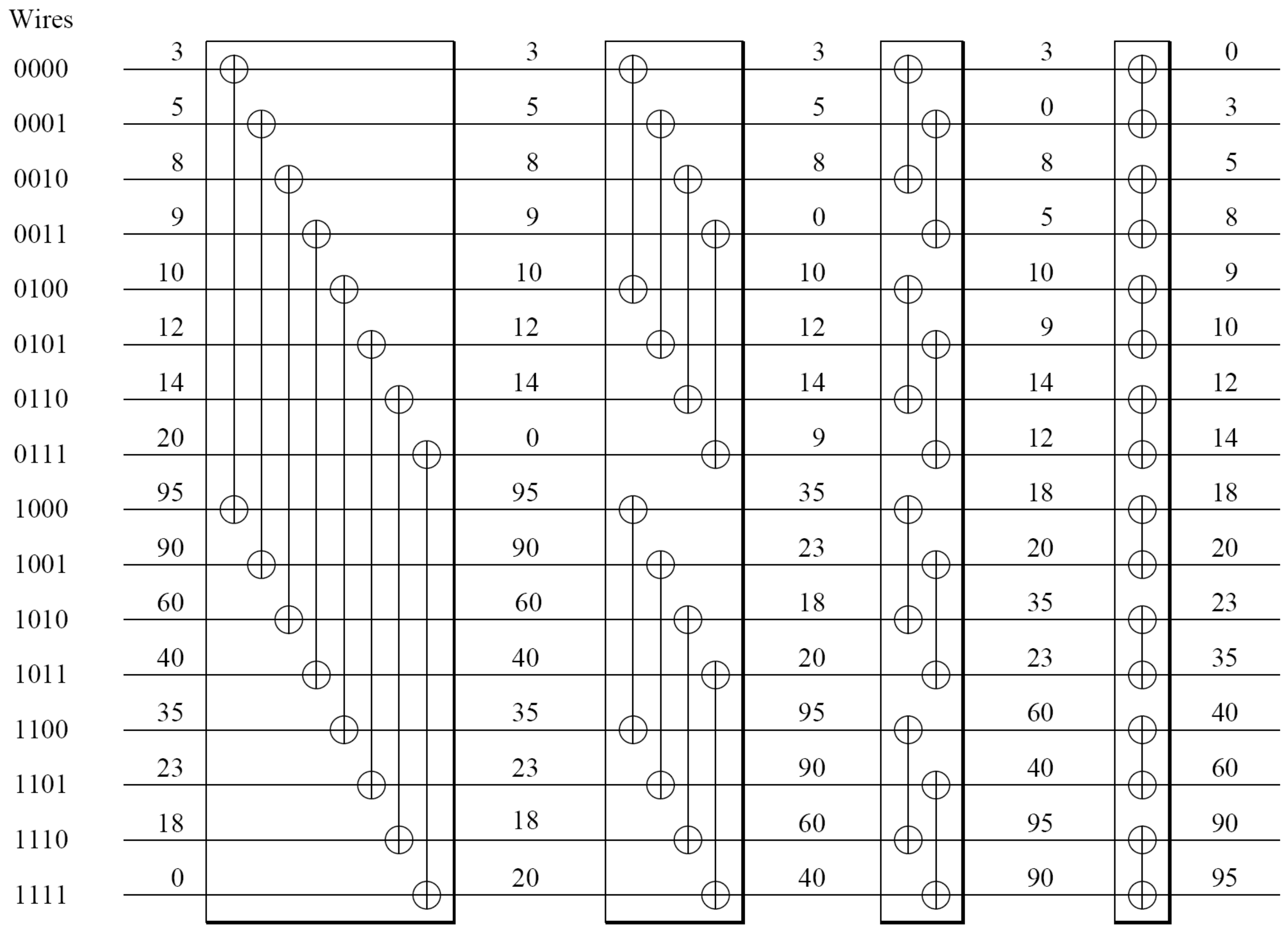


**Figure 9.6** A bitonic merging network for $n = 16$. The input wires are numbered $0, 1 \ldots, n-1$, and the binary representation of these numbers is shown. Each column of comparators is drawn separately; the entire figure represents a $\oplus$BM[16] bitonic merging network. The network takes a bitonic sequence and outputs it in sorted order.

# Sorting Networks : Bitonic Sort

***How do we sort a not ordered sequence using a bitonic merge?***

- We must first build a single bitonic sequence from the given sequence
- **We note that a sequence of length 2 is bitonic**. So any unordered sequence of elements can be thought of as a **concatenation** of bitonic sequences!
- A bitonic sequence of length 4 can be constructed by ordering the first two elements using **⊕BM[2]** and the next two with ⊖BM [2]
- This process can be **repeated** to generate more bitonic sequences

# Sorting Networks : Bitonic Sort

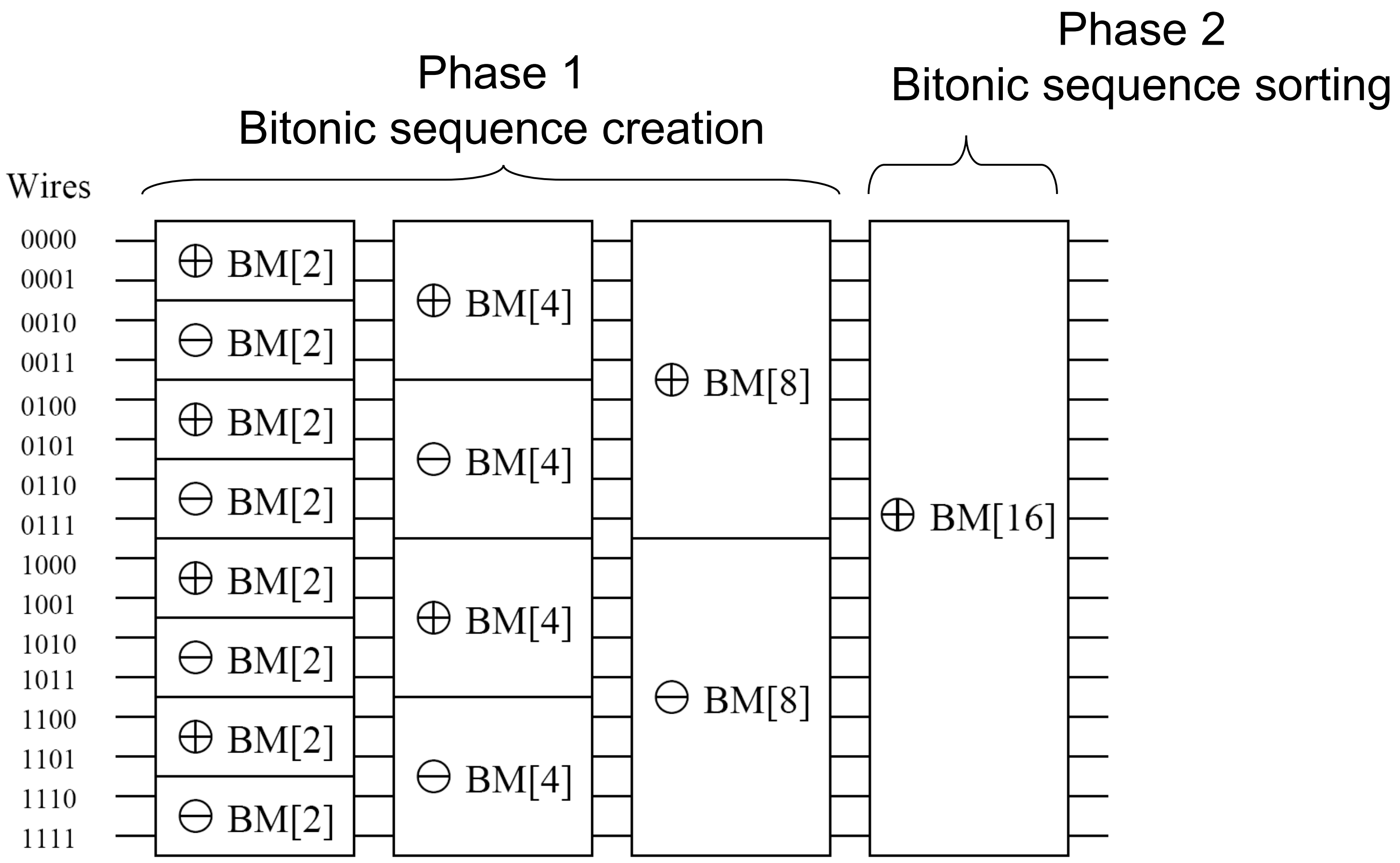


**Figure 9.7** A schematic representation of a network that converts an input sequence into a bitonic sequence. In this example, $\oplus$BM[k] and $\ominus$BM[k] denote bitonic merging networks of input size $k$ that use $\oplus$ and $\ominus$ comparators, respectively. The last merging network ($\oplus$BM[$16$]) sorts the input. In this example, $n = 16$.

## Sorting Networks : Bitonic Sort (Phase 1)

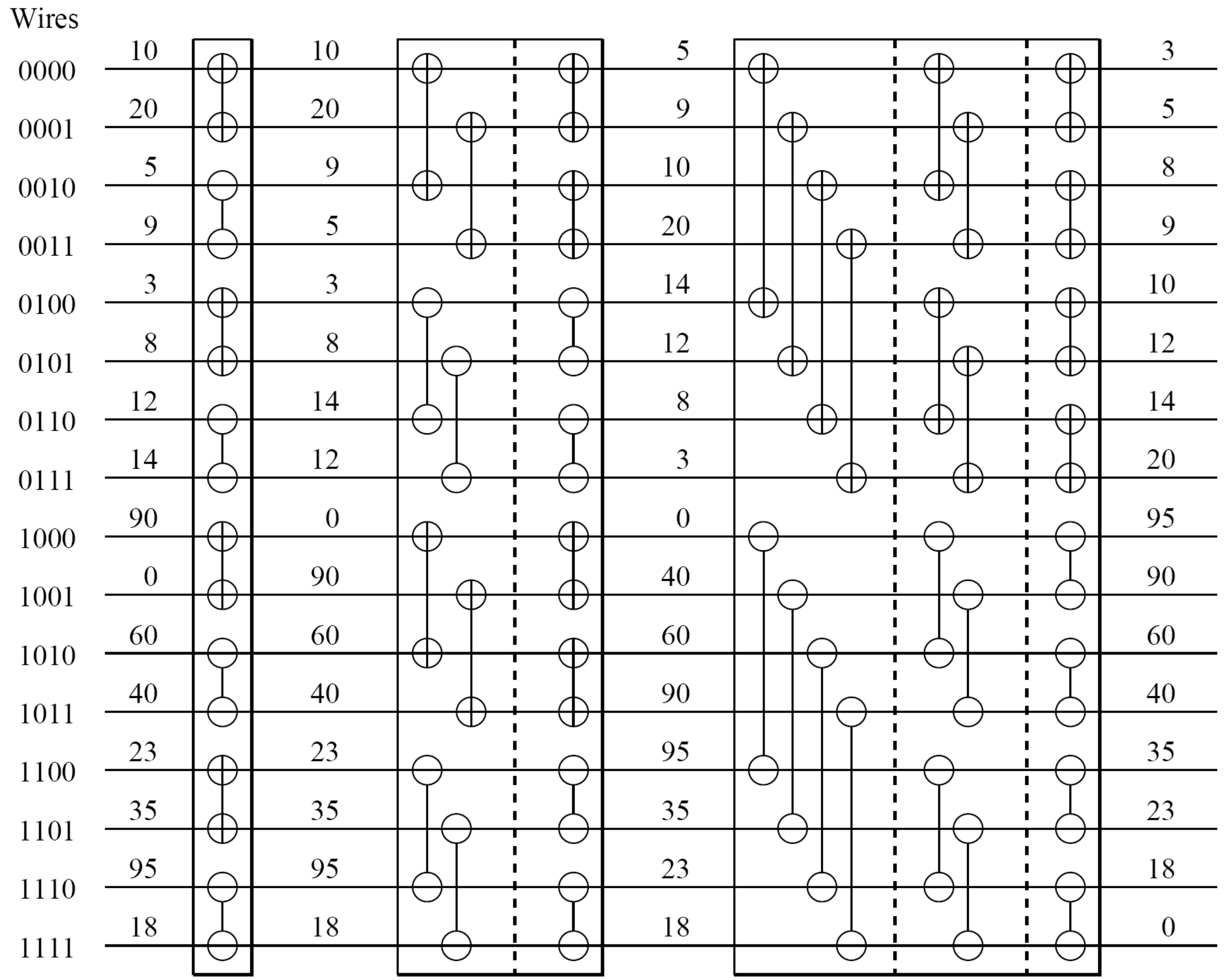


**Figure 9.8** The comparator network that transforms an input sequence of 16 unordered numbers into a bitonic sequence. In contrast to Figure 9.6, the columns of comparators in each bitonic merging network are drawn in a single box, separated by a dashed line.

# Sorting Networks : Bitonic Sort

- So, the **complete algorithm (GENERAL BITONIC SORT)** consists in:
  - **Creation of a bitonic sequence** starting from a non order one (fig. 9.8) (Phase 1)
  - Application of the **bitonic merging network** to such sequence (fig. 9.6), with depth log n (Phase 2 – Bitonic sort)
- Phase 1 totally orders n/2 elements
- So, resolving the following recurrence relation (d = tree depth)

$$d(n) = d\left(\frac{n}{2}\right) + \log n$$

$$d(n) = \sum_{i=1}^{\log n} i = \frac{log^2 n + log n}{2} = \Theta(log^2 n)$$

# Sorting Networks : Bitonic Sort

$$
\begin{aligned}
T(n) &= \log(n) + \log(n/2) + \log(n/4) + \dots + 1 = \\
&= \log(n) + [\log(n)-1] + [\log(n)-2] + [\log(n) - 3] + \dots + 1 = \\
&= \log(n)*\log(n) - (2 + 3 + \dots \log(n)) = \\
&= \log(n)*\log(n) - ([(2+\log(n))*\log(n)]/2) = \\
&= \log(n)*\log(n) - \sim 0.5\log(n)*\log(n) = \\
&= \sim 0.5\log(n)*\log(n)
\end{aligned}
$$

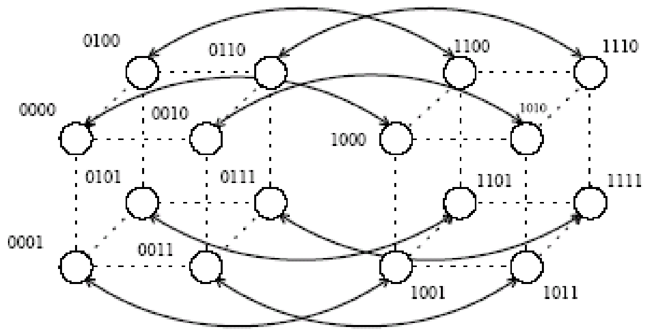
So,

**$T(n) = O(\log(n)*\log(n))$**

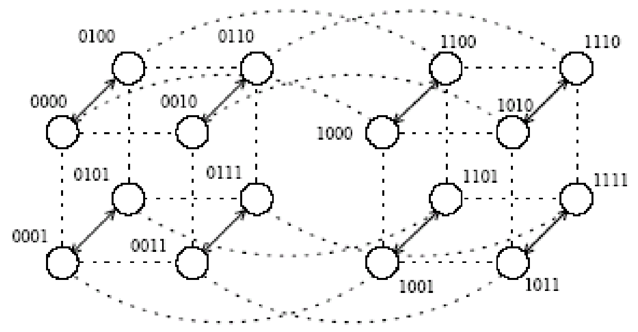
## Bitonic Sort for hypercubes

- Consider the case in which we have **one element** for processor
- Note that the compare-exchange operation takes place between two "connections" (therefore, processors) that differ by exactly one bit!
- It’s therefore **natural** to implement a connections-processors direct mapping using a hypercube where, let’s recall, two processes are "close" if their binary representations differ by exactly one bit!

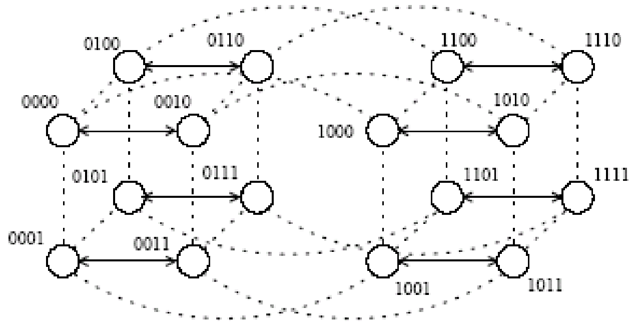
# Bitonic Sort for hypercubes



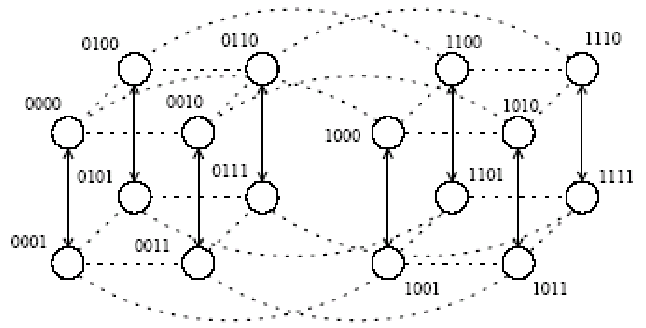
Step 1



Step 2



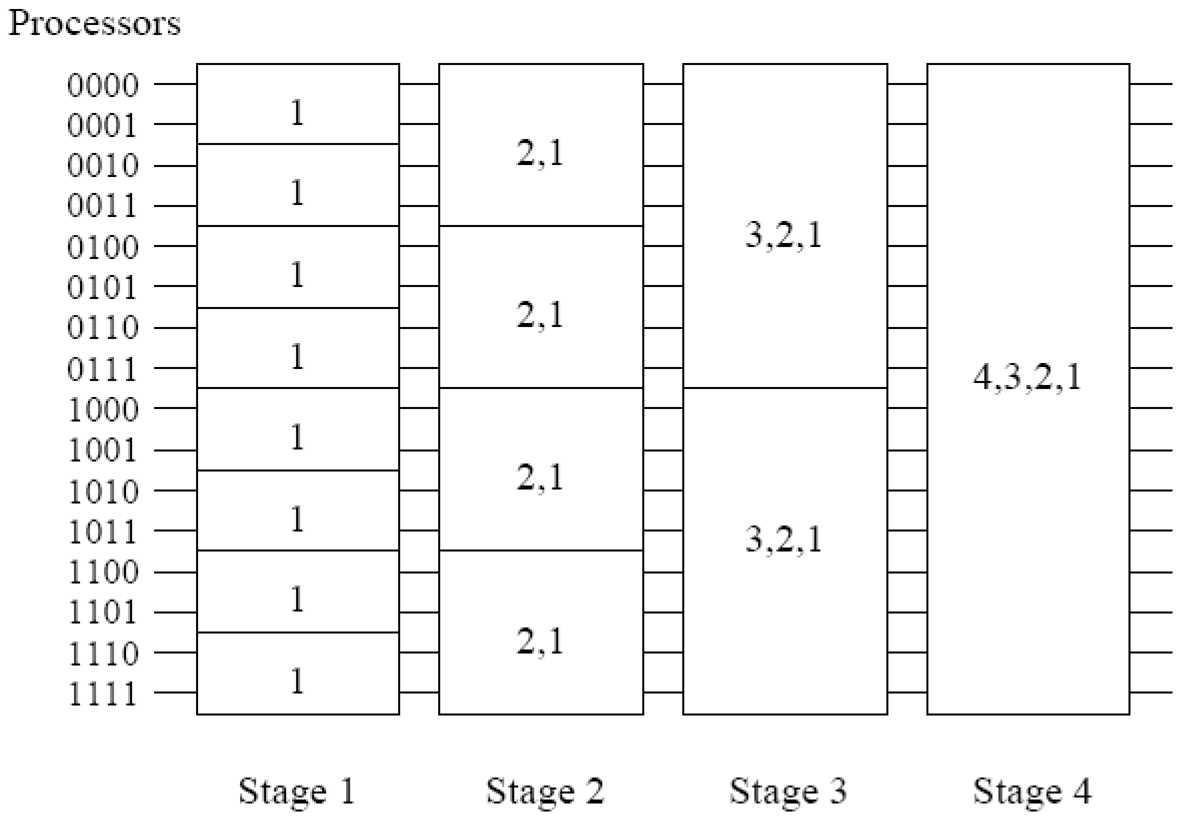
Step 3



Step 4

Communications during the **last step** of the bitonic sort. Each connection is mapped to a process of the hypercube; each connection is a compare-exchange between processes

# Bitonic Sort for hypercubes



Features of the communications in bitonic sort on a hypercube. During each phase of the algorithm, processors communicate in dimensions shown in figure

## Bitonic Sort for hypercubes

```
procedure BITONIC_SORT($label, d$)
begin
    for $i := 0$ to $d - 1$ do
        for $j := i$ downto $0$ do
            if $(i + 1)^{st}$ bit of $label \neq j^{th}$ bit of $label$ then
                comp_exchange_max(j);
            else
                comp_exchange_min(j);
end BITONIC_SORT
```

Parallel formulation of bitonic sort on a hypercube with $n = 2^d$ processes

## Bitonic Sort for hypercubes

- During each phase of the algorithm, each process executes a compare-exchange operation (single communication to the “direct” neighbour)
- Since each step takes Θ(1), the parallel time is:

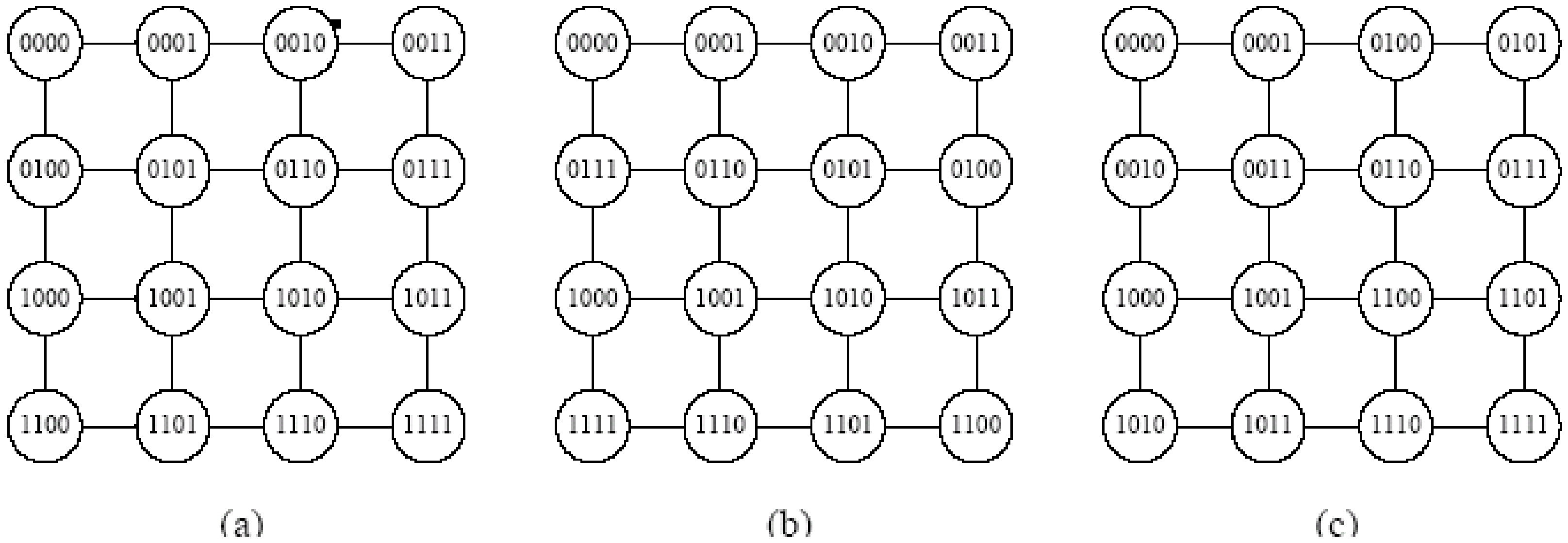
$$T_p = \Theta(\log^2 n)$$

- The algorithm is cost optimal with respect to the sequential version, but not with respect to the best parallel sorting algorithm

## Bitonic Sort for Grids

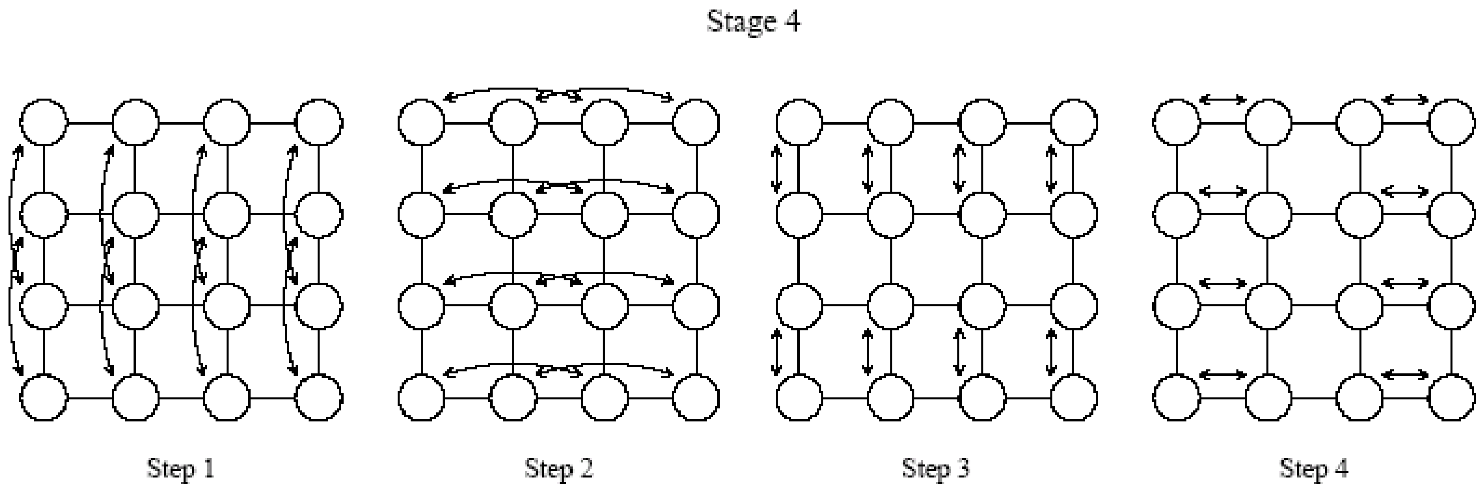
- The connectivity of a grid / mesh is lower than that of a hypercube, so we expect some overhead ...
- A shuffled row-major mapping is usually considered on processes ...

# Mapping Bitonic Sort to Meshes



Different ways of mapping the input wires of the bitonic sorting network to a mesh of processes: (a) row-major mapping, (b) row-major snakelike mapping, and (c) row-major shuffled mapping

# Mapping Bitonic Sort to Meshes



The **last stage** of the bitonic sort algorithm for $n = 16$ on a mesh, using the row-major shuffled mapping. During each step, process pairs compare-exchange their elements. Arrows indicate the pairs of processes that perform compare-exchange operations.

## Block of Elements Per Processor

- Each process is assigned a block of $n/p$ elements.
- The first step is a local sort of the local block (v. applicazioni seguenti)
- **Each subsequent compare-exchange operation is <u>replaced</u> by a compare-split operation**
- We can effectively view the bitonic network as having $(1 + \log p)(\log p)/2$ steps.

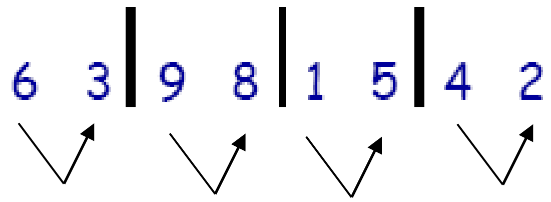
## Applications/
## Appendix

# General Bitonic Sort: Phase 1 (Example)

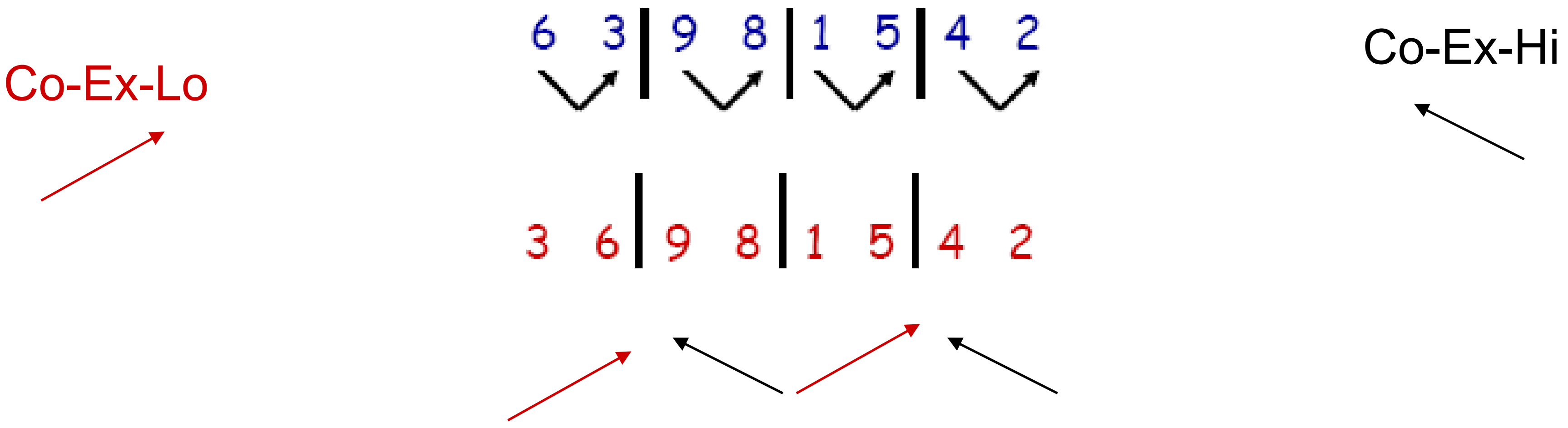
To transform a vector of 8 elements in a bitonic one, partition it in 4 pairs, or in 4 bitonic sub-sequences of length = 2 and order them so you have 2 bitonic subsequences of length = 4

Initial vector: 6 3 | 9 8 | 1 5 | 4 2

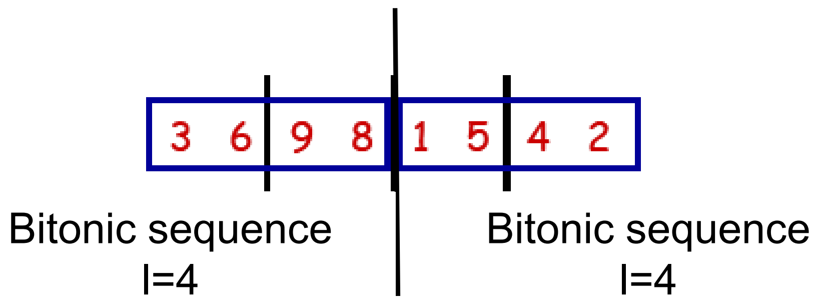
4 subsequences of length 2



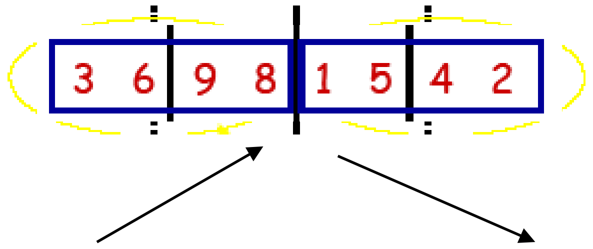
# General Bitonic Sort: Phase 1 (Example)



We obtain:



# General Bitonic Sort: Phase 1 (Example)



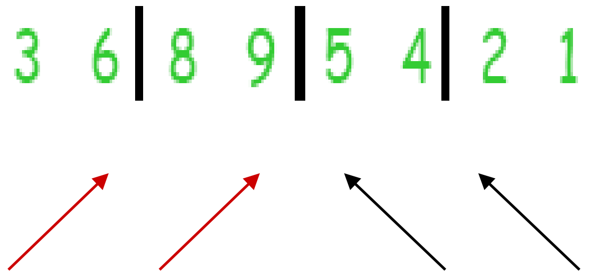
We order each of the two subvectors (with the bitonic Sort -1 phase) in order to have a single bitonic vector

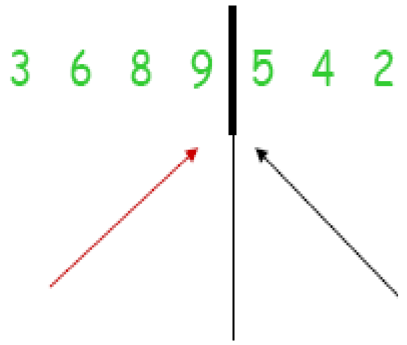
⇩

The first in ascending order
the second in descending order

# General Bitonic Sort: Phase 1 (Example)

We have:



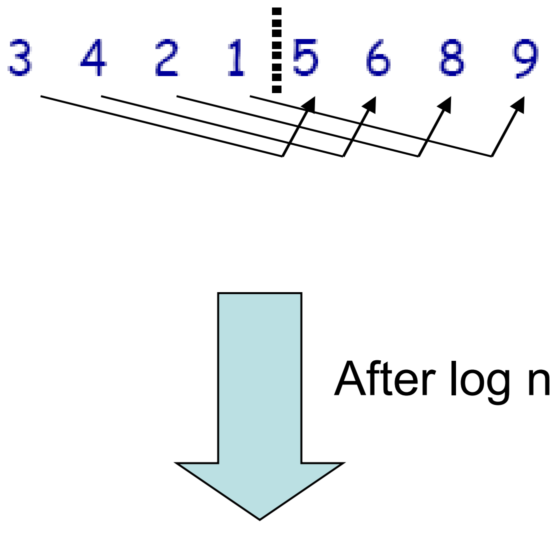


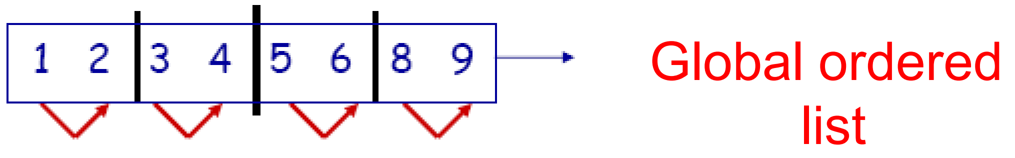
Ordered subsequence in increasing order

Ordered subsequence in decreasing order

## General Bitonic Sort: Phase 2 (Example)

- **Last step:** Sorting a bitonic sequence of length 8 obtained before (with bitonic SORT):





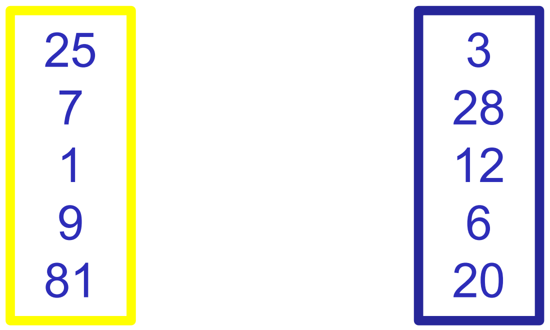
# Sorting in Parallel: General Example

- How can we implement a parallel sorting algorithm, starting from any **sequential** algorithm?

Example p=2

Let's distribute the vector

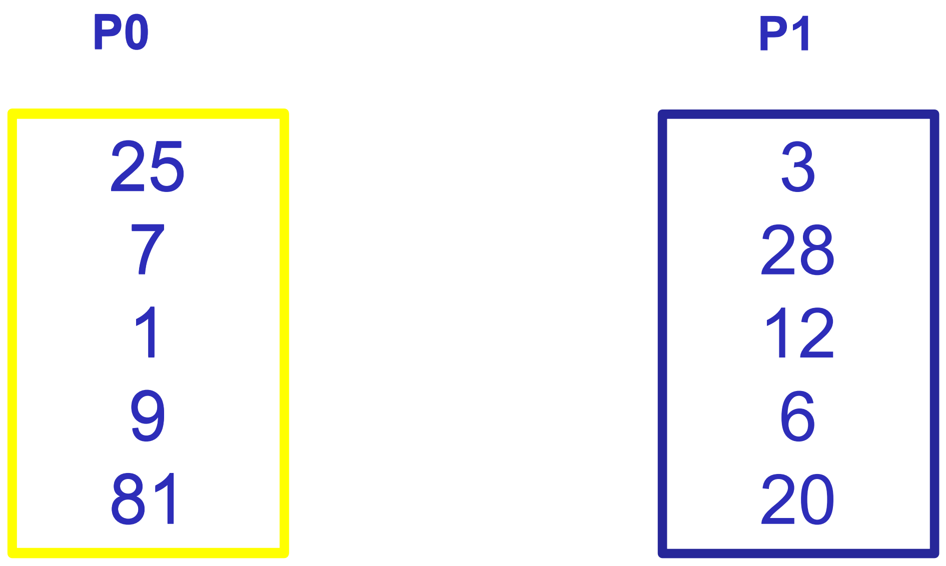
- **L = (25, 7, 1, 9, 81, 3, 28, 12, 6, 20)**



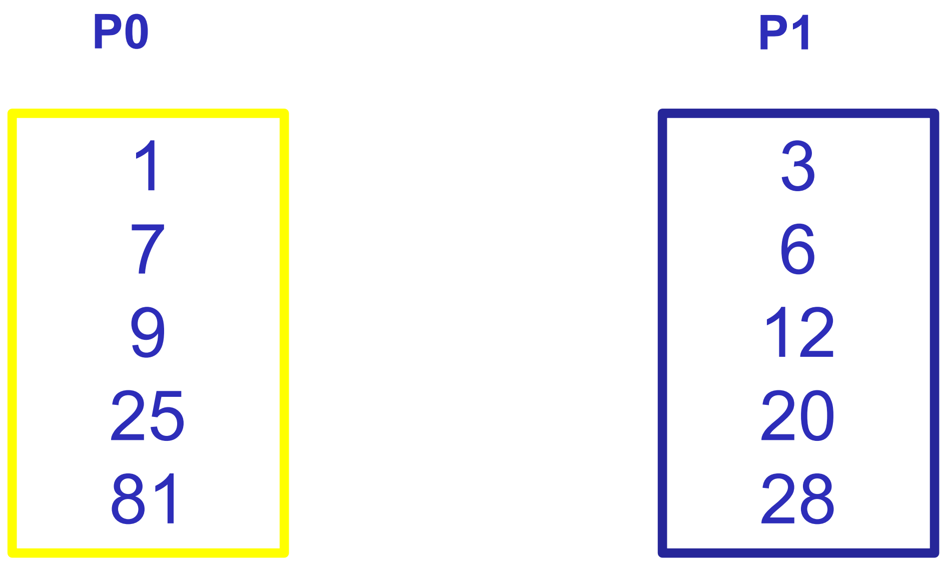
## General Strategy

- Phase 1 (Local Sort)
  - Each processor sorts its vector with a standard sorting algorithm (e.g., quicksort)
- Phase 2 (Merging)
  - The (sub)vectors are oppurtenly combined to obtina a ordered global vector

## Phase 1 (Local sort)



Each process sorts in increasing manner its vector



Last phase, local sort

## Phase 2 (Merging)

The compare-exchange operation is applied to the distriued sequence between the 2 processes

Low process

High process

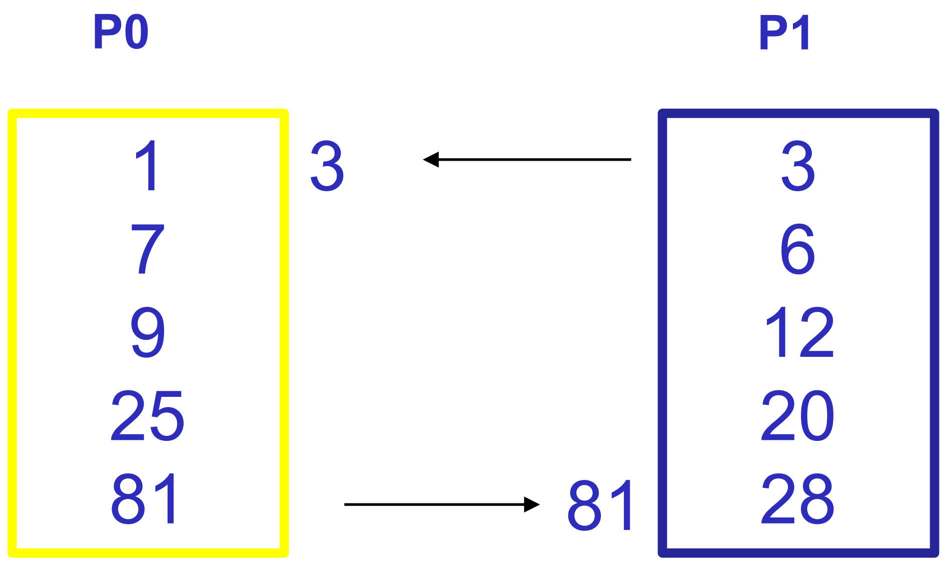
Process P0 contains the smallest elements

Process P1 contains the biggest elements

Co-Ex-Lo

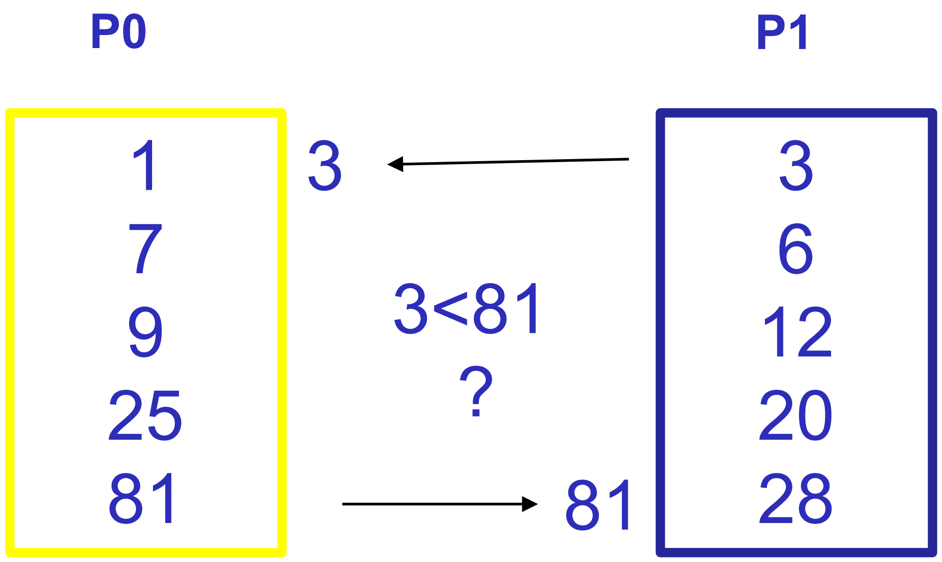
Co-Ex-Hi

## I PASSO: step 1.0 → COMUNICAZIONE - Compare Split?



**Step 1.0:** Po sends to P1 its max= 81, while P1 sends to Po its min=3

## I PASSO : step 1.1 → CONFRONTO

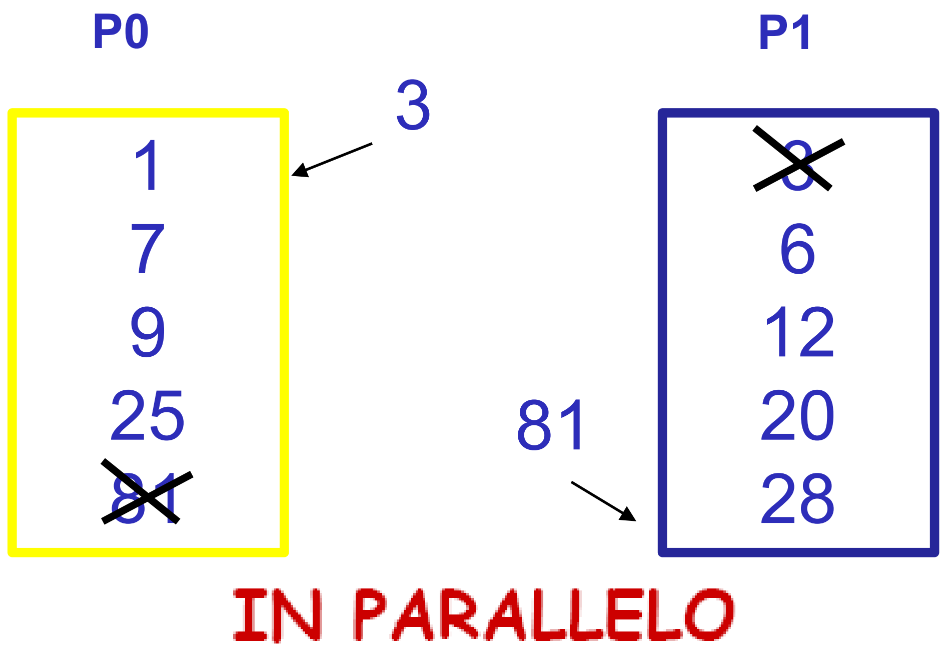


**IN PARALLELO**

**Step 1.1:** Po compares 3 with 81, if smaller it inserts it

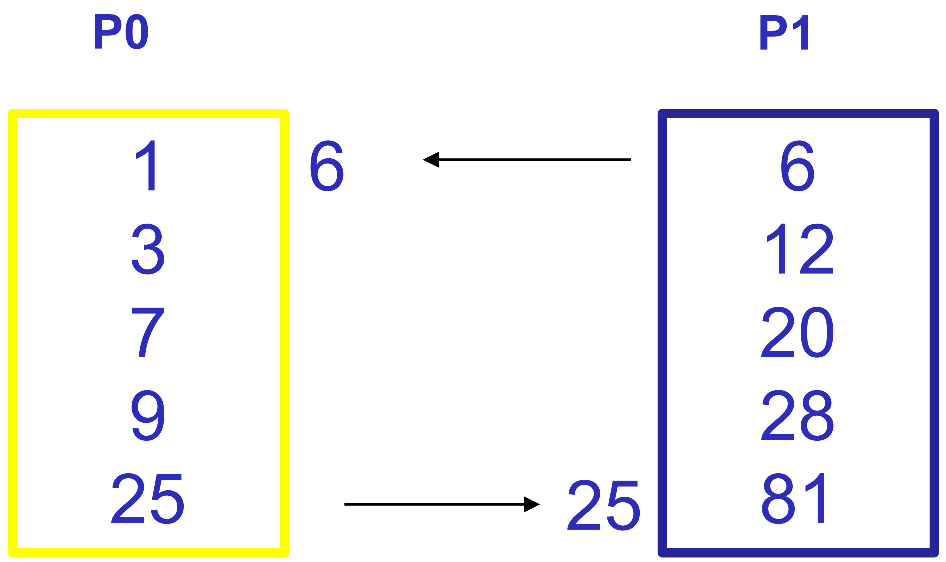
P1 compares 81 with 3, if greater it inserts it

## I PASSO: step 1.2 → INSERIMENTO



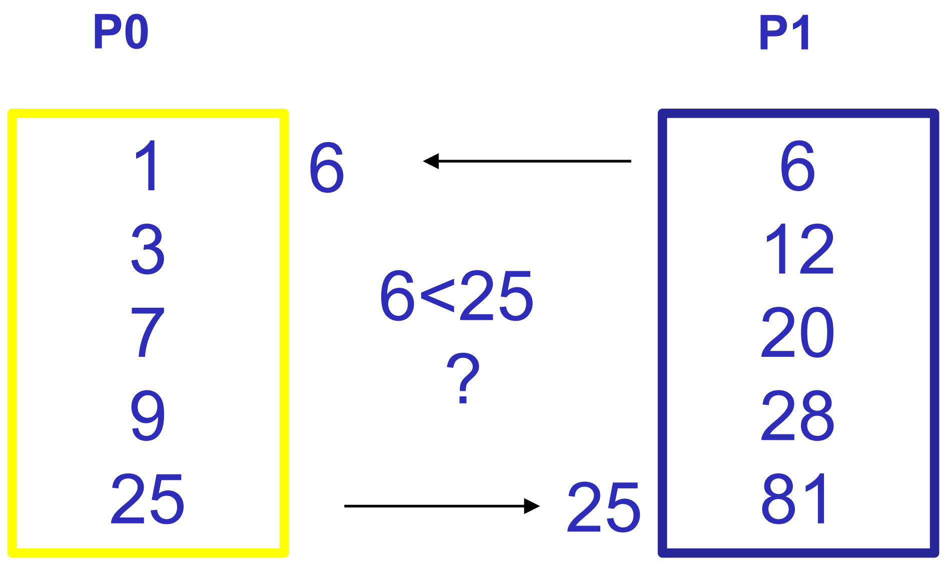
Step 1.2: Each process
1)Inserts the received number
2)Erases the sent number
3)Reorders vector

## II PASSO: step 2.0 → COMUNICAZIONE



**Step 1.0**: $P_0$ invia a $P_1$ il proprio max = 25,
mentre $P_1$ invia a $P_0$ il proprio min = 6

## II PASSO: step 2.1 → CONFRONTO

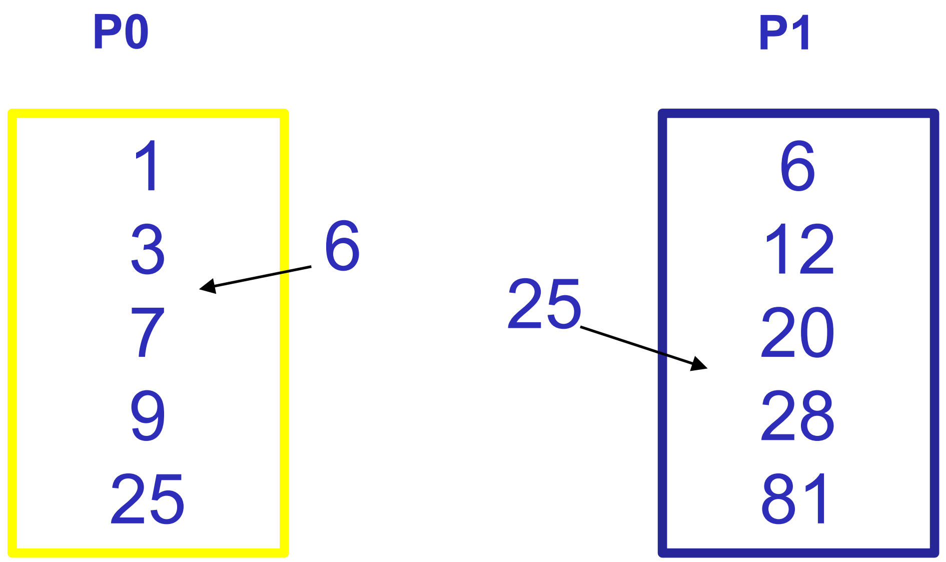


**IN PARALLELO**

**Step 1.1**: $P_0$ confronta 6 con 25: se è minore lo inserisce;

$P_1$ confronta 25 con 6: se è maggiore lo inserisce

## II PASSO: step 2.2 → INSERIMENTO

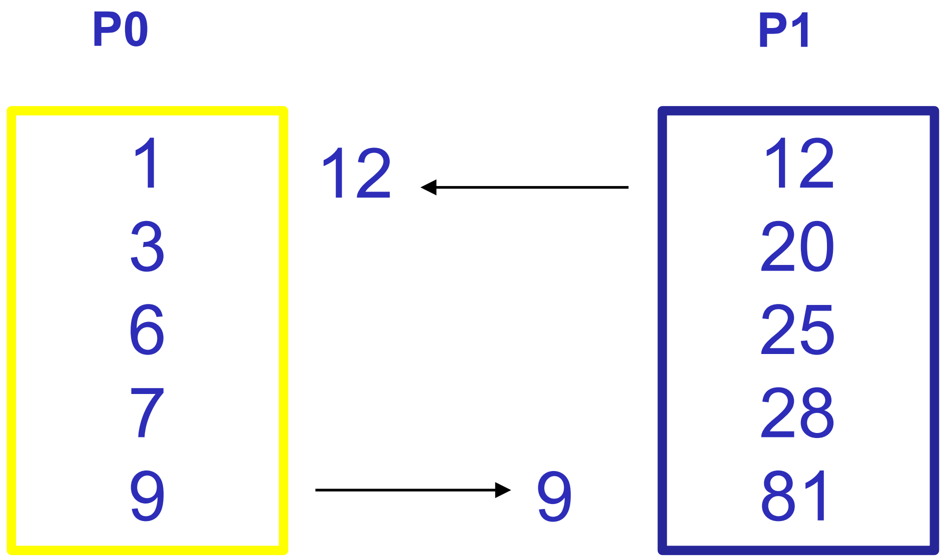


**IN PARALLELO**

**Step 1.2**: Ciascun processore
1) inserisce il numero ricevuto
2) Elimina il numero inviato
3) riordina il vettore

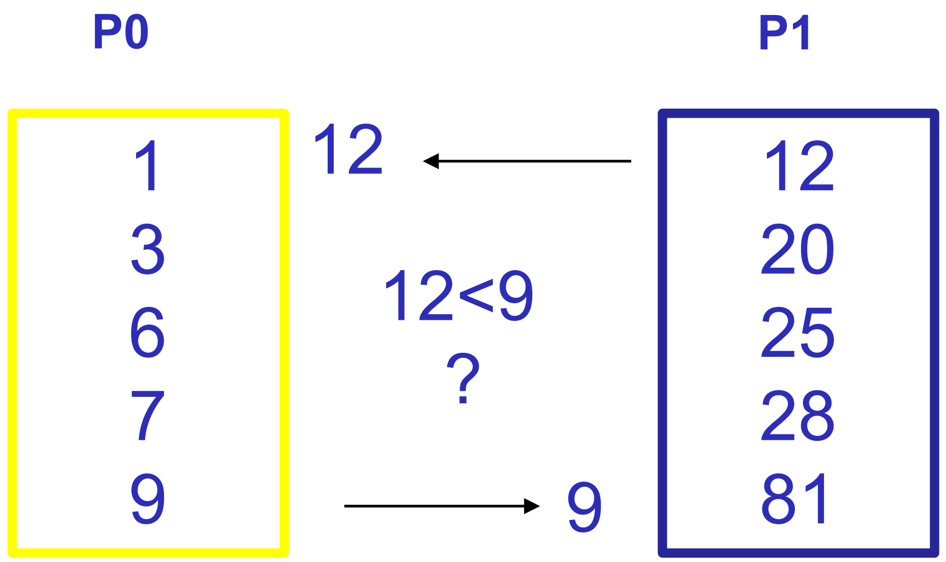
## III PASSO: step 3.0 → COMUNICAZIONE

FASE 2 (MERGING)



**Step 1.0:** $P_0$ invia a $P_1$ il proprio max = 9,
mentre $P_1$ invia a $P_0$ il proprio min = 12

## III PASSO: step 3.1 → CONFRONTO

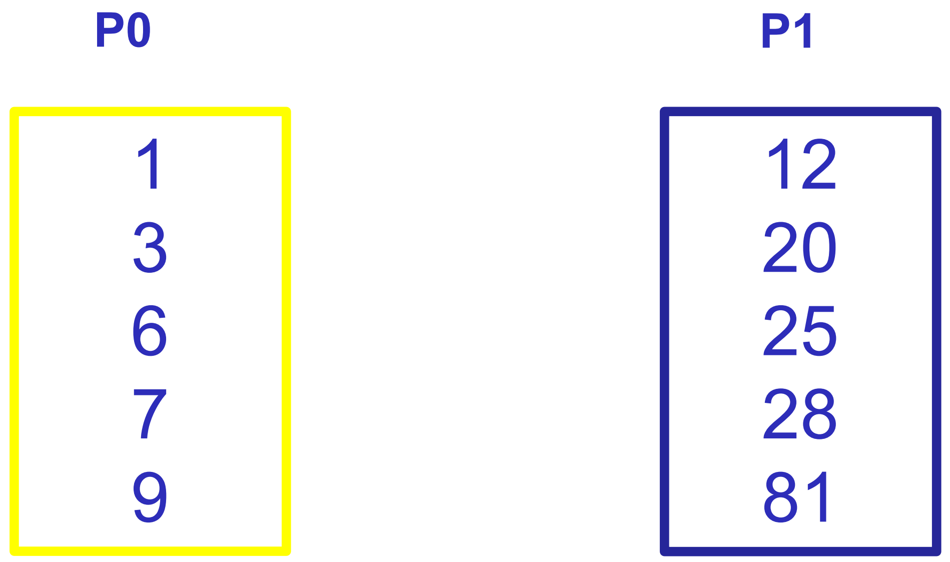


IN PARALLELO

Step 1.1: $P_0$ confronta il numero 12 con 9: non è minore e termina l' esecuzione.

$P_1$ confronta il numero 9 con 12: non è maggiore e termina l'esecuzione

# PGBS:Ordinamento parallelo di un vettore

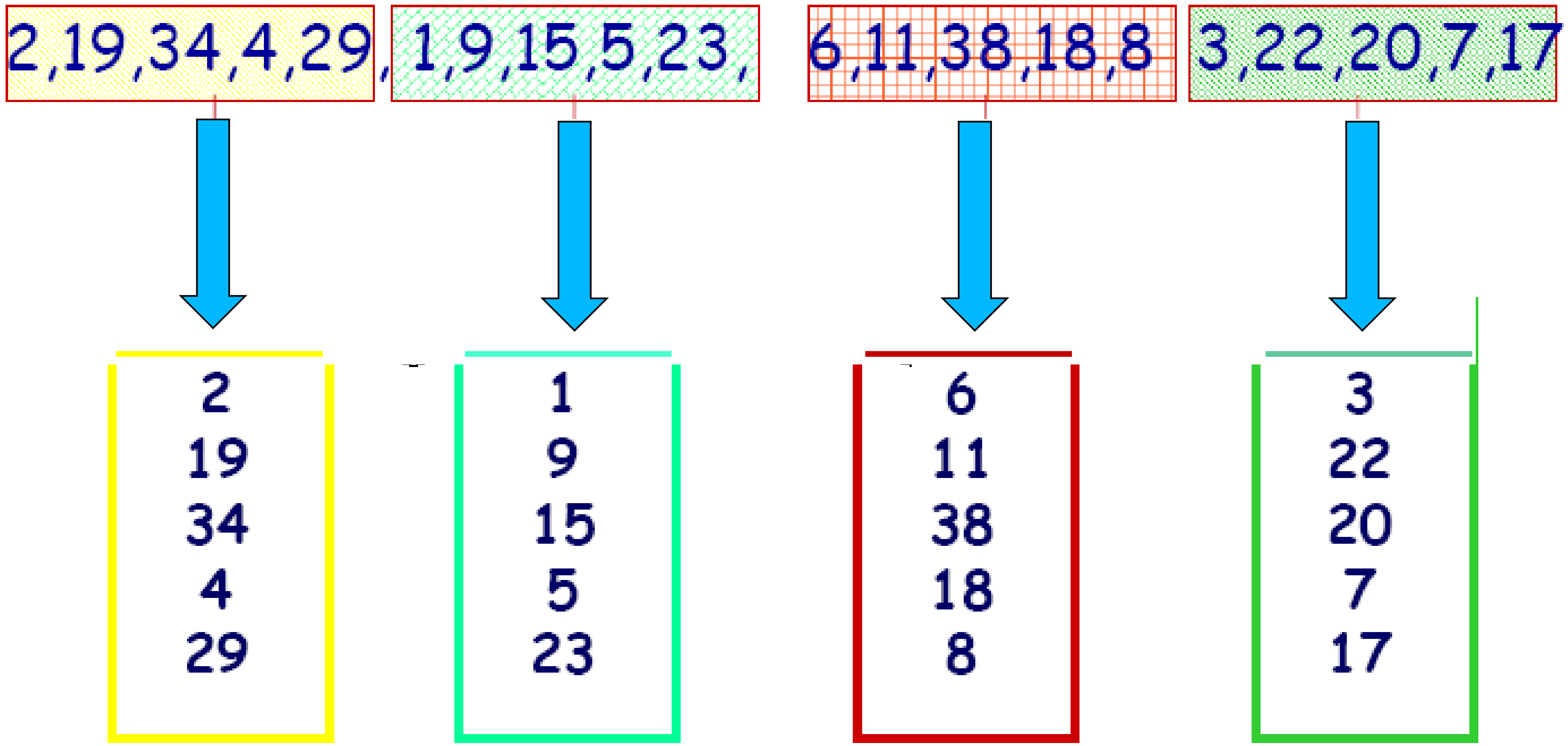


I 5 numeri più **piccoli** sono in $P_0$ mentre i 5 più **grandi** sono in $P_1$

Le due sottoliste iniziali sono globalmente ordinate secondo l'ordine crescente dell'identificativo dei processi

## Esempio: p=4, n=20

Passo 1: distribuzione dati



## Esempio: p=4, n=20

**FASE 1 (SORTING LOCALE)**

| $P_0$ | $P_1$ | $P_2$ | $P_3$ |
|---|---|---|---|
| 2 | 1 | 6 | 3 |
| 4 | 5 | 8 | 7 |
| 19 | 9 | 11 | 17 |
| 29 | 15 | 18 | 20 |
| 34 | 23 | 38 | 22 |

Ogni processore ordina in modo crescente
la propria sottolista

## Esempio: p=4, n=20

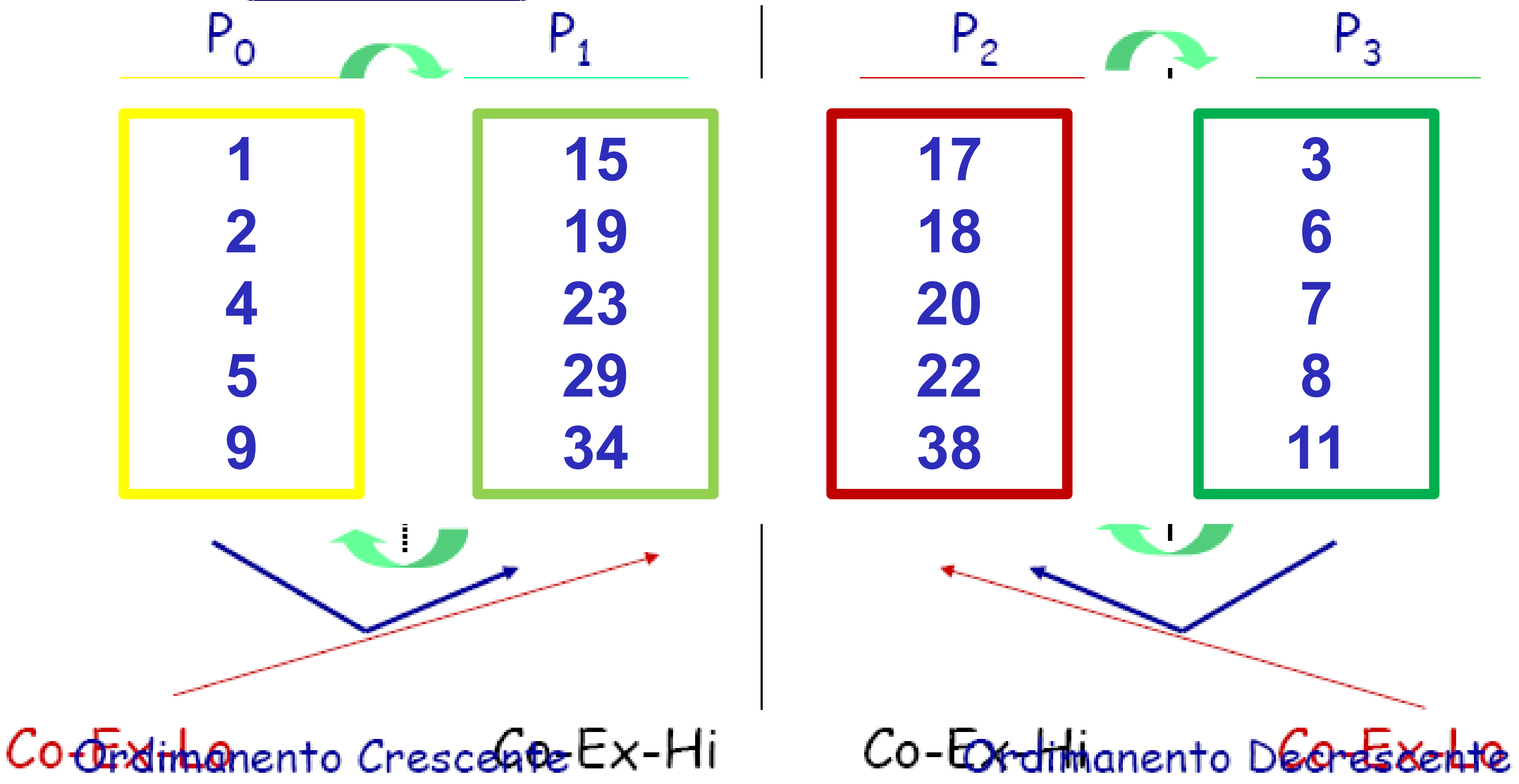
**FASE 2 (MERGING)**

| $P_0$ | $P_1$ | $P_2$ | $P_3$ |
|---|---|---|---|
| 2 | 1 | 6 | 3 |
| 4 | 5 | 8 | 7 |
| 19 | 9 | 11 | 17 |
| 29 | 15 | 18 | 20 |
| 34 | 23 | 38 | 22 |

Si applica l'algoritmo GBS a tutto il vettore

# Esempio: p=4, n=20

## FASE 2 (MERGING)



## Esempio: p=4, n=20

**FASE 2 (<u>MERGING</u>)**

| $P_0$ | $P_1$ | $P_2$ | $P_3$ |
|---|---|---|---|
| 1 | 15 | 38 | 11 |
| 2 | 19 | 22 | 8 |
| 4 | 23 | 20 | 7 |
| 5 | 29 | 18 | 6 |
| 9 | 34 | 17 | 3 |

Fine passo i=1 e j=1

Il vettore è bitonico

## Esempio: p=4, n=20

**FASE 2 (MERGING)**

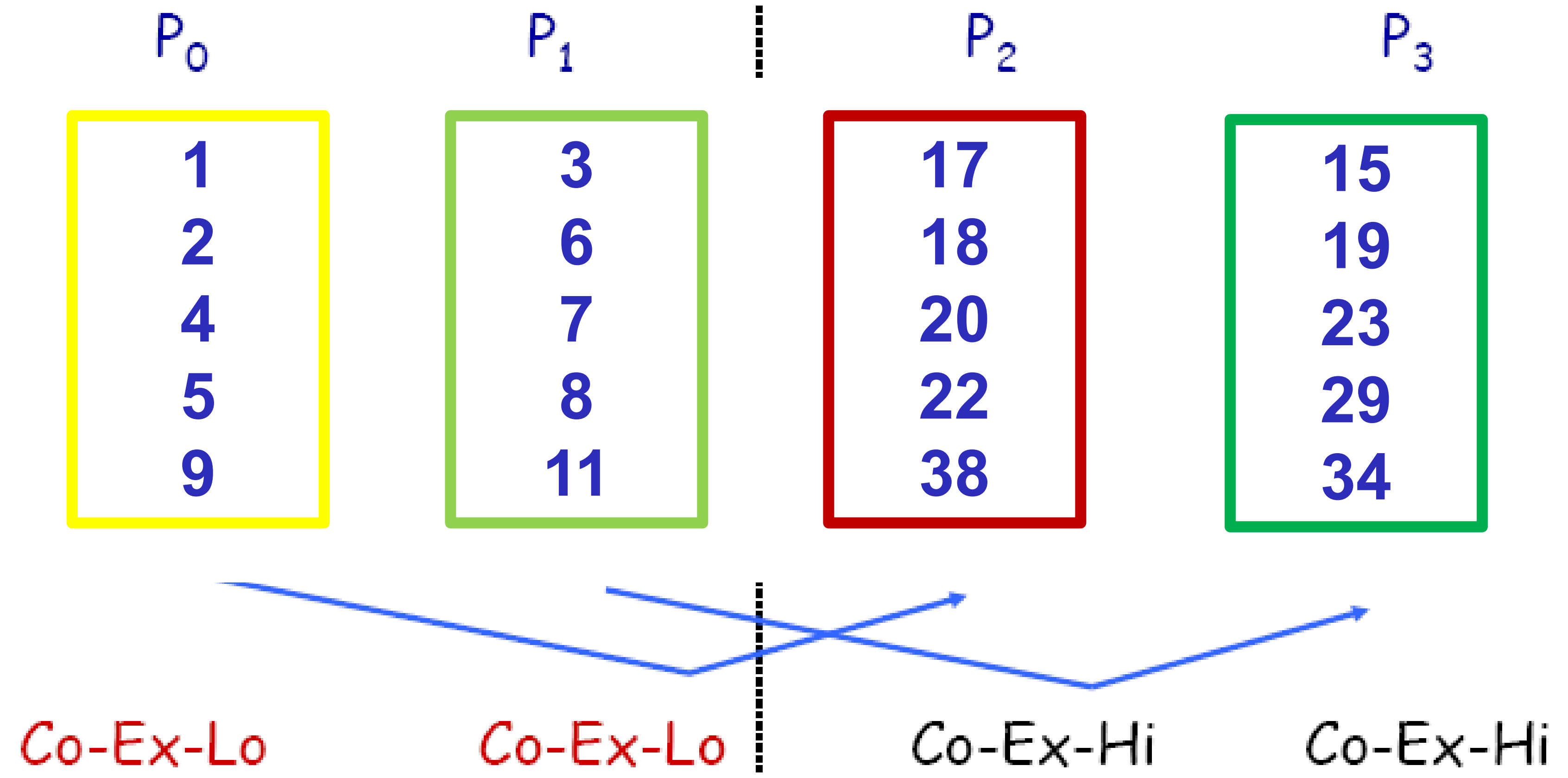
| $P_0$ | $P_1$ | $P_2$ | $P_3$ |
|---|---|---|---|
| 1 | 15 | 38 | 11 |
| 2 | 19 | 22 | 8 |
| 4 | 23 | 20 | 7 |
| 5 | 29 | 18 | 6 |
| 9 | 34 | 17 | 3 |

Passo i=2

ordiniamo globalmente il vettore

# Esempio: p=4, n=20

## FASE 2 (<u>MERGING</u>)



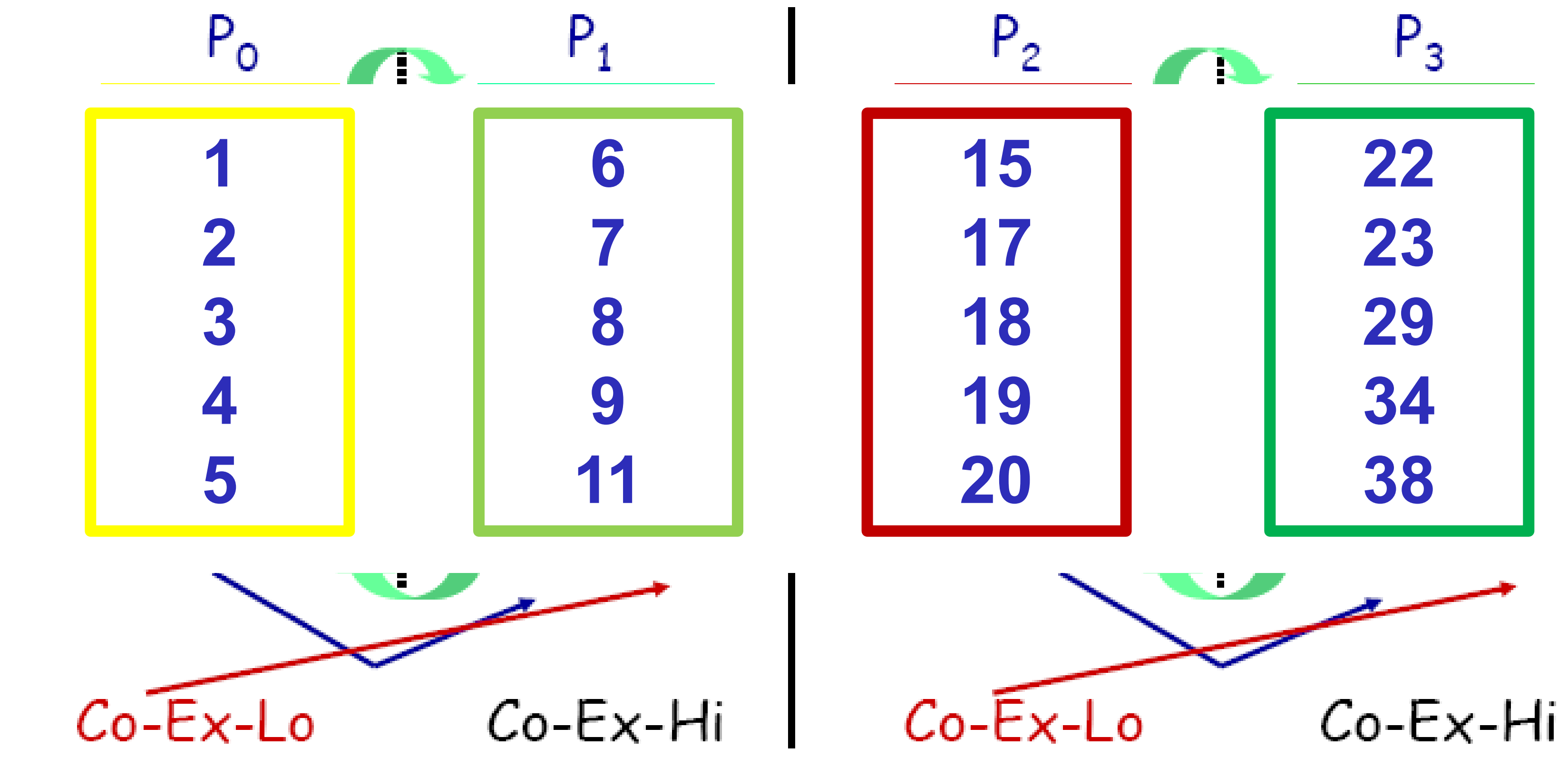
Ordinamento Crescente Ordinamento Crescente

Passo i=2 ( j=1 )

Esempio: p=4, n=20

FASE 2 (MER

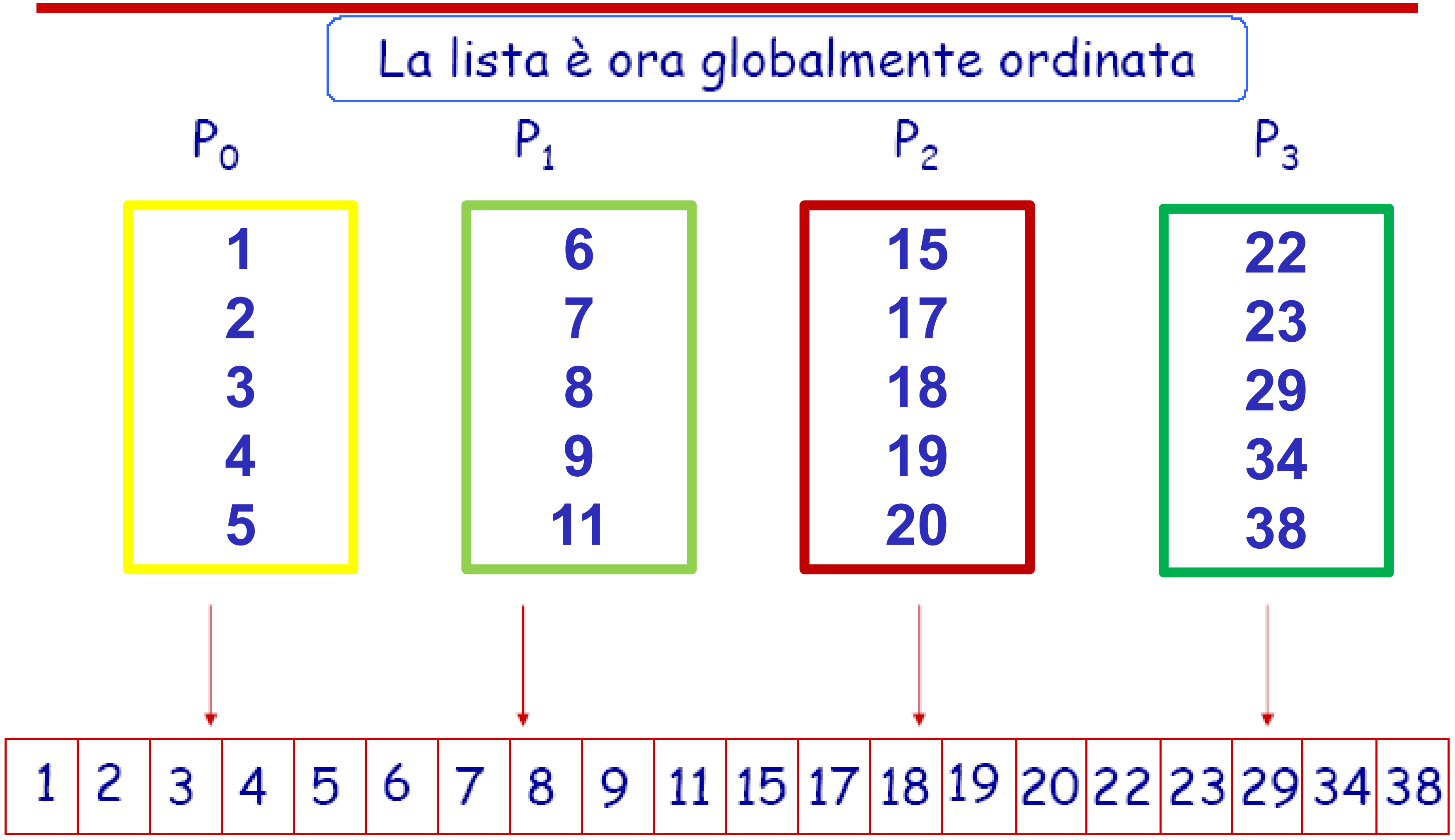
La lista è ora globalmente ordinata



Ordinamento Crescente Ordinamento Crescente

Passo i=2 ( j=2 )

## Esempio: p=4, n=20



## More Appendix

## 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.

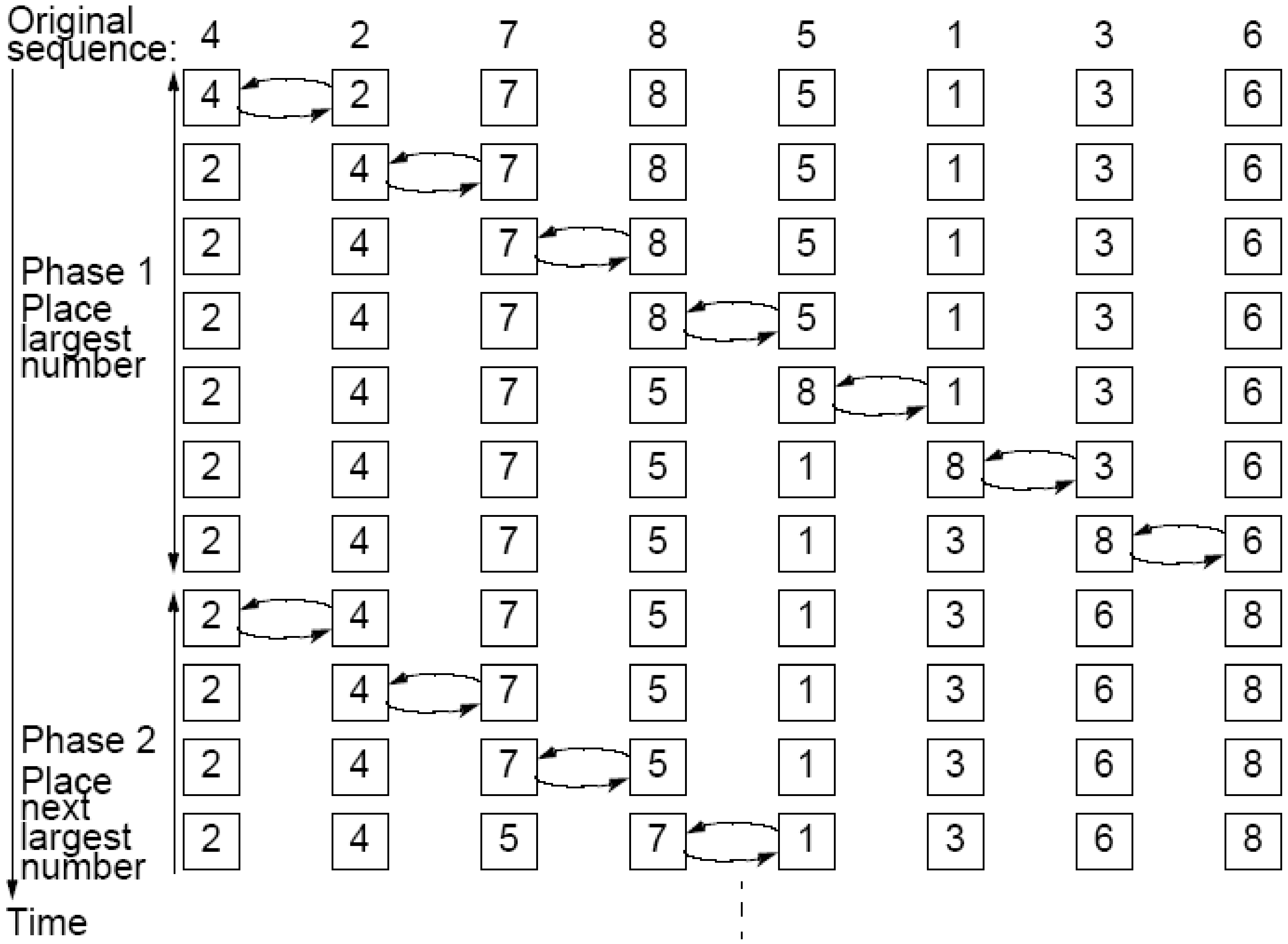
In this way, the larger numbers move (“bubble”) toward one end,

## Bubble Sort and its Variants

The sequential bubble sort algorithm compares and exchanges adjacent elements in the sequence to be sorted:

```
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 BUBBLE_SORT
```

Sequential bubble sort algorithm.


Original sequence: 4 2 7 8 5 1 3 6
Phase 1
Place largest number
Phase 2
Place next largest number
Time

## Time Complexity

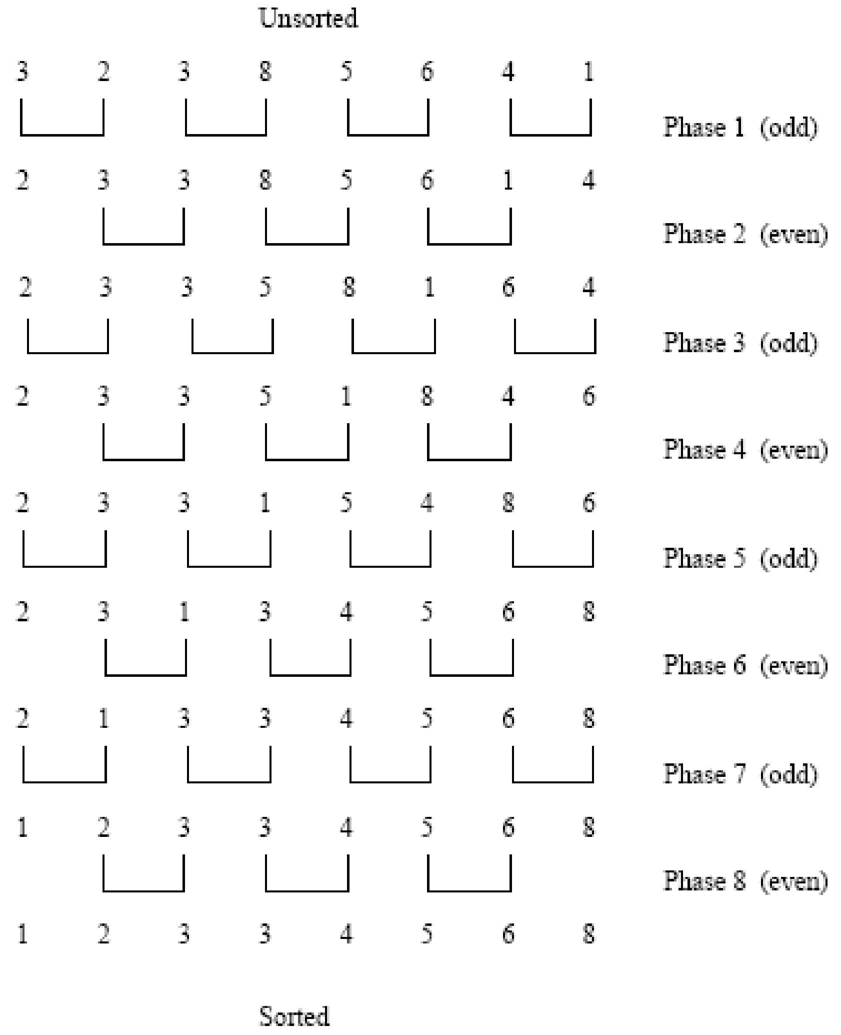
Number of compare and exchange operations $= \sum_{i=1}^{n-1} i = \frac{n(n-1)}{2}$

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

# Bubble Sort and its variants

- The complexity of bubble sort is $\Theta(n^2)$.
- Bubble sort is difficult to parallelize because the algorithm has no explicit parallelism!
- A simple variant (called Odd-Even Transposition), however, reveals an implicit parallelism ...

# Odd-Even Transposition



Sorting $n = 8$ elements, using the odd-even transposition sort algorithm. During each phase, $n = 8$ elements are compared.

## Odd-Even Transposition

```
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
```

Sequential odd-even transposition sort algorithm.

## Odd-Even Transposition

- After $n$ phases of odd-even exchanges, the sequence is sorted.
- Each phase of the algorithm (either odd or even) requires $\Theta(n)$ comparisons.
- Serial complexity is $\Theta(n^2)$.

## Parallel Odd-Even Transposition

- Consider the one item per processor case.
- There are $n$ iterations, in each iteration, each processor does one compare-exchange.
- The parallel run time of this formulation is $\Theta(n)$.
- This is cost optimal with respect to the base serial algorithm but not the optimal one.

# Parallel Odd-Even Transposition

```
procedure ODD-EVEN_PAR($n$)
begin
    $id$ := process's label
    for $i := 1$ to $n$ do
    begin
        if $i$ is odd then
            if $id$ is odd then
                compare-exchange_min($id + 1$);
            else
                compare-exchange_max($id - 1$);
        if $i$ is even then
            if $id$ is even then
                compare-exchange_min($id + 1$);
            else
                compare-exchange_max($id - 1$);
    end for
end ODD-EVEN_PAR
```

Parallel formulation of odd-even transposition.

# Parallel Odd-Even Transposition

- Consider a block of $n/p$ elements per processor.
- The first step is a local sort.
- In each subsequent step, the compare exchange operation is replaced by the compare split operation.
- The parallel run time of the formulation is

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

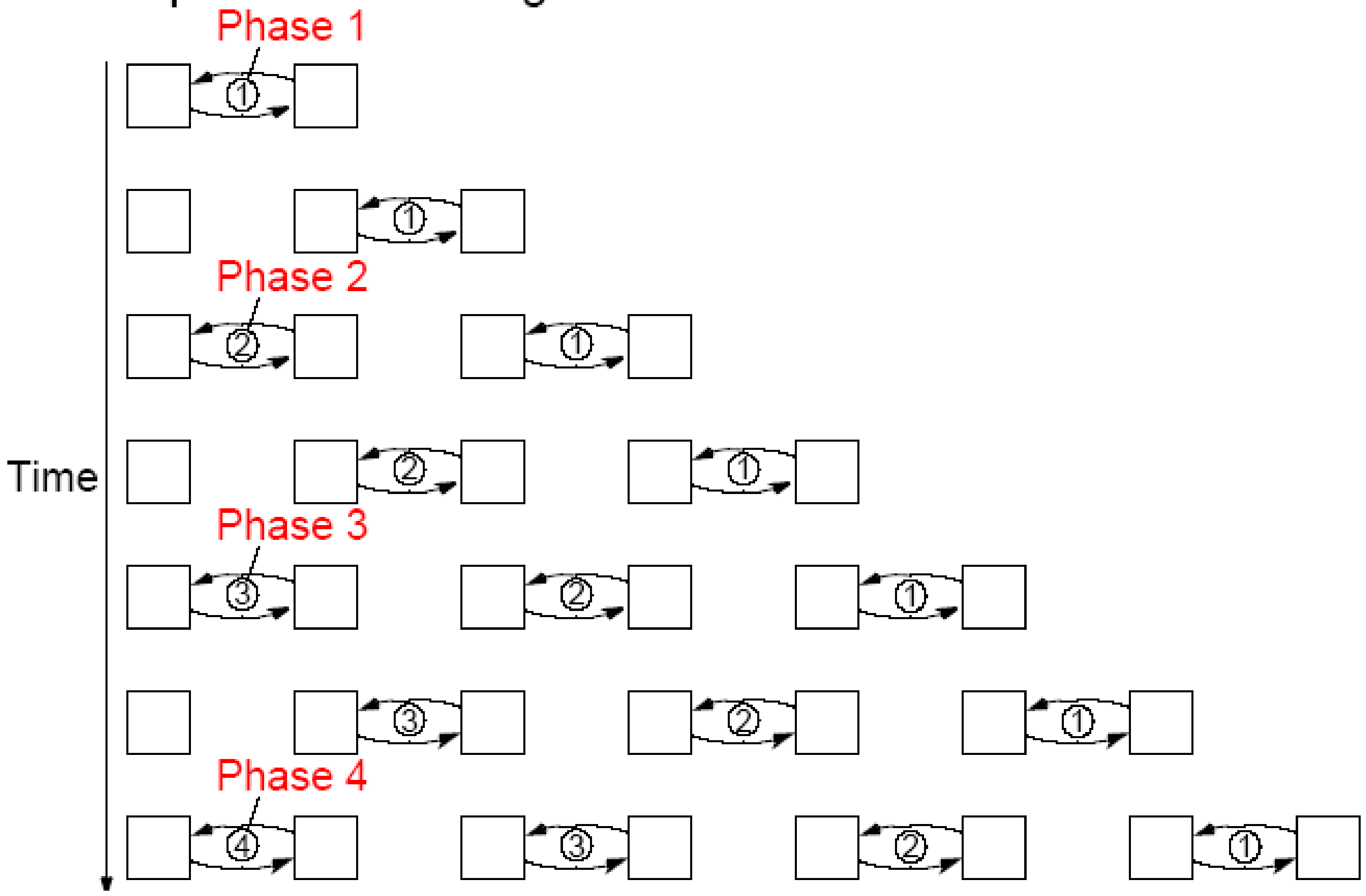
# Parallel Odd-Even Transposition

- The parallel formulation is cost-optimal for $p = O(\log n)$.
- The isoefficiency function of this parallel formulation is $\Theta(p\,2^p)$

(other slides on Parallel Odd-Even follow…)

## 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**

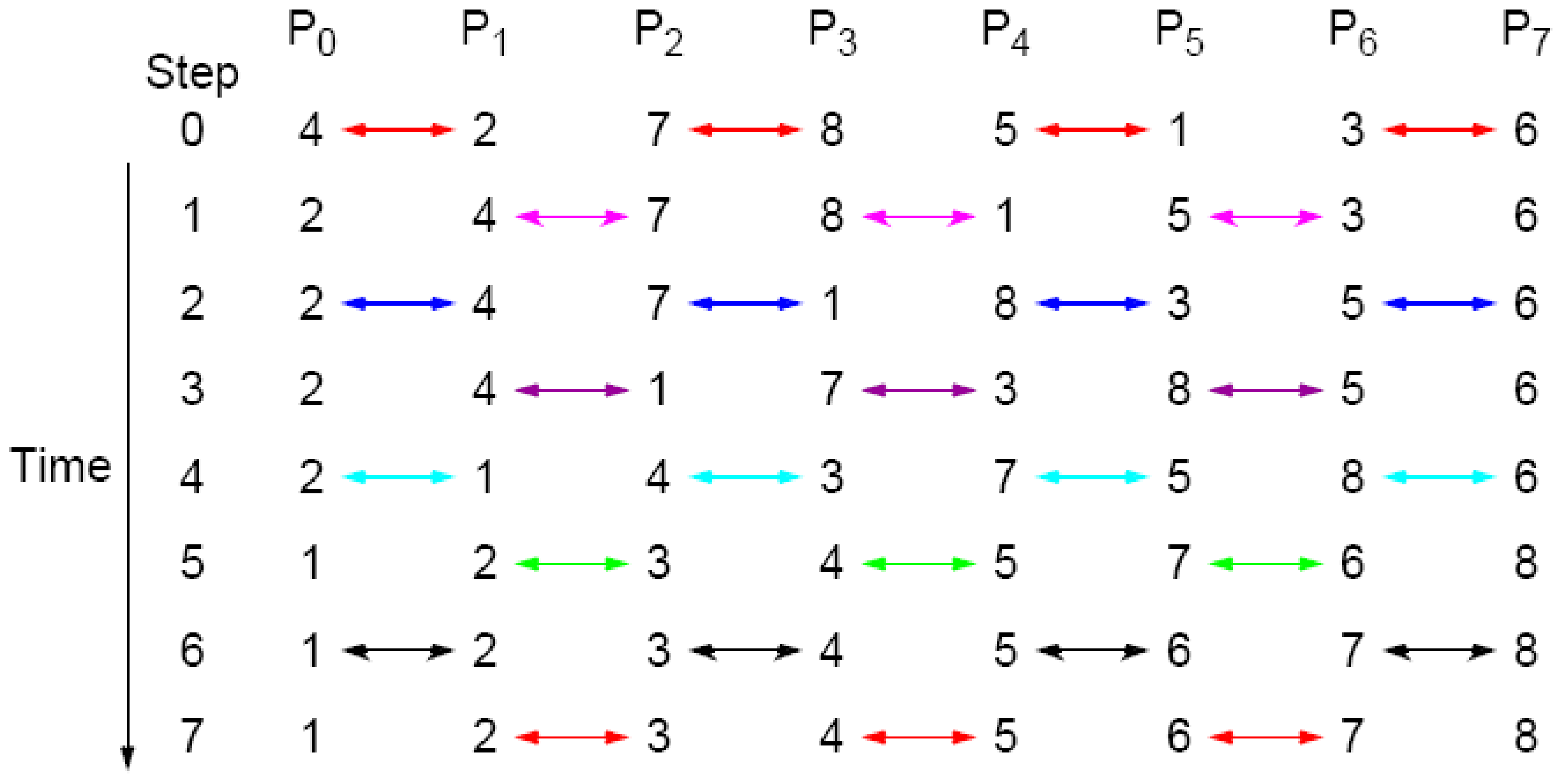
Even-numbered processes exchange numbers with their right neighbor.

**Odd phase**

Odd-numbered processes exchange numbers with their right neighbor.

# Odd-Even Transposition Sort

## Sorting eight numbers



| Step | $P_0$ | $P_1$ | $P_2$ | $P_3$ | $P_4$ | $P_5$ | $P_6$ | $P_7$ |
|---|---|---|---|---|---|---|---|---|
| 0 | 4 | 2 | 7 | 8 | 5 | 1 | 3 | 6 |
| 1 | 2 | 4 | 7 | 8 | 1 | 5 | 3 | 6 |
| 2 | 2 | 4 | 7 | 1 | 8 | 3 | 5 | 6 |
| 3 | 2 | 4 | 1 | 7 | 3 | 8 | 5 | 6 |
| 4 | 2 | 1 | 4 | 3 | 7 | 5 | 8 | 6 |
| 5 | 1 | 2 | 3 | 4 | 5 | 7 | 6 | 8 |
| 6 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| 7 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |

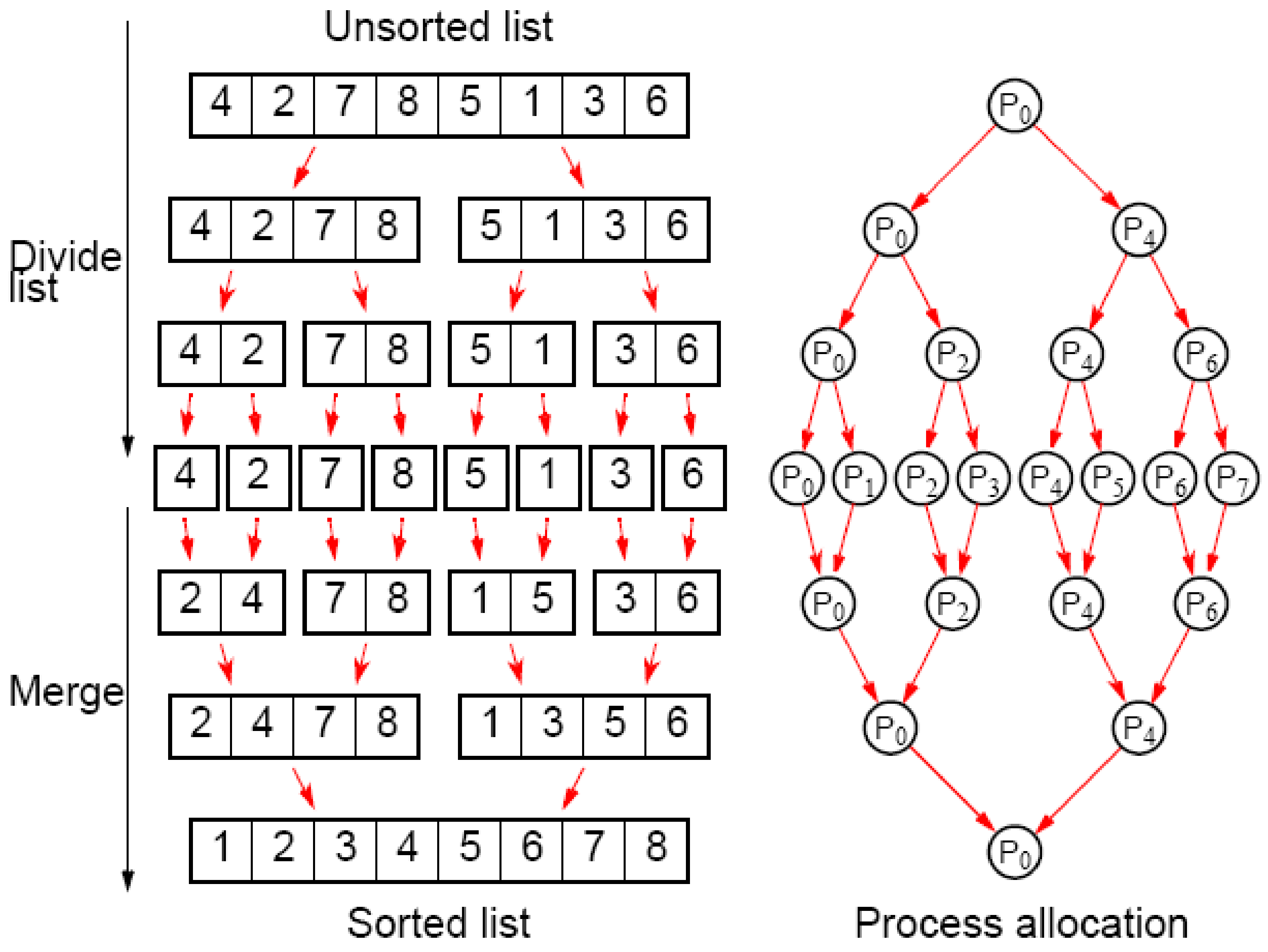
## Mergesort

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.

## 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

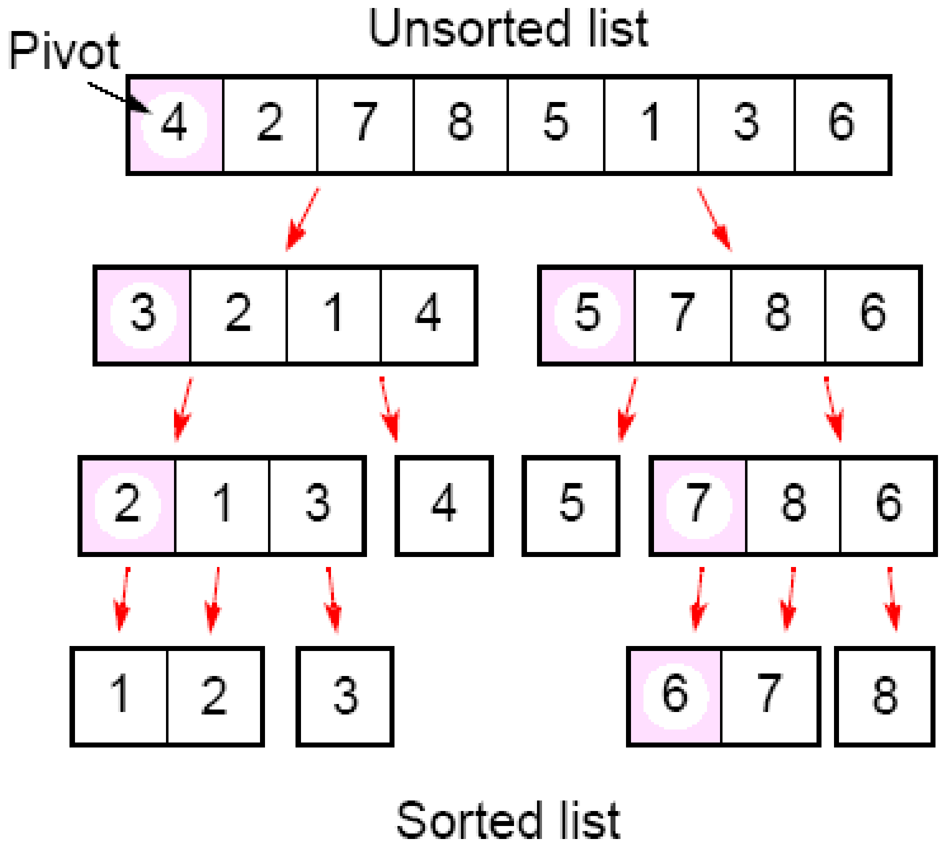
Very popular sequential sorting algorithm that performs well with an average sequential time complexity of $O(n \log n)$.

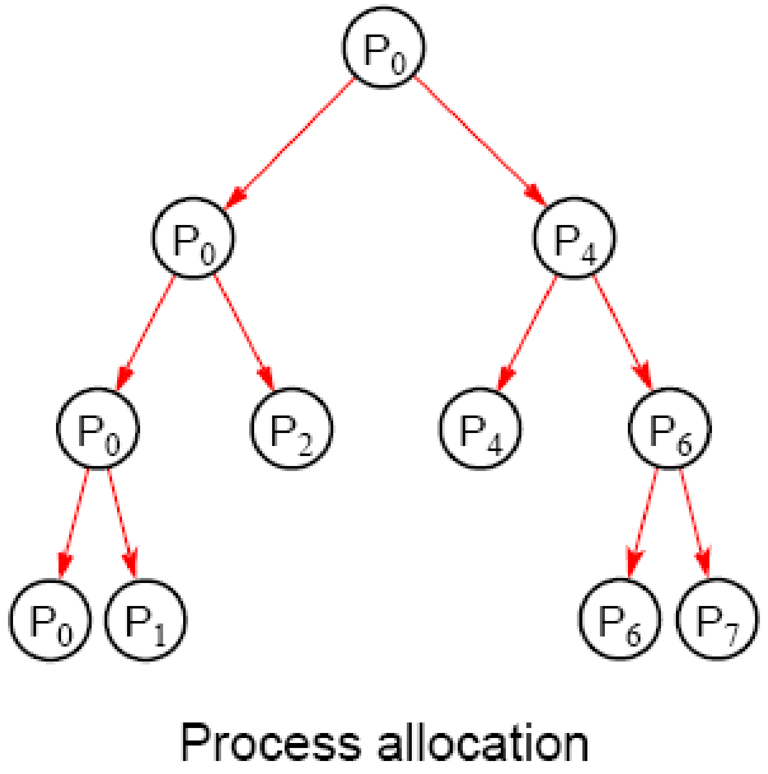
First list divided into two sublists. All the numbers in one sublist arranged to be smaller than all the numbers in the other sublist. Achieved by first selecting one number, called a *pivot*, against which every other number is compared. If the number is less than the pivot, it is placed in one sublist. Otherwise, it is placed in the other sublist.

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

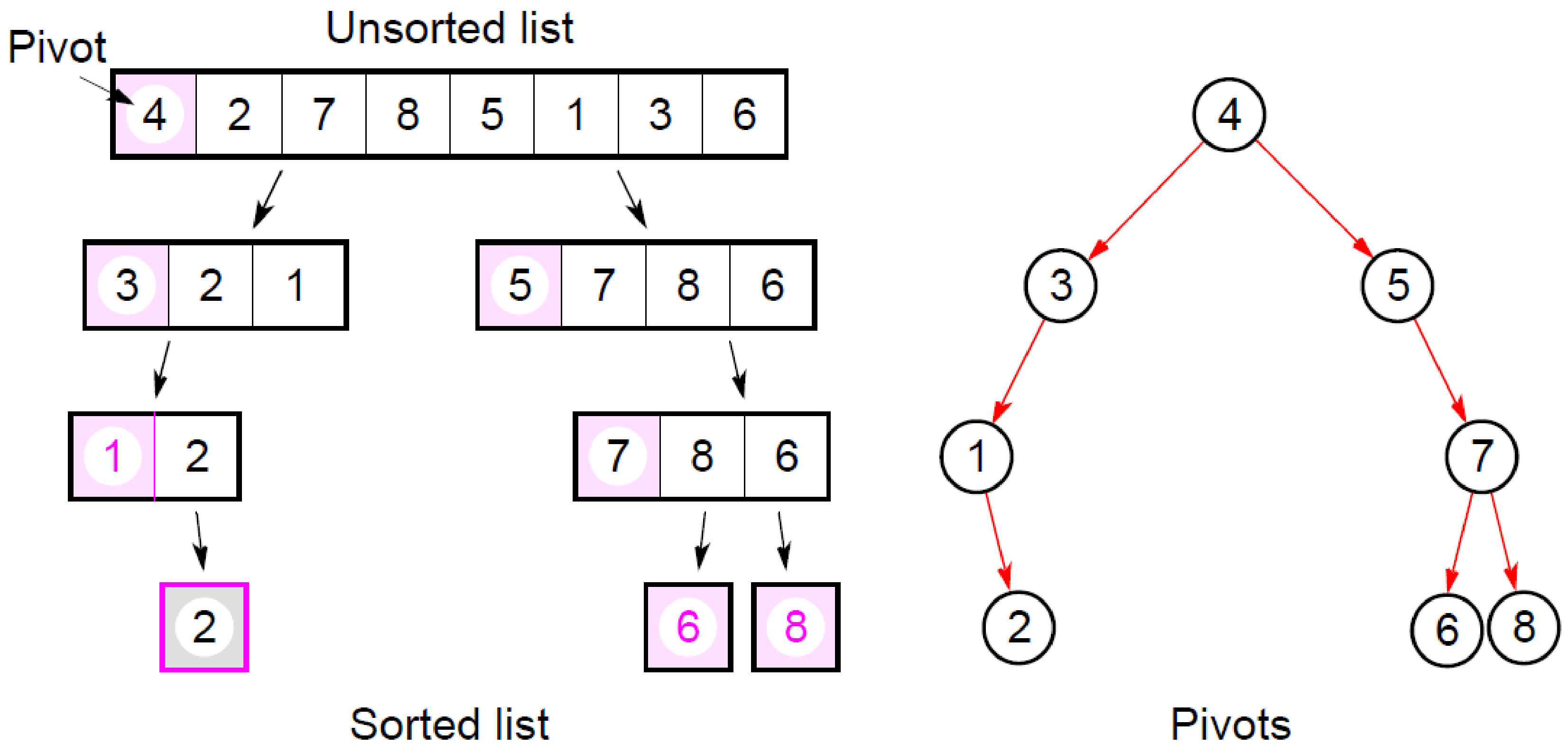
# Parallelizing Quicksort

Using tree allocation of processes





With the pivot being withheld in processes:



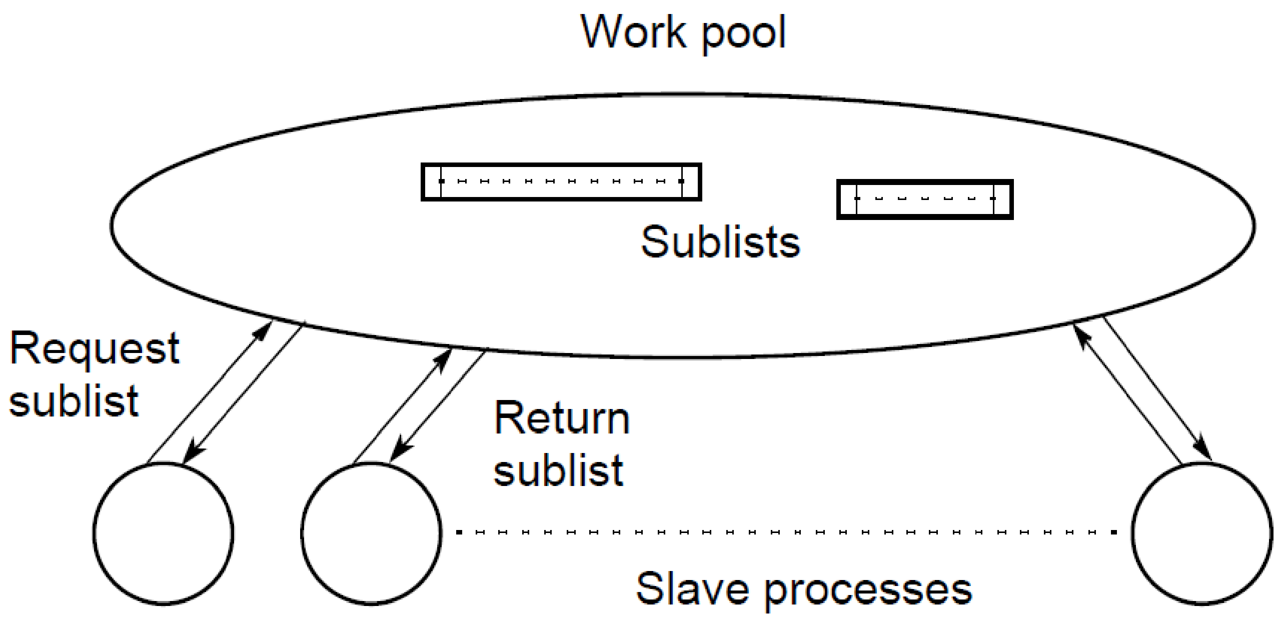
## Analysis

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.