Θέματα Κατανεμημένων και Παράλληλων Συστημάτων

Μάθημα #5

Αλγόριθμοι Κατανεμημένης Μνήμης

Shared address space (shared memory) programming

- Tasks operate and communicate via shared data, like bulletin boards

Message passing programming

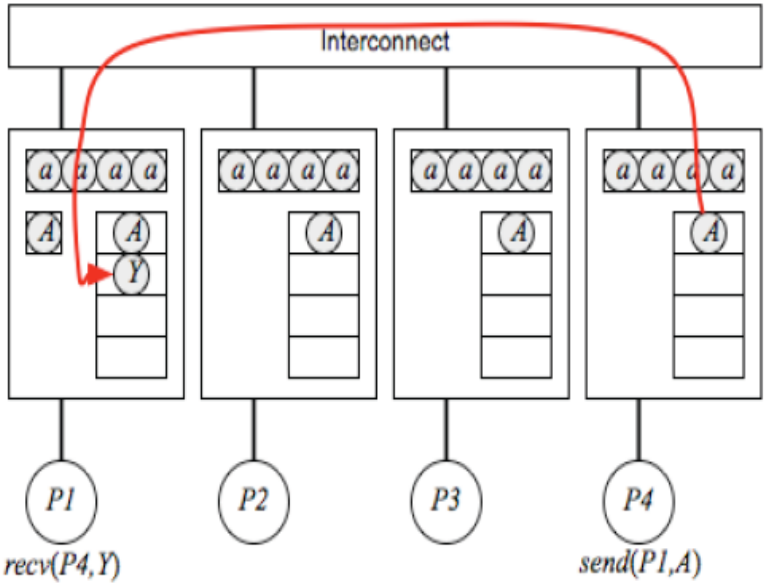
- Explicit point-to-point communication, like phone calls (connection oriented) or email (connectionless, mailbox posts)

Message Passing Programming

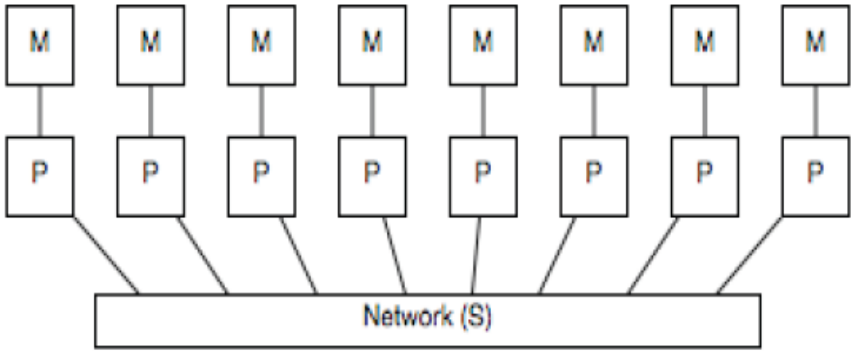
- *Message passing programming*
- Program is a set of *named* processes
  - Process has thread of control and local memory with local address space
- Processes communicate via explicit data transfers
  - Messages between source and destination, where source and destination are named processors P0…Pn (or compute nodes)
  - Logically shared data is explicitly partitioned over local memories
  - Communication with send/recv via standard message passing libraries, such as MPI and PVM

Message Passing Programming

- Message passing programming
- Each node has a network interface
  - Communication and synchronization via network
  - Message latency and bandwidth is dependent on network topology and routing algorithms



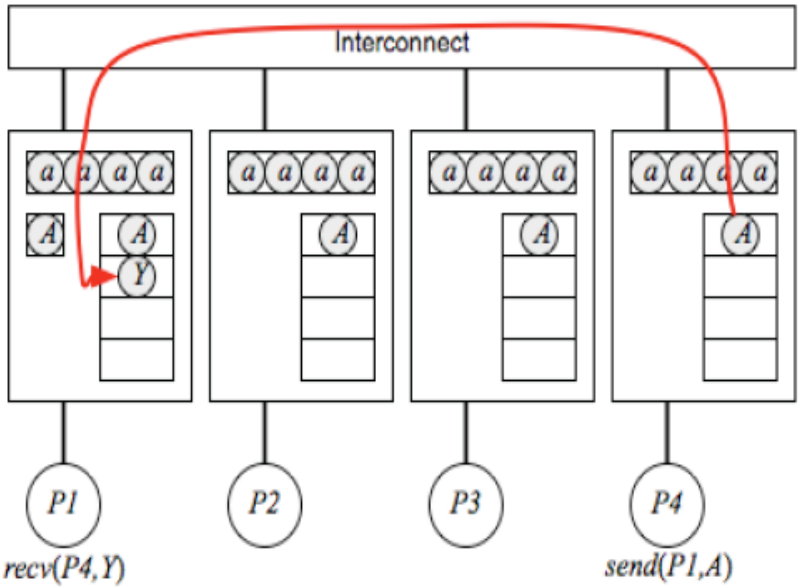
Programming model



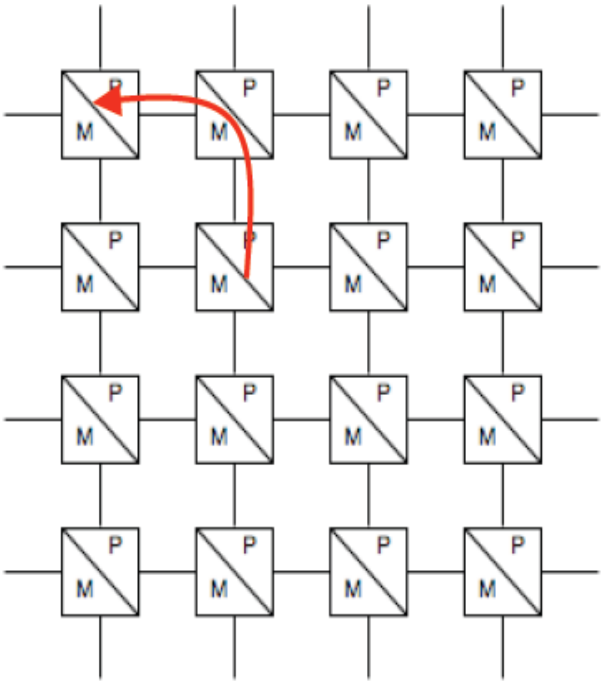
Machine model

Message Passing Programming

- Message passing programming
- Each node has a network interface
  - Communication and synchronization via network
  - Message latency and bandwidth is dependent on network topology and routing algorithms



Programming model

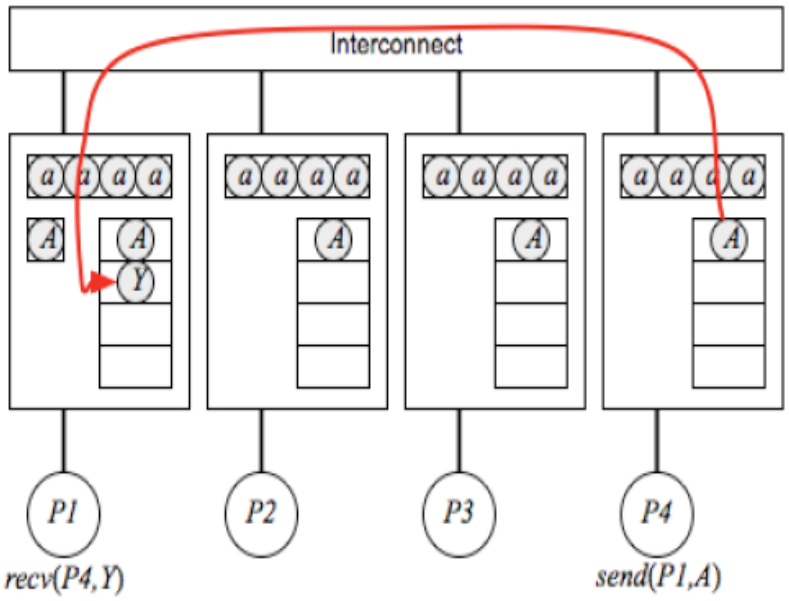


Machine model

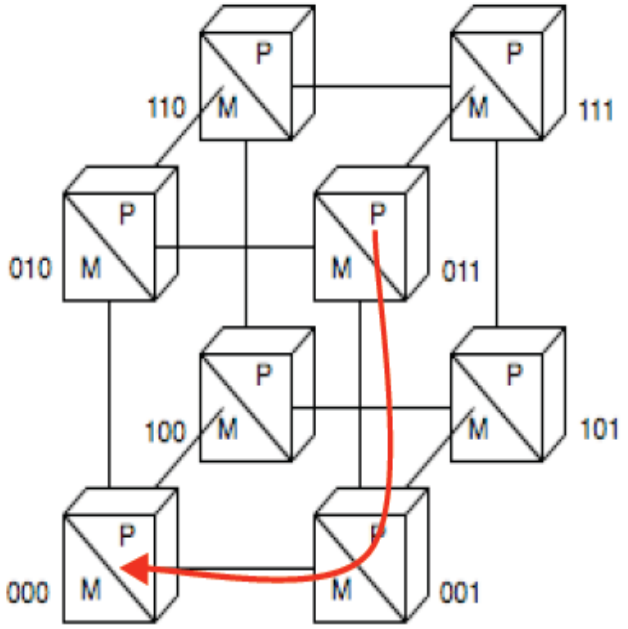
Message passing over mesh

Message Passing Programming

- Message passing programming
- Each node has a network interface
  - Communication and synchronization via network
  - Message latency and bandwidth is dependent on network topology and routing algorithms



Programming model

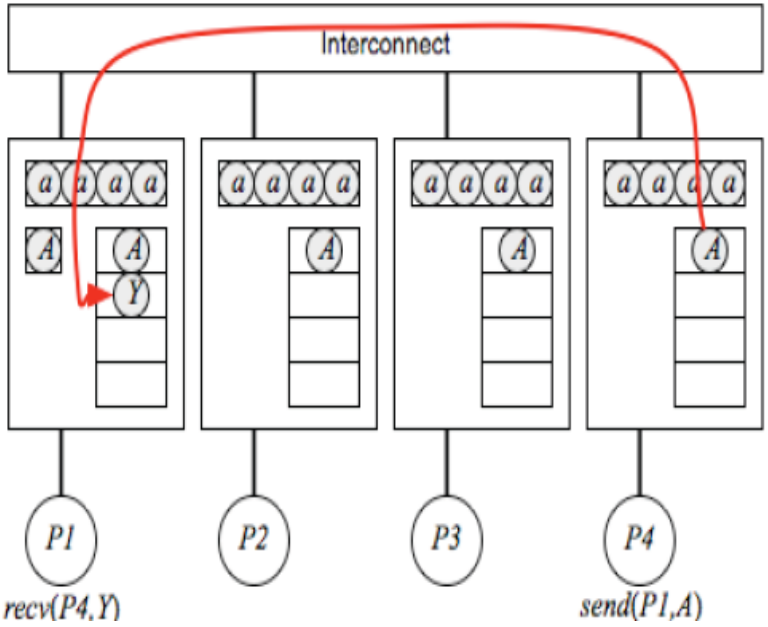


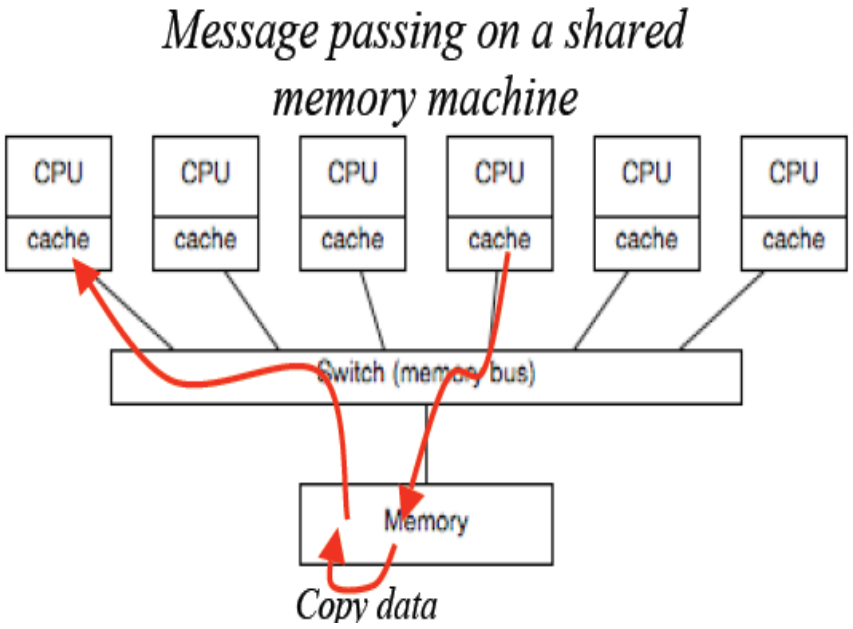
Machine model

Message passing over hypercube

Message Passing Programming

- Message passing programming
- On shared memory machine
  - Communication and synchronization via shared memory
  - Message passing library copies data (messages) in memory, less efficient (MPI call overhead) but portable

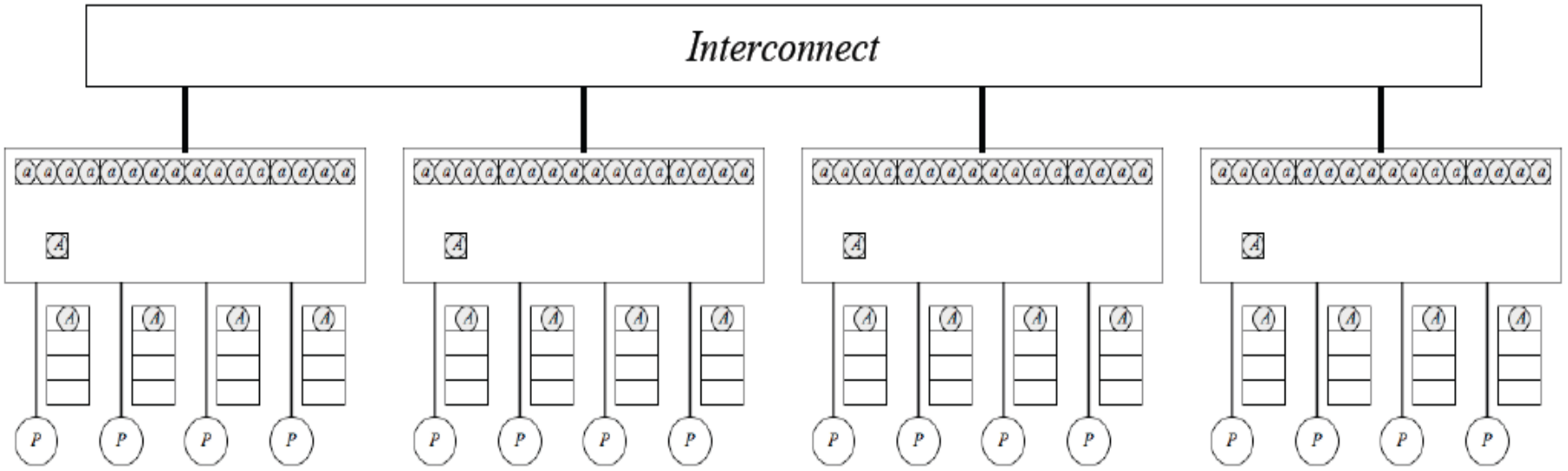




Hybrid Systems: Clusters of SMPs

- Shared memory within SMP, message passing outside
- Programming model with three choices:
  - Treat as "flat" system: always use message passing, even within an SMP
    - Advantage: ease of programming and portability
    - Disadvantage: ignores SMP memory hierarchy and advantage of UMA shared address space
  - Program in two layers: shared memory programming and message passing
    - Advantage: better performance (use UMA/NUMA intelligently)
    - Disadvantage: harder (and ugly!) to program

Hybrid Systems: Clusters of SMPs




```
shared a[1..N/numnodes]
private n = N/numnodes/numprocs
private x[1..n]
private lo = (pid-1)*n
private hi = lo+n
x[1..n] = f(a[lo..hi])
A[pid] := sum(x[1..n])
send A[pid] to node1
```

Shared part

Processor-local part

Vector (SIMD) part

```
A := 0
if node=1 and pid=1
  for j = 1..numnodes
    for i = 1..numprocs
      receive Aj from node(j)
      A := A + Aj
```

Extra code for node 1 proc 1

Αλγόριθμοι Κατανεμημένης Μνήμης

Βασικές Δομές:

send(X,Pi)

Ο επεξεργαστής που εκτελεί την εντολή send: **στέλνει ένα αντίγραφο της μεταβλητής *X* στον επεξεργαστή με κωδικό *Pi* και συνεχίζει αμέσως με την εκτέλεση της επόμενης εντολής.**

receive(Y,Pj)

Ο επεξεργαστής που εκτελεί την εντολή receive: **αναστέλλει την εκτέλεση του προγράμματος, έως ότου παραλάβει δεδομένα από τον *Pj*.**

Όταν παραλάβει τα δεδομένα, τα αποθηκεύει στη μεταβλητή *Y* και συνεχίζει με την επόμενη εντολή.

Γινόμενο πίνακα-διανύσματος σε δακτύλιο

A πίνακας $n \times n$,
x διάνυσμα διάστασης n.

Υπολογίσουμε $y=Ax$ σε δακτύλιο p επεξεργαστών, $p \leq n$.

- **Υποθέτουμε ότι το p διαιρεί ακριβώς το n, και *ότι* $k = n/p$.**

`Εστω $A=(A_1,A_2,\ldots,A_p)$, $x=(x_1, x_2,\ldots, x_p)$, όπου A_i πίνακας $n \times k$, και x_i διάνυσμα διάστασης k.

Υπολογίζουμε το γινόμενο Ax ως εξής:

***(1)* Υπολογίζουμε τα διανύσματα $z_1 = A_1x_1$, $z_2 = A_2x_2$, …, $z_p = A_px_p$.**
***(2)* Υπολογίζουμε το άθροισμα $z_1+z_2+\ldots+z_p$.**

Αλγόριθμος ΓΙΝΟΜΕΝΟ ΠΙΝΑΚΩΝ ΣΕ ΔΑΚΤΥΛΙΟ

Είσοδος:

(1) Το *i* **του επεξεργαστή.**

(2) Ο αριθμός *p* **των επεξεργαστών.**

(3) ***i*-οστός υποπίνακας** $B = A_i$ **διαστάσεων** $n \times k$

(4) **Το** ***i*-οστό υπόδιάνυσμα** $w = x_i$ **διάστασης** k.

`Εξοδος: **Το γινόμενο** Ax **αποθηκευμένο στον** P_1.

begin

1. Υπολόγισε το διάνυσμα $z = Bw$

2. if $(i = 1)$ **then** $y := 0$

else receive$(y, left\_processor)$

3. $y := y + z$

4. send$(y, right\_processor)$

5. if $(i = 1)$ **then receive**$(y, left\_processor)$

end

Πολυπλοκότητα Αλγορίθμου

- **Κάθε επεξεργαστής για την εκτέλεση των βημάτων 1 και 3 χρειάζεται χρόνο $T_c=O(n^2/p)$.**

- **Για την μεταφορά n αριθμών από έναν επεξεργαστή σε έναν άλλον απαιτείται χρόνος $s + r\,n$, όπου s ο χρόνος εκκίνησης για την μεταφορά μηνύματος και r ο ρυθμός μεταφοράς.**

Ο συνολικός χρόνος επικοινωνίας: $T_m = p(s + rn)$.

Ο συνολικός χρόνος του αλγορίθμου:

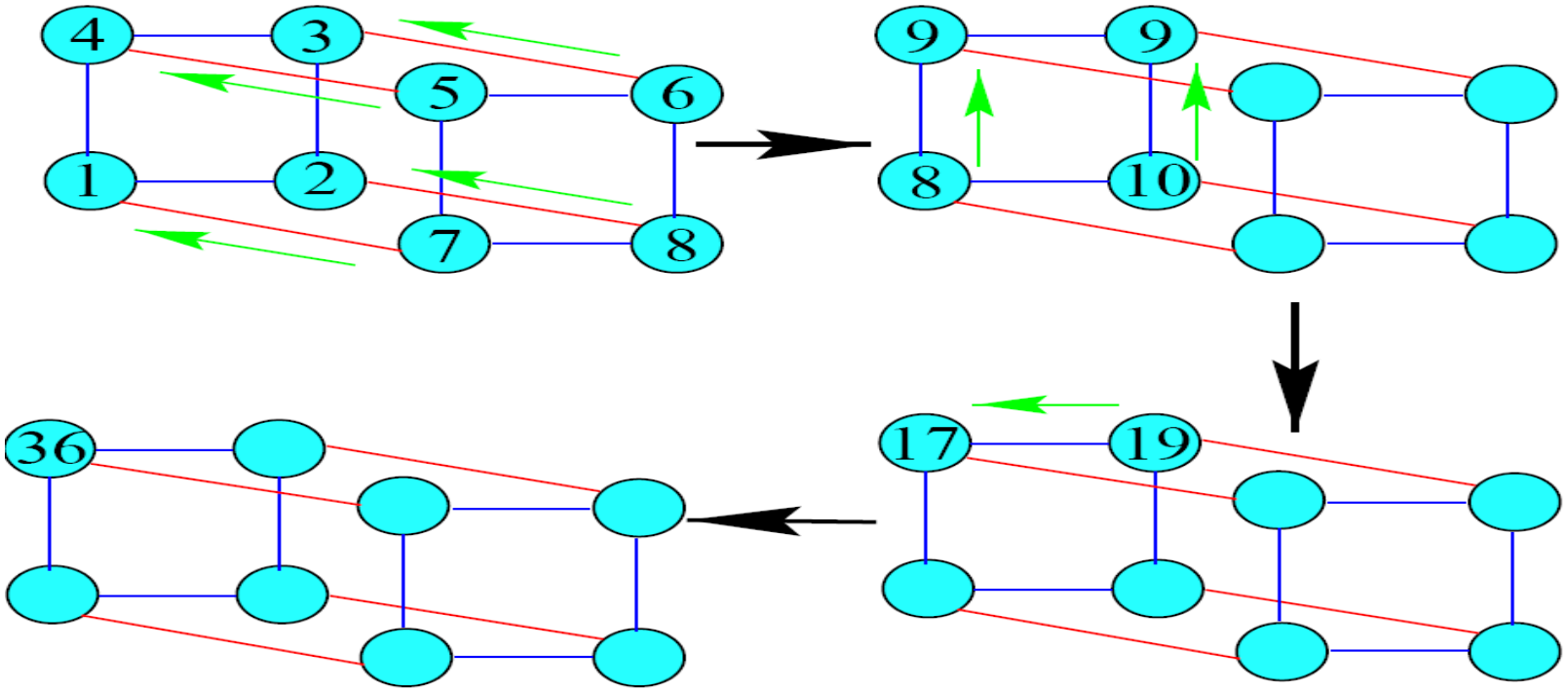
$$T_c+T_m = c(n^2/p) + p(s+rn), \text{ όπου } c \text{ σταθερά}$$

Πολυπλοκότητα Αλγορίθμου

Συνολικός χρόνος του αλγορίθμου: $O(n^2/p+pn)$

Επομένως, βέλτιστος χρόνος όταν $p=O(n^{1/2})$

Πρόσθεση στον Υπερκύβο



Η πρόσθεση *n* αριθμών απαιτεί *O(log n)* βήματα

Πρόσθεση στον Υπερκύβο

Αλγόριθμος υπολογισμού του αθροίσματος των στοιχείων ενός διανύσματος *A* σε δίκτυο υπερκύβου.

Ο αλγόριθμος εκτελείται από κάθε επεξεργαστή P_i με δυαδικό κωδικό i, $0 \leq i \leq n-1$.

Με $i^{(k)}$ δηλώνεται ο δυαδικός αριθμός i του οποίου το bit στη θέση k έχει αντικατασταθεί με το συμπληρωματικό του.

Π.χ., αν $i=10010$, τότε $i^{(0)} =10011$ και $i^{(4)} =00010$.

Πρόσθεση στον Υπερκύβο διάστασης *d*

Είσοδος: Διάνυσμα *A[0..n-1]* $n=2^d$ αριθμών. Το *A(i)* είναι αποθηκευμένο στην τοπική μνήμη του επεξεργαστή με δυαδικό κωδικό *i.*
Έξοδος: Το άθροισμα των στοιχείων του *A[0..n-1]*, αποθηκευμένο στον P_0.

Αλγόριθμος για τον επεξεργαστή P_i με δυαδικό κωδικό *i*
begin

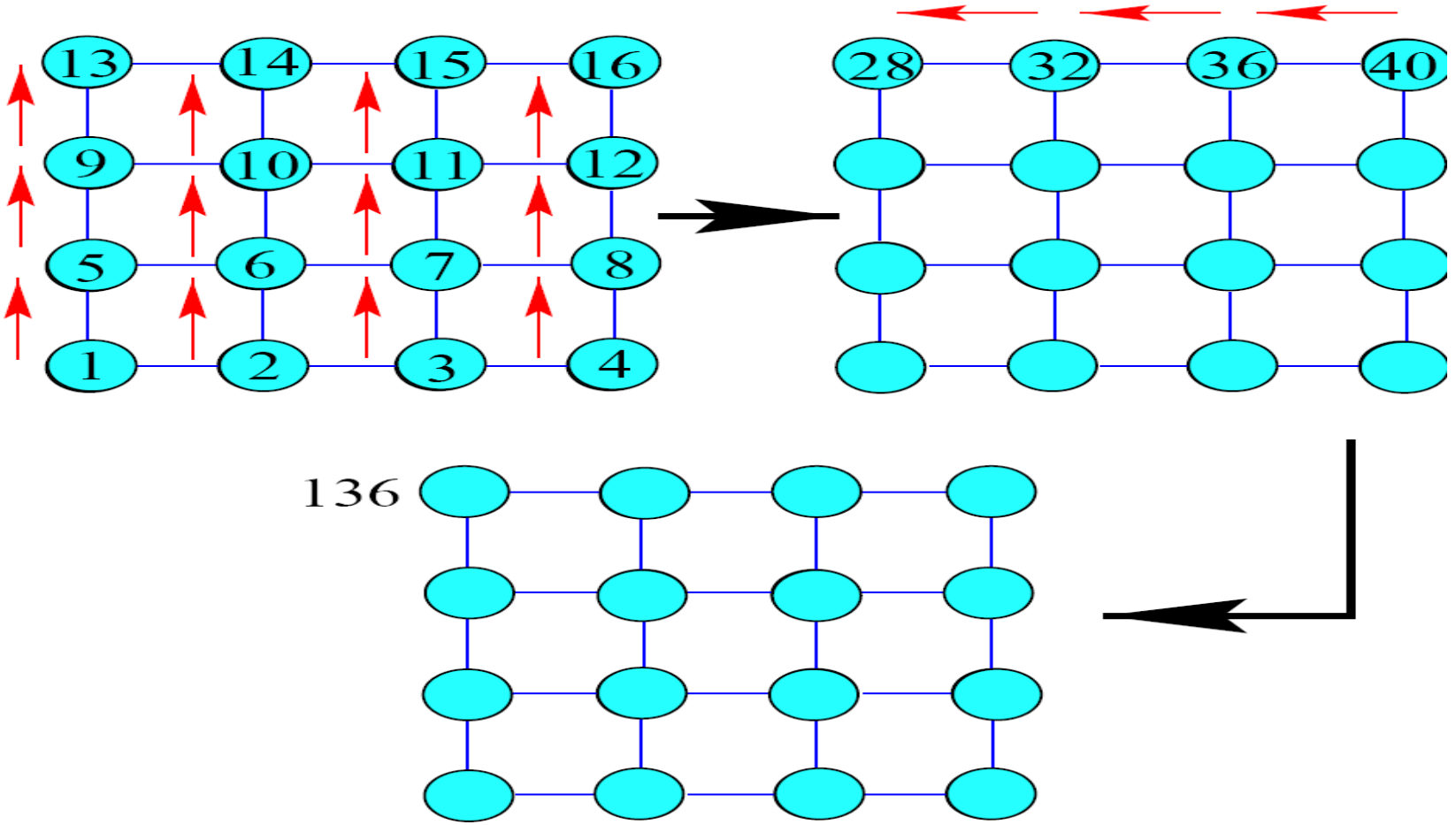
1. if $2^{d-1} \le i \le 2^d - 1$ **then**
send$(A(i), i^{(d-1)})$
exit

2. for $k := d-1$ **to** 1 **do**
2.1 if $0 \le i \le 2^k - 1$ **then**
receive$(X, i^{(k)})$
$Y := X + A(i)$
2.2 if $2^{k-1} \le i \le 2^k - 1$
send$(Y, i^{(k-1)})$

3. if $i=0$ **then**
receive$(X, i^{(0)})$
$Y := X + A(i)$

end

Πρόσθεση στο 2-D Πλέγμα



Η πρόσθεση *n* αριθμών απαιτεί $O(n^{1/2})$ βήματα

Παράλληλη Ταξινόμηση (Parallel Sorting)

Ο(n log n) είναι η πολυπλοκότητα του βέλτιστου ακολουθιακού αλγορίθμου ταξινόμησης n αριθμών.

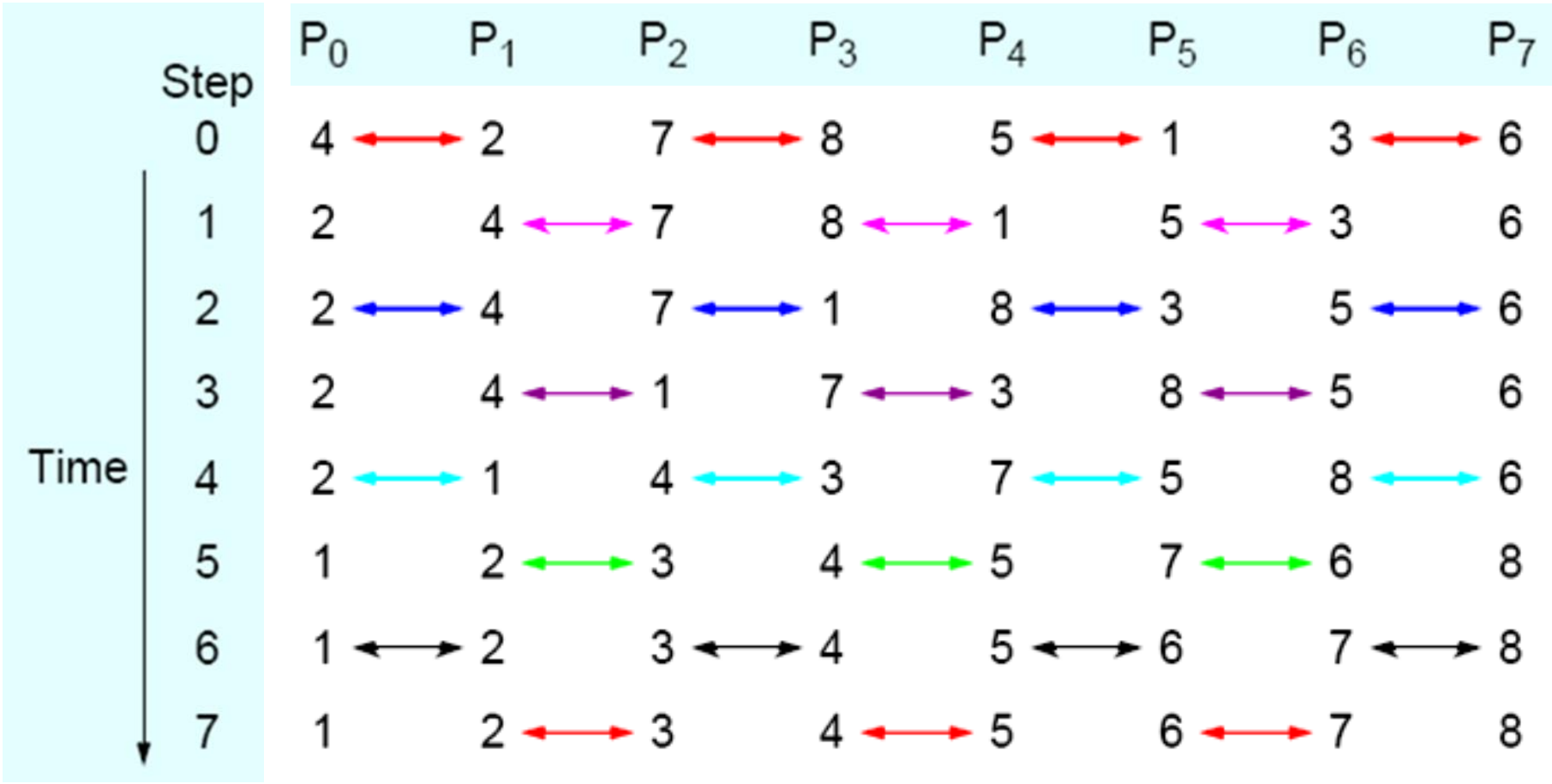
Επομένως, με n επεξεργαστές:

Odd-Even Transposition Sort
Ταξινόμηση περιττής-άρτιας μετάθεσης

n = p **επεξεργαστές** σε **γραμμικό δίκτυο**

- Η συνολική εκτέλεση οργανώνεται αρχικά σε *n / 2* άρτιες και *n / 2* περιττές φάσεις οι οποίες εναλλάσσονται συγχρονισμένα μεταξύ τους δηλαδή, πρώτα εκτελείται μία άρτια φάση, μετά μία περιττή, μετά μία άρτια κ.ο.κ.
- ***n*** στάδια, κάθε στάδιο ***O(1)*** βήματα και ***O(n)*** έργο
- Συνολικός παράλληλος χρόνος ***T* = O(*n*)** και συνολικό έργο ***O(n²)***: μη βέλτιστος αλγόριθμος

Odd-Even Transposition Sort - Παράδειγμα



21

Odd-Even Transposition Sort *(n=p)*

for *i:=0* **to** *p-1* **do**

begin

if *(i mod 2 = 0)* **then**

if *(k mod 2 = 0)* **then** *<Σύγκριση-Εναλλαγή με* P_{k+1}*>;*

else *<Σύγκριση-Εναλλαγή με* P_{k-1}*>;*

else if *(k mod 2 <> 0* **and** *k <> p-1)* **then**

<Σύγκριση-Εναλλαγή με P_{k+1}*>;*

else if *(k mod 2 = 0* **and** *k <> 0)* **then**

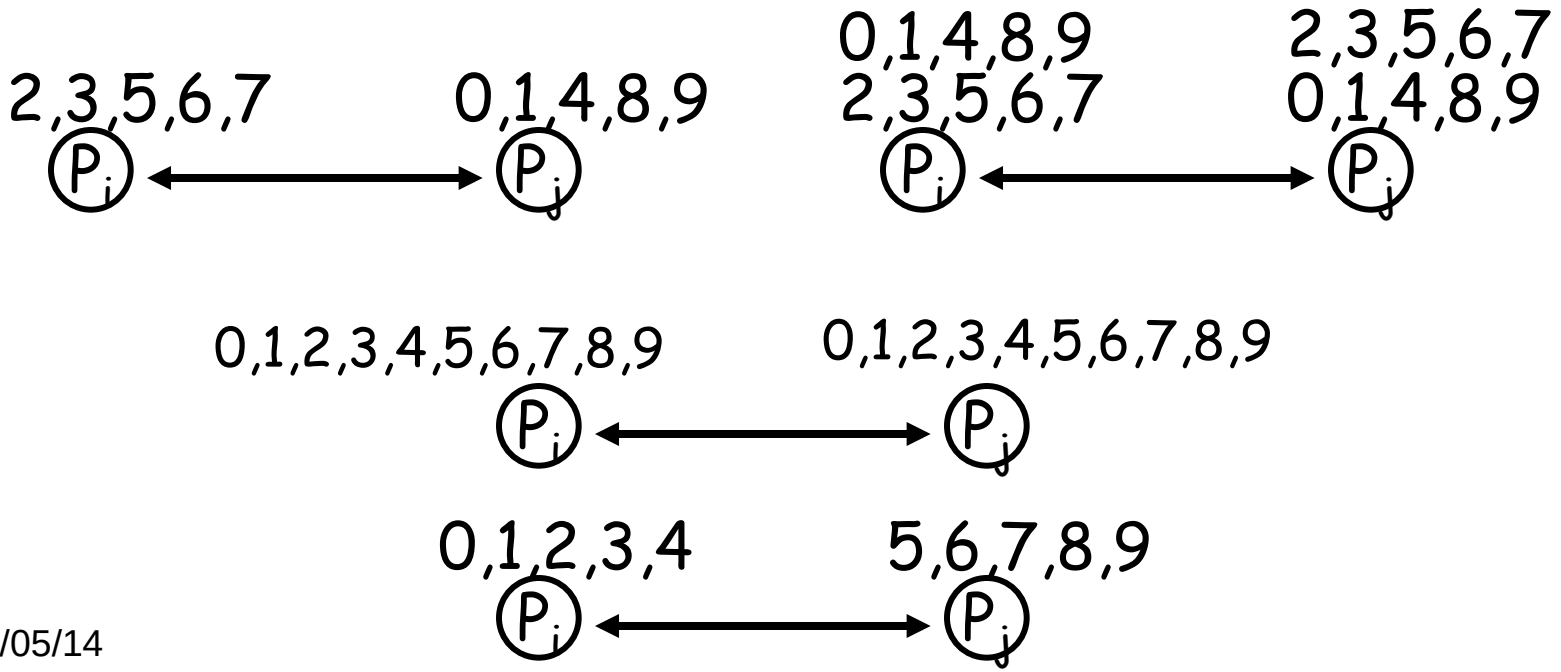
<Σύγκριση-Εναλλαγή με P_{k-1}*>;*

end

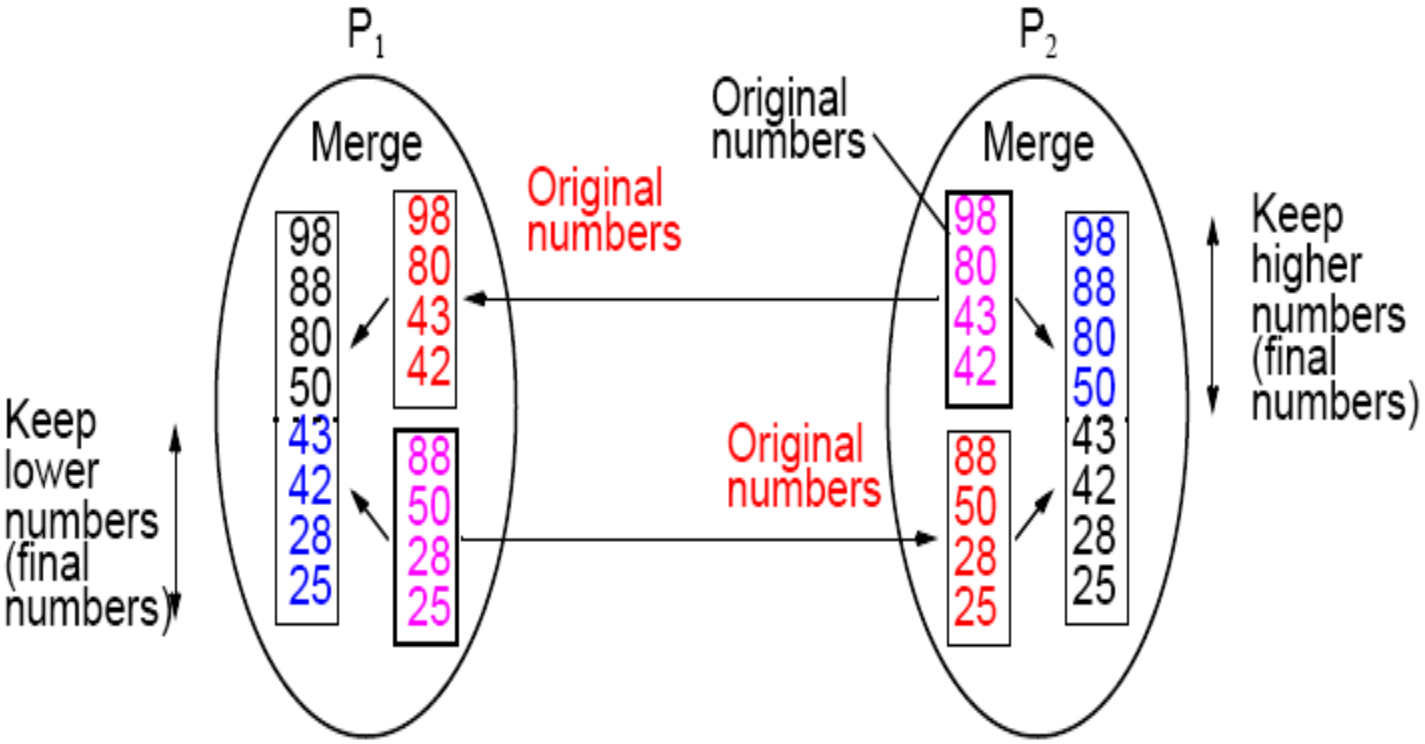
Exchange - Merge - Split

Exchange - Merge – Split δύο ταξινομημένων λιστών που βρίσκονται σε διαφορετικούς επεξεργαστές

Η Πολυπλοκότητα για Exchange - Merge – Split *k* αριθμών είναι *O(k)*

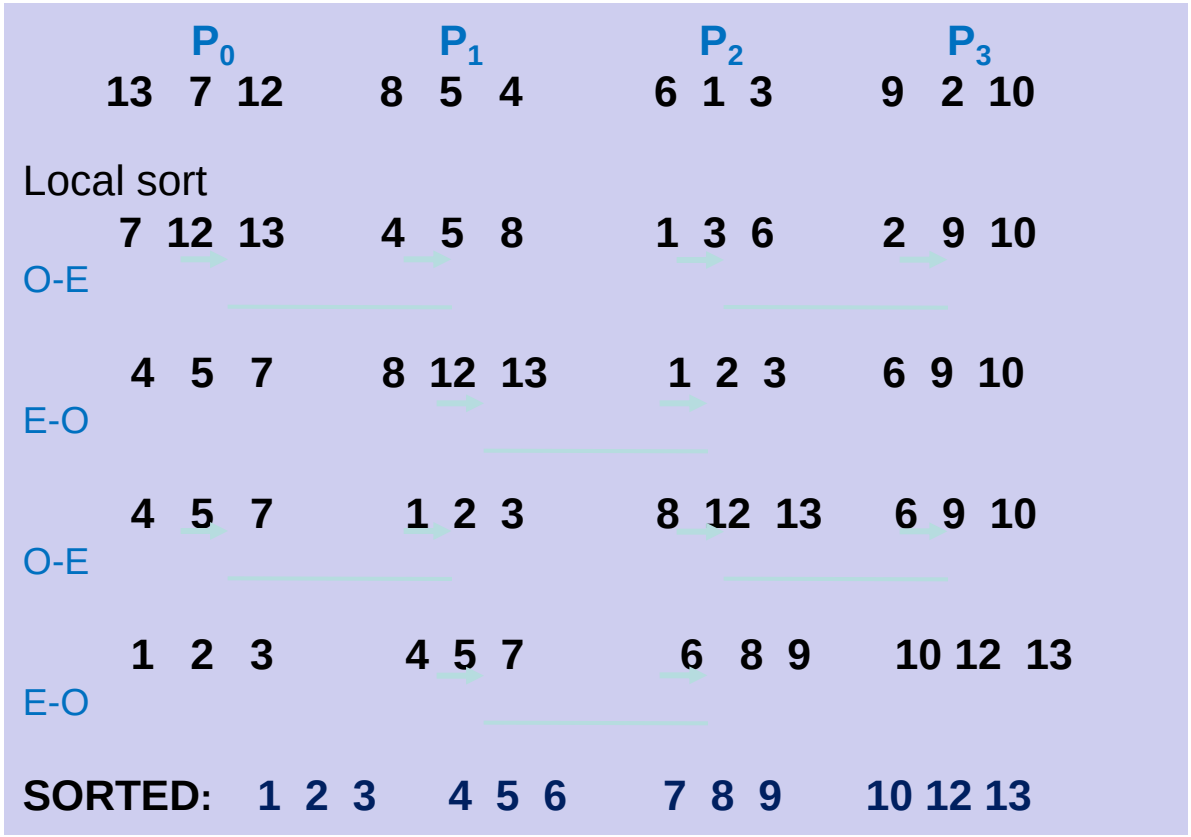


Exchange - Merge - Split



Odd-Even Transposition Sort (n >> p)

Κάθε επεξεργαστής ταξινομεί τοπικά *n/p* αριθμούς. Καλείται ο odd-even αλγόριθμος σε κάθε βήμα του οποίου, όμως, μεταξύ δύο επεξεργαστών γίνεται exchange-merge-split *2n/p* αριθμών.



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

T_{par} = (n/p)log(n/p) + 2n/p + p(n/p) + p(n/p) = (n/p)log(n/p) + cn, c σταθερά

Shearsort - Ταξινόμηση στο 2-D Πλέγμα

Είσοδος: μη ταξινομημένος πίνακας διάστασης *n*x*n*

Έξοδος: ο πίνακας ταξινομημένος σε διάταξη «φιδιού»

Μέθοδος:

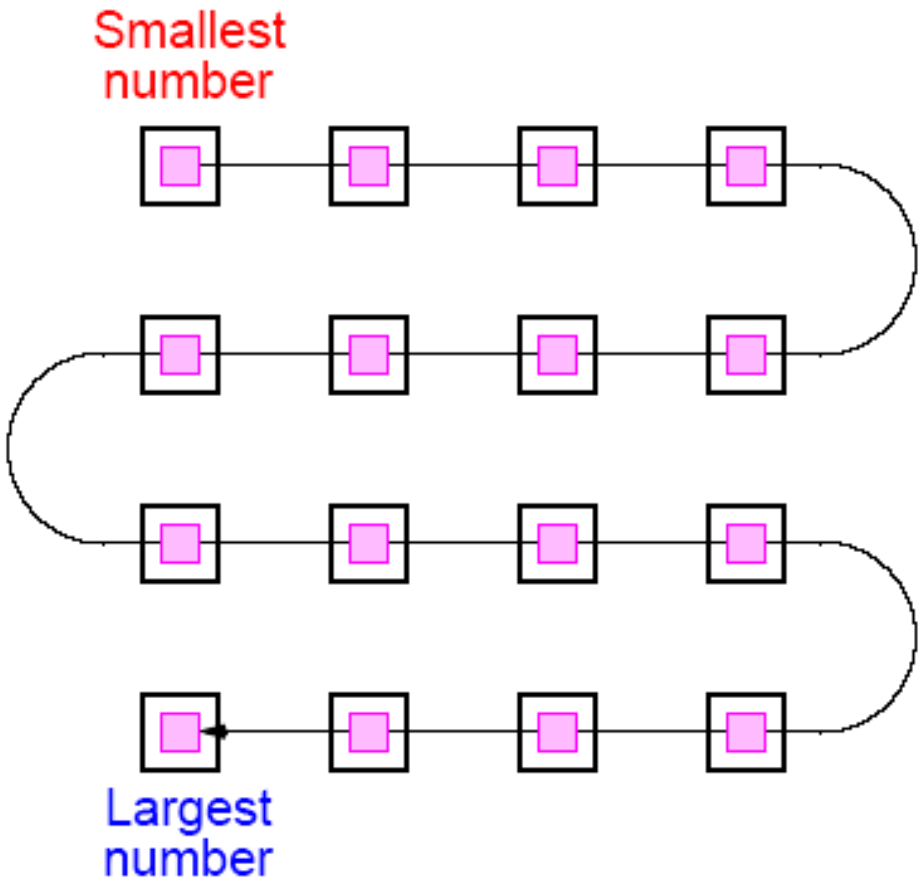
Επανέλαβε log *n* φορές

- Ταξινόμησε τις γραμμές (σε αντίθετη κατεύθυνση από γραμμή σε γραμμή);
- Ταξινόμησε τις στήλες;

Ταξινόμησε τις γραμμές (κατά την επιθυμητή κατεύθυνση)

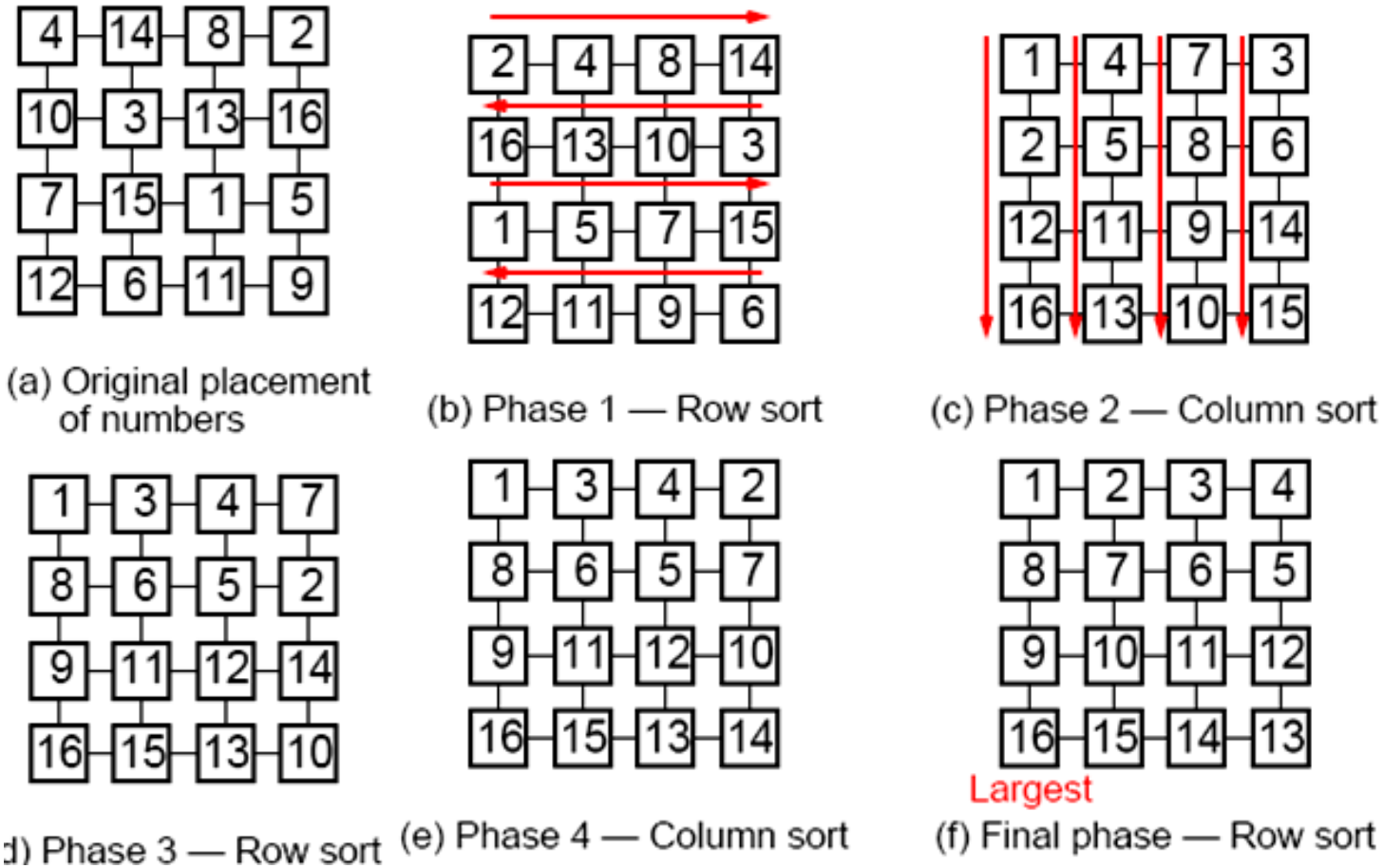
Ταξινόμηση στο 2-D Πλέγμα

Η ταξινομημένη ακολουθία αριθμών στο πλέγμα:



Shearsort - Ταξινόμηση στο 2-D Πλέγμα

Εναλλαγή ταξινόμησης στις σειρές και στις στήλες του πλέγματος ως εξής:



Shearsort - Correctness

Correctness

- The correctness of shearsort is shown using the 0-1-principle.
- The 0-1-principle states the following: If a comparator network sorts every sequence of 0's and 1's, then it sorts every sequence of arbitrary values. Without loss of generality, we may therefore restrict the proof to input sequences consisting of 0's and 1's.

Definition: A row is called clean, if it consists of 0's only or 1's only, otherwise it is called dirty.

Shearsort - Correctness

- After sorting the rows in the first step, there are $n/2$ rows that are sorted from left to right and $n/2$ rows that are sorted from right to left.
- When sorting the columns, every two of these rows are combined to at least one clean row.
- Thus, after the first iteration the array consist of some clean 0-rows, some clean 1-rows and an unsorted zone in between consisting of at most $n/2$ rows.

- In the second iteration, every two rows of the unsorted zone are again combined to at least one clean row. Thus, after the second iteration the unsorted zone consists of at most $n/4$ rows, and so on.

- After $\log(n)$ steps, there is at most one dirty row. In the additional last step this row is sorted, and the whole array is sorted.

Shearsort - Correctness

1	1	0	1
1	0	1	1
1	0	1	0
0	1	1	0

0	1	1	1
1	1	1	0
0	0	1	1
1	1	0	0

0	0	0	0
0	1	1	0
1	1	1	1
1	1	1	1

0	0	0	0
1	1	0	0
1	1	1	1
1	1	1	1

0	0	0	0
1	1	0	0
1	1	1	1
1	1	1	1

0	0	0	0
0	0	1	1
1	1	1	1
1	1	1	1

Shearsort - Correctness

1	1	0	1
1	0	1	1
1	0	1	0
0	0	0	0

0	1	1	1
1	1	1	0
0	0	1	1
0	0	0	0

0	0	0	0
0	0	1	0
0	1	1	1
1	1	1	1

0	0	0	0
1	0	0	0
0	1	1	1
1	1	1	1

0	0	0	0
0	0	0	0
1	1	1	1
1	1	1	1

0	0	0	0
0	0	0	0
1	1	1	1
1	1	1	1

Shearsort – Πολυπλοκότητα

Σε $n \times n$ πλέγμα, απαιτούνται $\log n$ επαναλήψεις για την ταξινόμηση n^2 αριθμών. Επομένως η συνολική πολυπλοκότητα είναι $O(n \log n)$

Άρα,
σε πλέγμα $\sqrt{n} \times \sqrt{n}$ η ταξινόμηση n αριθμών, η συνολική πολυπλοκότητα είναι $O(\sqrt{n} \log n)$

Hypercube

Quicksort

Hypercube network has structural characteristics that offer scope for implementing efficient divide-and-conquer sorting algorithms, such as quicksort.

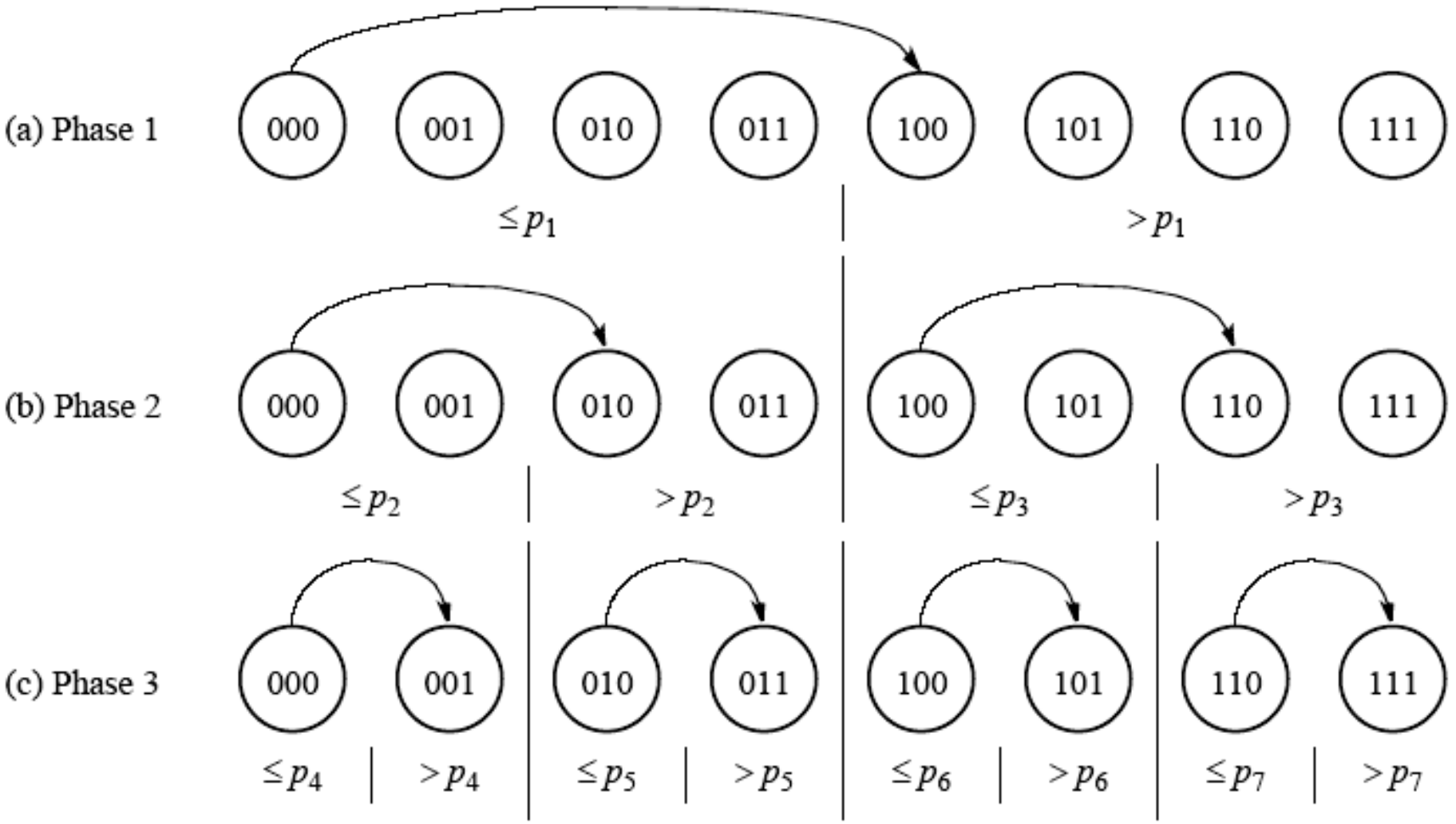
Περίπτωση 1: Η λίστα των αριθμών βρίσκεται σε έναν επεξεργαστή του υπερκύβου

List is divided into two parts according to the quicksort algorithm by using a pivot determined by the processor, with one part sent to the adjacent node in the highest dimension.

Then the two nodes can repeat the process dividing their lists into two parts using locally selected pivots.

Finally, the parts are sorted using a sequential algorithm, all in parallel. If required, sorted parts can be returned to one processor in a sequence that allows processor to concatenate sorted lists to create final sorted list.

Hypercube quicksort algorithm - Numbers originally in node 000



Hypercube quicksort algorithm

- Numbers initially distributed across all processors

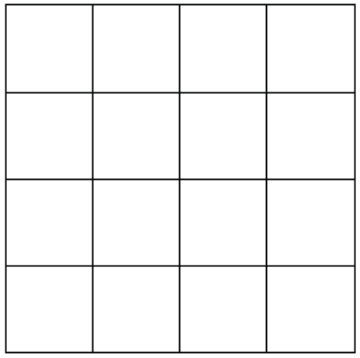
The algorithm is as follows:

1. One processor (say *0…0*) of the *d*-dimensional cube computes a pivot and broadcasts this to all others in the cube
2. The processors in the lower subcube send their numbers, which are greater than the pivot, to their partner processor in the upper subcube. The processors in the upper subcube send their numbers, which are less than or equal to the pivot, to their partner processor in the lower subcube.
3. Each processor concatenates the list received with what remains of its own list.

The process is repeated recursively on the two (*d-1*)-dimensional subcubes.

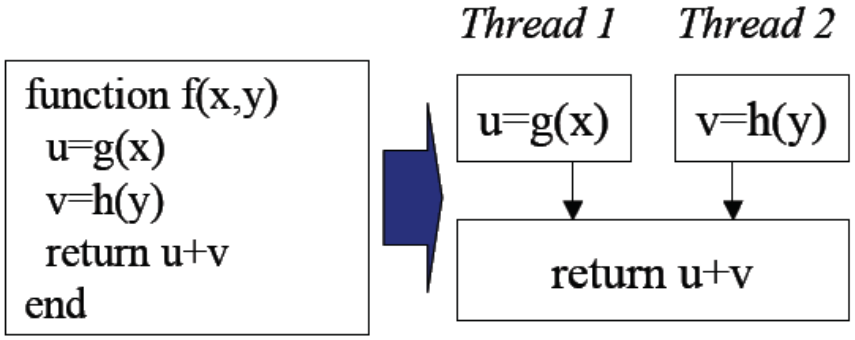
Finally, the numbers in each processor are sorted sequentially.

Data vs. Task Partitioning



Block partitioning of a 2D domain

- *Data partitioning*
  - Perform *domain decomposition* to run parallel tasks on subdomains
  - "Scatter-compute-gather" where local computation may require communication and scatter/gather may involve computations



- *Task partitioning*
  - Decompose functions into independent subfunctions and execute the subfunctions in parallel

Partitioning/Divide and Conquer (Διαμέριση/ Διαίρει και Βασίλευε)

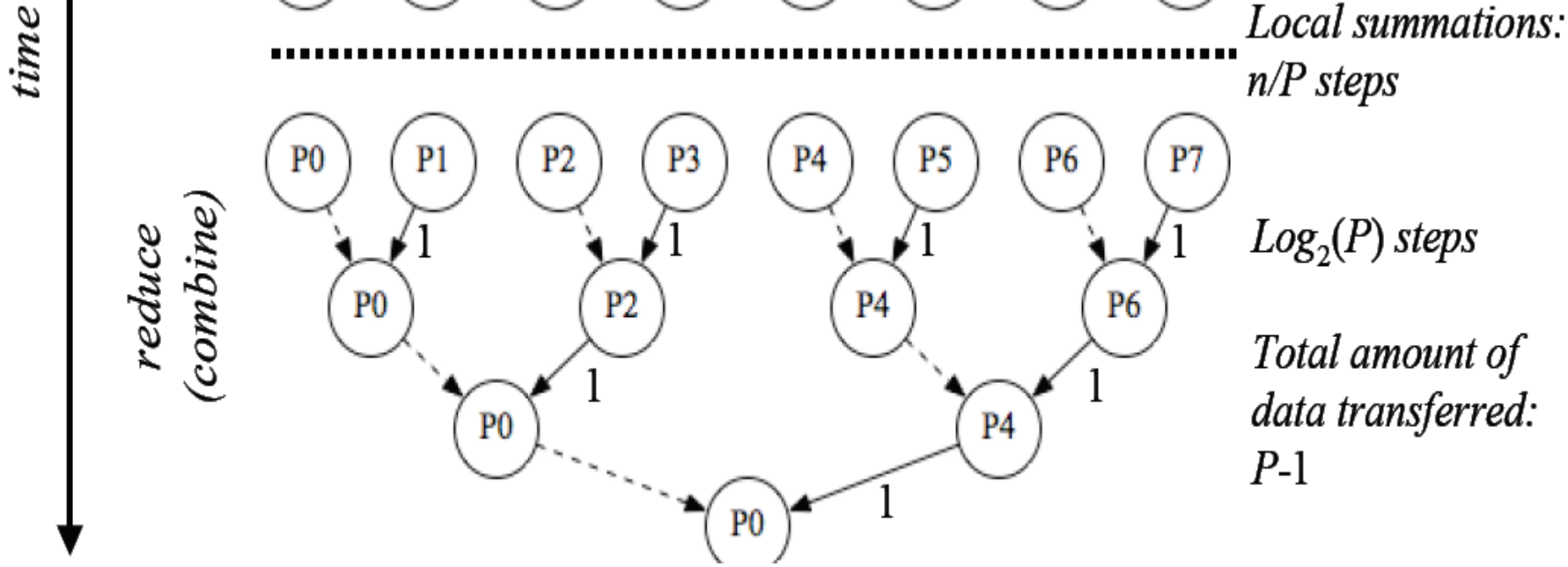
Basic steps:

1. **Break up a given problem into *P* independent subproblems**
2. **Solve the *P* subproblems concurrently**
3. **Collect and combine the *P* solutions**

Embarrassingly parallel computation: **a simple form of data partitioning without initial work and interaction between workers**

Διαμέριση/ Διαίρει και Βασίλευε – Παραδείγματα

- **Πρόσθεση**
- **Prefix Computations**
- **Mergesort**
- **Bucketsort**


time
reduce
(combine)
P0
P1
P2
P3
P4
P5
P6
P7
1
P0
P2
P4
P6
P0
P4
P0
Local summations:
n/P steps
Log2(P) steps
Total amount of
data transferred:
P-1

Ανάλυση Πολυπλοκότητας

- Χρόνος Επικοινωνίας: $t_{comm} = log\ P\ (t_{startup} + t_{data})$
- Χρόνος Υπολογισμού: $t_{comp} = n/P + log\ P$

Παράδειγμα 2:Υπολογισμός προθεμάτων αθροίσματος (prefix sum)

Δοθέντος του διανύσματος Α*[0..n-1]*, υπολόγισε τις *n* ποσότητες:

$$S[i] = A[0] + A[1] + \ldots + A[i], \quad i = 0, \ldots, n-1$$

Ακολουθιακός αλγόριθμος:

I. $S[0]:=A[0]$

II. υπολόγισε $S[i+1]$ **for** $i=0,\ldots,n-2$, **ως εξής:**

$S[i+1]:=S[i]+A[i+1]$

Ο αλγόριθμος απαιτεί *O(n)* βήματα.

Παράλληλος Αλγόριθμος υπολογισμού προθεμάτων αθροίσματος

Έστω ότι οι *n* αριθμοί είναι ομοιόμορφα μοιρασμένοι στους *p* επεξεργαστές, $P_0,\ldots,P_{p-1}$, δηλαδή, κάθε επεξεργαστής έχει *n/p αριθμούς*.

I. Κάθε επεξεργαστής ακολουθιακά υπολογίζει τα προθέματα των *n/p* αριθμών του.

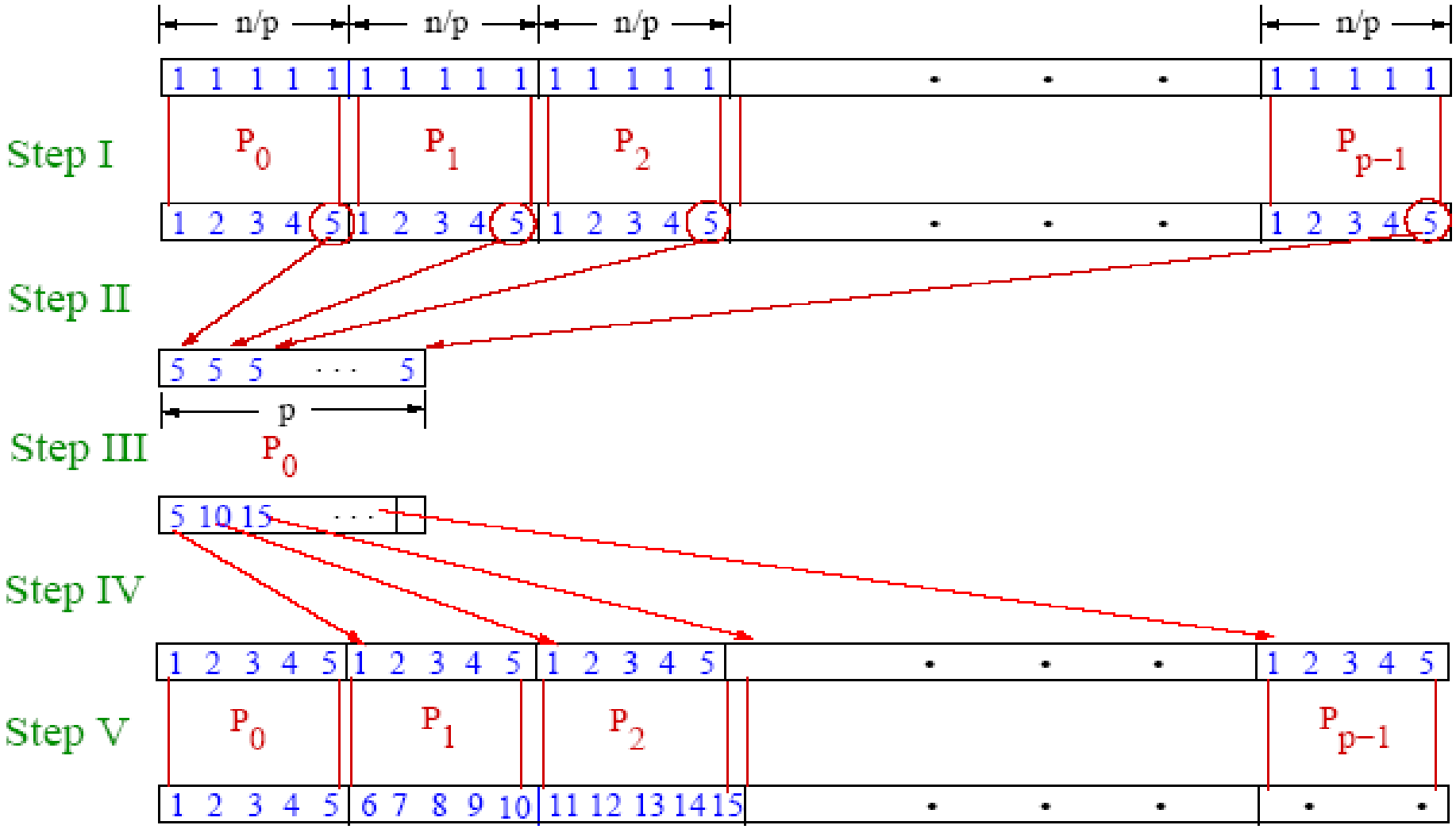
II. Κάθε επεξεργαστής *i*, $1 \leq i \leq p-1$, στέλνει το τελευταίο πρόθεμα *(S[n/p− 1])* στον επεξεργαστή *0*.

III. Ο επεξεργαστής *0* υπολογίζει τα προθέματα των *p* αριθμών που έλαβε από τους άλλους επεξεργαστές.

IV. Ο επεξεργαστής *0* στέλνει το *i* -στό πρόθεμα που υπολόγισε, στον επεξεργαστή *i*, $1 \leq i \leq p-1$.

V. Κάθε επεξεργαστής προσθέτει την τιμή που έλαβε σε κάθε ένα από τα *n/p* προθέματα που ο ίδιος έχει υπολογίσει.

Παράλληλος Αλγόριθμος υπολογισμού προθεμάτων αθροίσματος



Παράλληλος Αλγόριθμος Υπολογισμού Προθεμάτων Αθροίσματος

Ανάλυση Πολυπλοκότητας

$$
\begin{aligned}
t_{\text{comm}} &= (p-1)(t_{\text{startup}} + t_{\text{data}}) + (p-1)(t_{\text{startup}} + t_{\text{data}}) = \Theta(p) \\
t_{\text{comp}} &= \frac{n}{p} + (p-1) + \frac{n}{p} = \Theta(\frac{n}{p} + p) \\
T_p(n) &= t_{\text{comm}} + t_{\text{comp}} = \Theta(\frac{n}{p} + p) \\
T^*(n) &= \Theta(n) \\
\text{Speedup } S_p(n) &= \Theta\left(\frac{n}{\frac{n}{p} + p}\right) = \Theta\left(\frac{p}{\frac{p^2}{n} + 1}\right)
\end{aligned}
$$

Μέγιστη επιτάχυνση όταν p = √n

Παράδειγμα 3: Mergesort

Mergesort: Ακολουθιακός αλγόριθμος

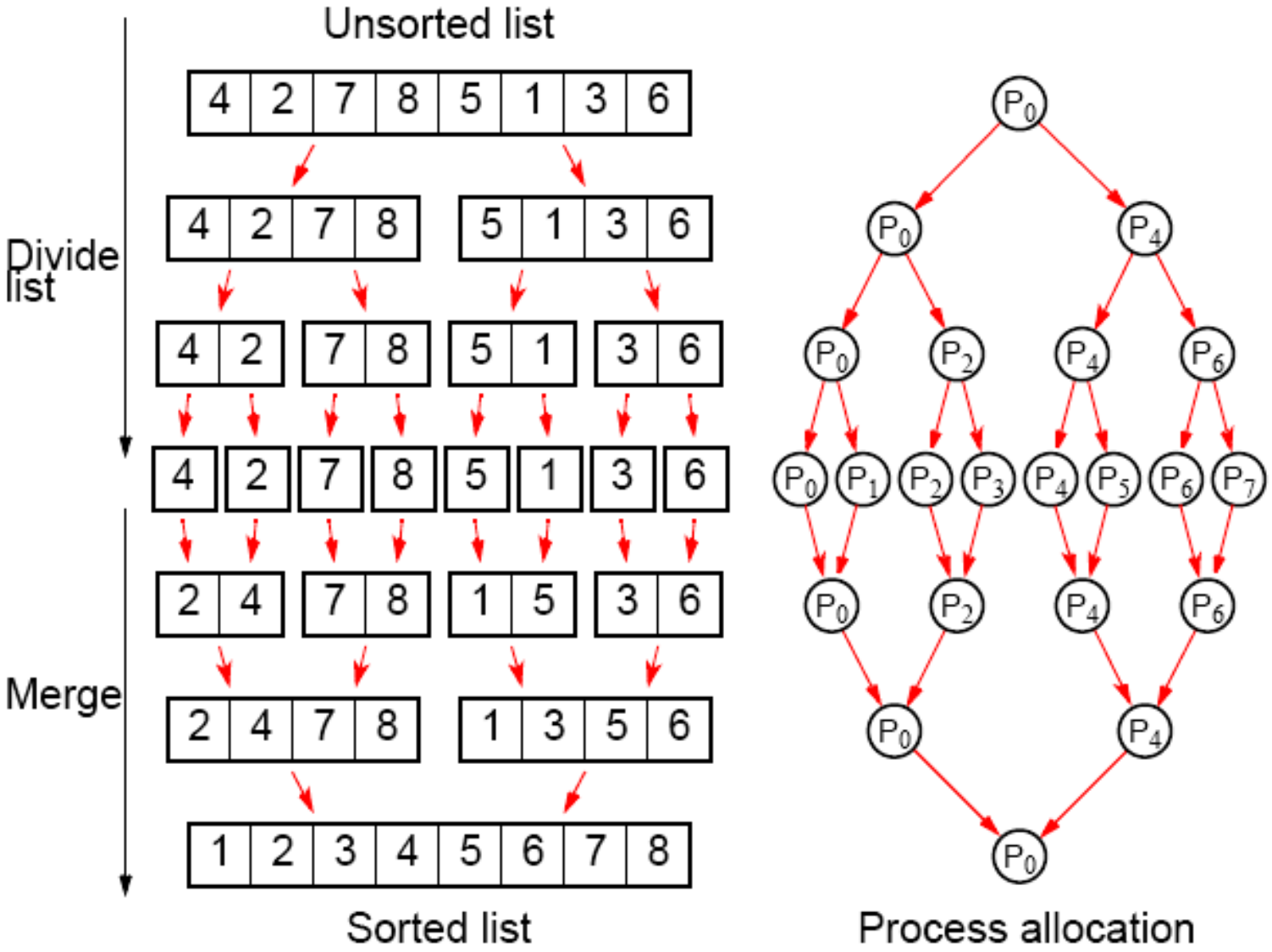
Sequential Mergesort Algorithm for sorting the elements of an array *A*, |*A*| = *n*

```
MERGESORT(A)
begin
    if ( |A|=1) then return A
                else
                    A1 := MERGESORT(S[0 .. ⌊|A|/2⌋])
                    A2 := MERGESORT(S( ⌊|A|/2⌋..|A| ])
                    return MERGE(A1,A2)
end
```

Sequential Complexity: $O(n \log n)$

Αρκεί να παρατηρήσουμε ότι η λύση της αναδρομικής σχέσης
$T(n) = 2T(n/2) + O(n)$, **όταν** $n>1$ **και**
$T(n) = O(1)$, **όταν** $n=1$ **δίνει** $T(n) = O(n \log n)$

Mergesort – Παράλληλος Αλγόριθμος *(n=p)*



Ανάλυση Πολυπλοκότητας (*n*=*p*):

# of processors = *n* then *2 log n* phases are required

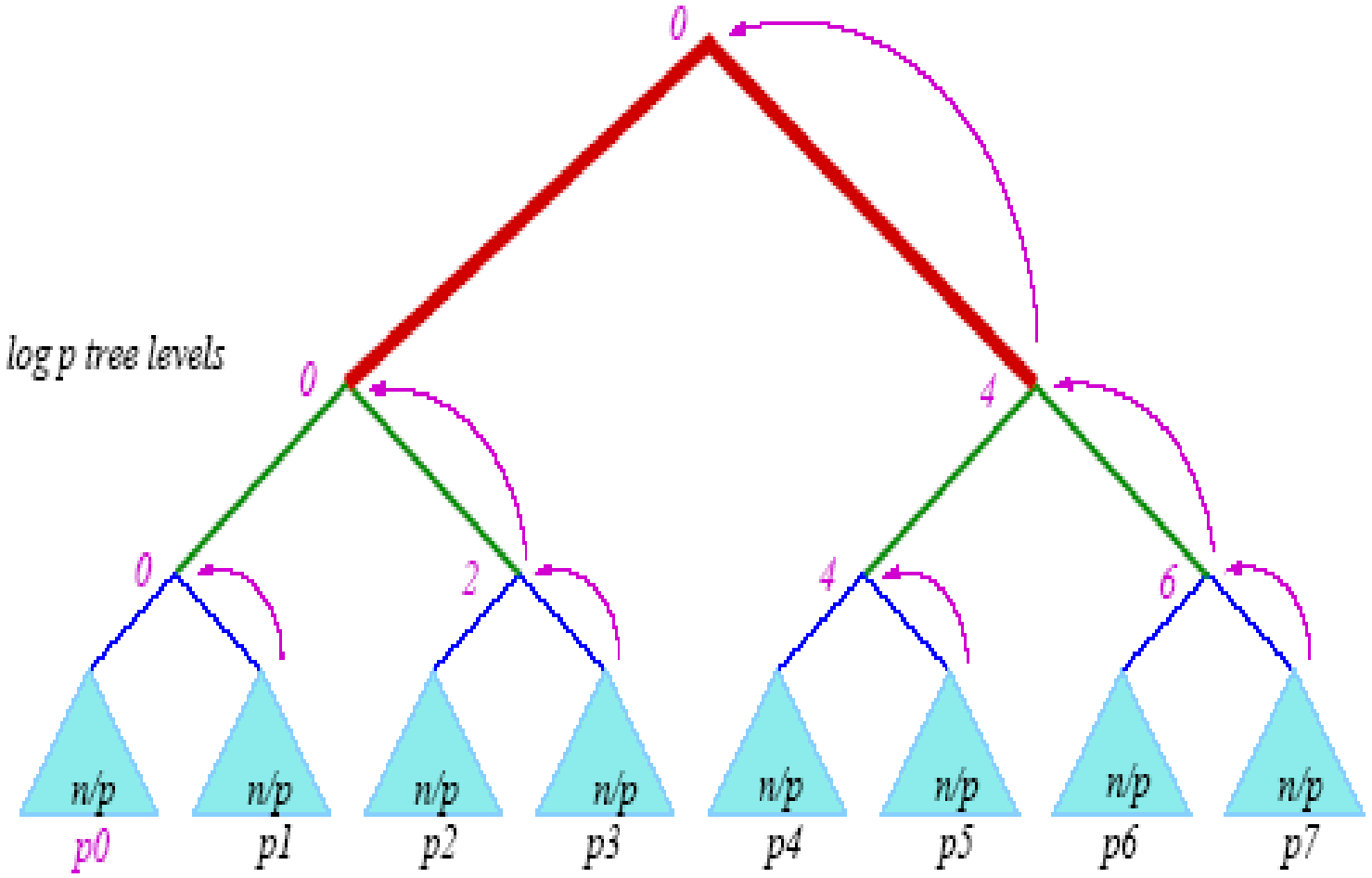
$T_{comm}=2(t_{startup}+n/2\ t_{data}+\ t_{startup}+ n/4\ t_{data} + \ldots +t_{startup}+n/2^{\log n}\ t_{data})$

$T_{comm}= 2\log n\ t_{startup} + 2n\ t_{data} = O(n)$

$T_{comp} = \Sigma_{i=1}^{\log n} (2^i -1) = O(n)$

Thus, the parallel time complexity $T(n) = O(n)$

Mergesort – Παράλληλος Αλγόριθμος *(n >> p)*



Mergesort – Παράλληλος Αλγόριθμος *(n >> p)*

Παράλληλος *bottom-up* αλγόριθμος:

1. **Κάθε επεξεργαστής ξεκινάει με περίπου *n/p* στοιχεία, όπου *n* ο αριθμός των στοιχείων που πρέπει να ταξινομηθούν και *p* ο αριθμός των επεξεργαστών.**

1. **Κάθε επεξεργαστής ανεξάρτητα ταξινομεί τα στοιχεία που του αντιστοιχούν.**

1. **Κάθε επεξεργαστής ανεξάρτητα συγχωνεύει τις 2 ταξινομημένες λίστες που του αντιστοιχούν σε μία ταξινομημένη λίστα.**

Mergesort – Παράλληλος Αλγόριθμος (*n* >> *p*)

parallel_mergesort(A, low, high, pid)
//Processes numbered, $0 \leq pid \leq p-1$
//Assume that $p = 2^k$
//Each process has n/p numbers at the start.

1. sort n/p elements locally
2. **for** (h=1; h ≤ lg p; h++)
3. **if** (pid mod $2^h == 0$)
4. recv $\frac{n}{p}2^{h-1}$elements from process $pid + 2^{h-1}$
5. **else**
6. send $\frac{n}{p}2^{h-1}$ elements from process $pid - 2^{h-1}$
7. break //this process is done
8. merge $\frac{n}{p}2^{h-1}$ elements with $\frac{n}{p}2^{h-1}$ local elements

Mergesort – Ανάλυση Πολυπλοκότητας ($n >> p$)

Υπάρχουν *log p* στάδια και σε κάθε στάδιο *h* μεταβιβάζονται $2^{h-1}n/p$ στοιχεία. Επομένως, ο χρόνος επικοινωνίας (communication time) είναι

$$t_{comm} = O(n/p + 2n/p + 2^2 n/p + \ldots + 2^{\log p - 1} n/p) =$$
$$O(n/p(1 + 2^1 + 2^2 + \ldots + 2^{\log p - 1})) = O(n)$$

Ο χρόνος υπολογισμού ισούται με το χρόνο που χρειάζεται κάθε επεξεργαστής για την ταξινόμηση των n/p στοιχείων, ήτοι

$$O(n/p \log (n/p))$$

συν το χρόνο συγχώνευσης ο οποίος διπλασιάζεται σε κάθε στάδιο:

$$O(n/p + 2n/p + 2^2 n/p + \ldots + 2^{\log p - 1} n/p) = O(n)$$

Επομένως, ο χρόνος υπολογισμού (computation time) είναι

$$t_{comp} = O(n + n/p \log (n/p))$$

Άρα, ο συνολικός παράλληλος χρόνος είναι $t_p = O(n + n/p \log (n/p))$

ενώ ο ακολουθιακός χρόνος είναι $t_p = O(n \lg n)$

- ✓ **Poor speedup because of large communication and a large number of idle processes during the merging phase!**

Παράδειγμα 4: Bucketsort

We have *n* numbers distributed uniformly in the range *[0.. α− 1]*.

The algorithm uses *m* buckets, with the i^{th} bucket representing key values in the range

$$[i\alpha/m \,..\, (i+1)\alpha/m-1],\ for\ 0 \le i \le m-1$$

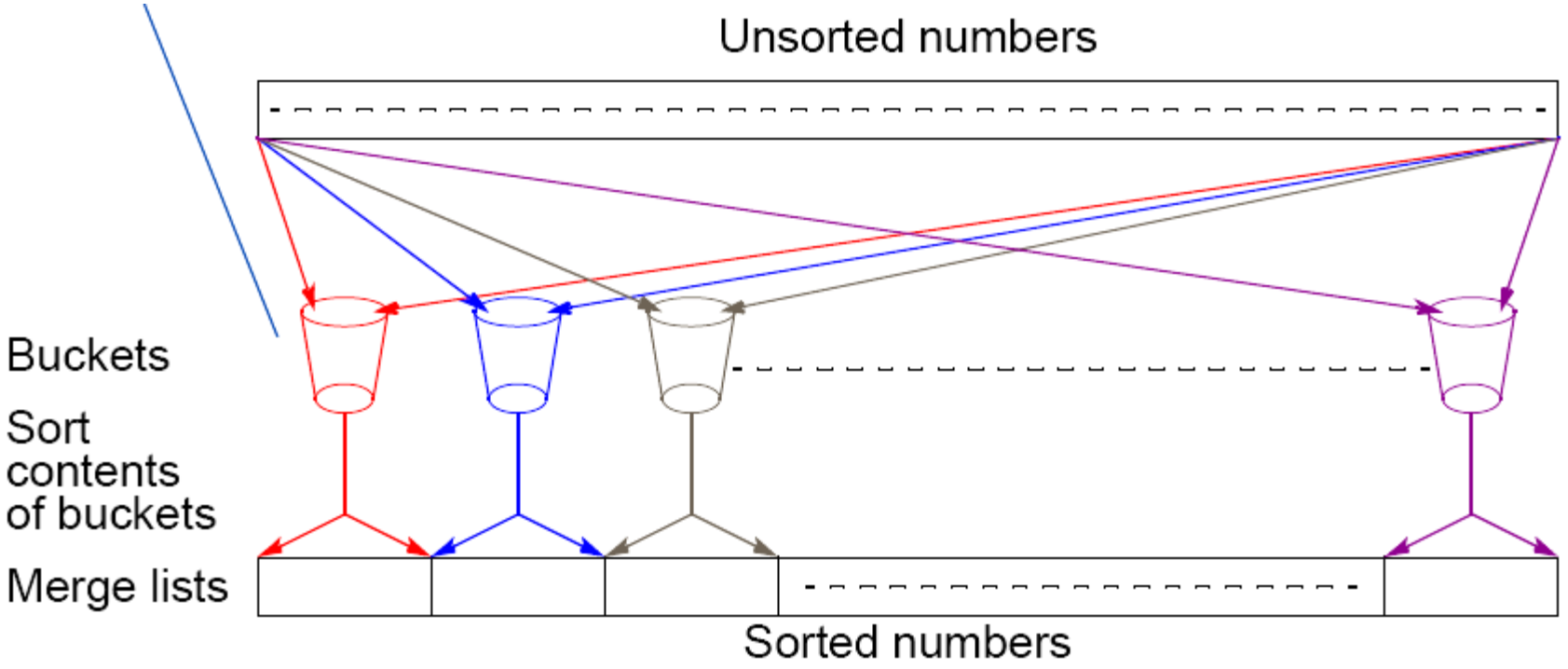
Sequential Bucketsort Algorithm

I. Place each number *x* in the bucket $\lfloor x/(\alpha/m) \rfloor$.

II. Sort the numbers in each bucket using an optimal sorting algorithm.

III. Concatenate the results together into one array.

Bucketsort


Unsorted numbers
Buckets
Sort
contents
of buckets
Merge lists
Sorted numbers

Ακολουθιακός Bucketsort – Ανάλυση Πολυπλοκότητας

Step I requires $O(n)$ time to go through the list and place each number into the appropriate bucket. If the original numbers are uniformly distributed across the interval $[0.. \alpha - 1]$, at the end of Step I, **each bucket** obtains $O(n/m)$ **numbers**.

Step II requires $O(n/m \log (n/m))$ time per bucket and therefore for m buckets requires $O(n \log (n/m))$ time

Step III requires $O(n)$ time.

Thus the time for sequential bucketsort is:

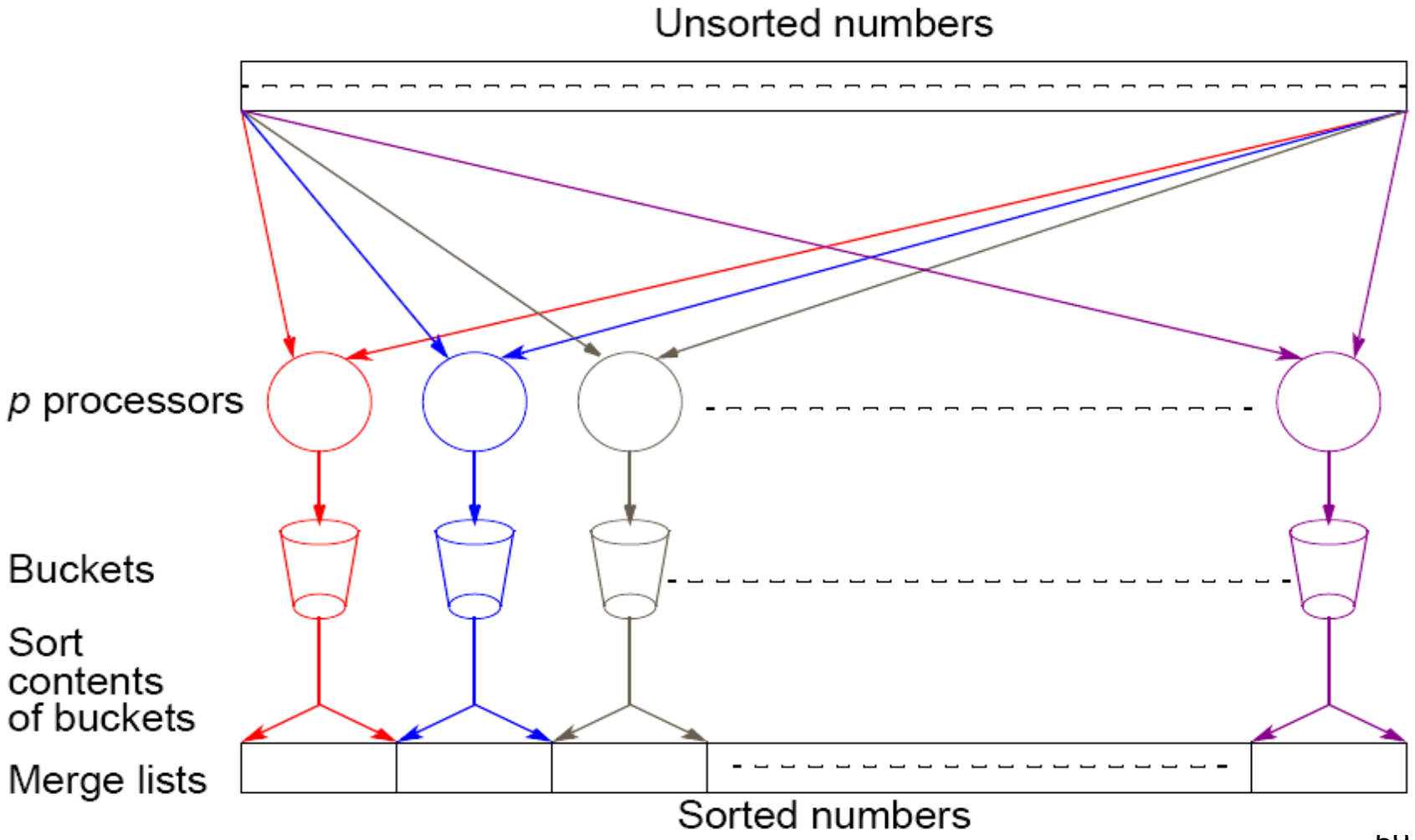
$$t_s = O(n \log (n/m))$$

If the number of buckets is $m = kn$, where k is a constant between 0 and 1, then the sequential run time for bucketsort is

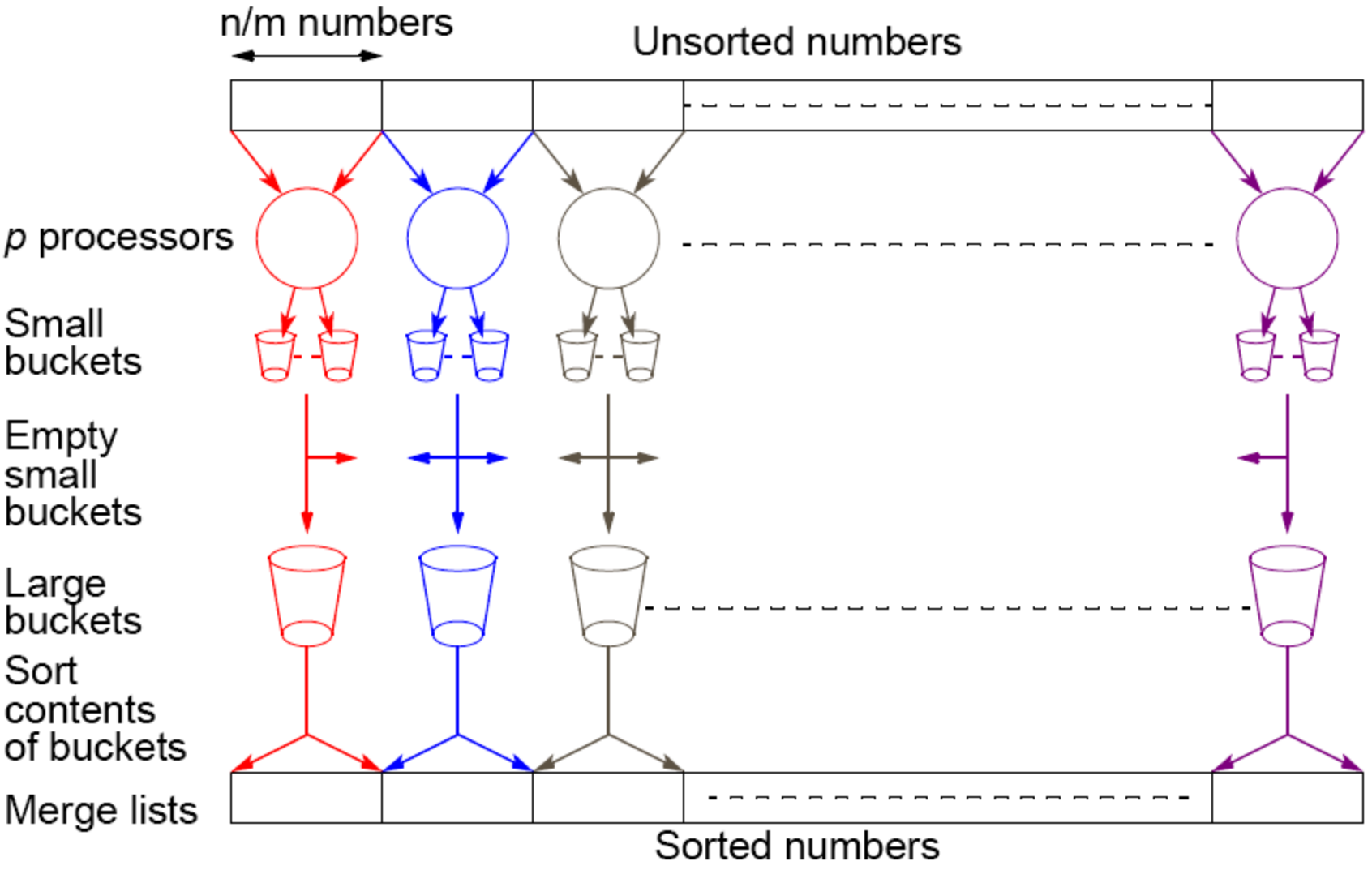
$$t_s = O(n)$$

Παράλληλος Αλγόριθμος Bucketsort

Assign one processor for each bucket.



Παράλληλος Αλγόριθμος Bucketsort



Introduces new message-passing operation - all-to-all broadcast.

Παράλληλος Αλγόριθμος Bucketsort

Input: ***n*** **numbers are distributed among the** ***p*** **processes such that each process initially holds** ***n/p*** **numbers.** ***m=p.***

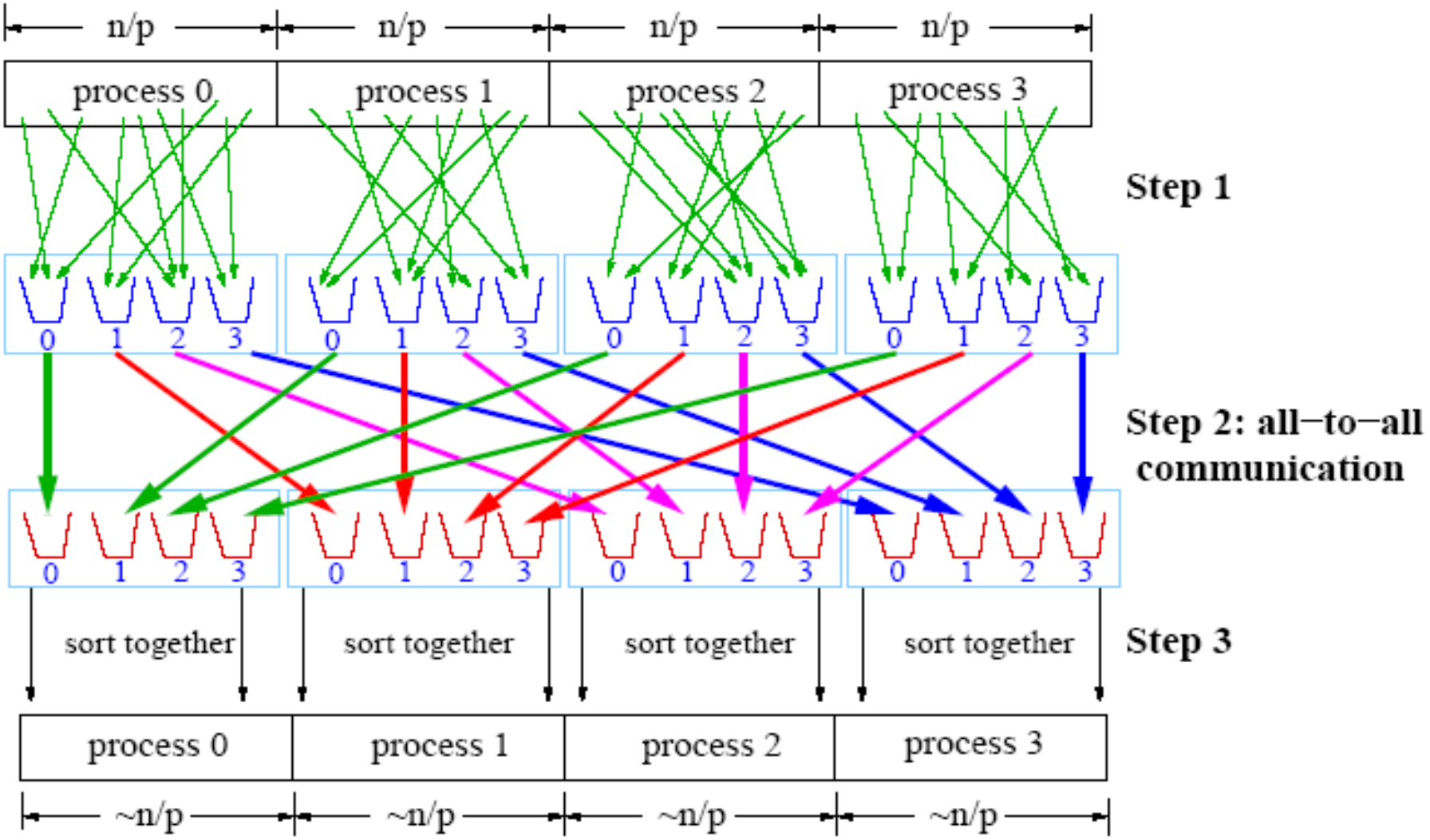
Output: **For each** ***i*****, process** ***i*** **has sorted numbers. Also, any number on process** ***j*** **is greater than or equal to any number on process** ***i*****, where** $i < j$**, while any number on process** ***i*** **is less than or equal to any number on process** ***k*****, where** $k > i$**.**

I. Each process places its ***n/p*** **numbers into** ***p*** **"small" buckets. Number** ***x*** **is placed to the "small" bucket number** $\lfloor x/(\alpha/p) \rfloor$**.**

II. For each process ***i*****,** $0 \leq i \leq p-1$**, send the** j^{th} **"small" bucket to process** j**,** $0 \leq j \leq p-1$**,** $j \neq i$ ***(all-to-all communication).***

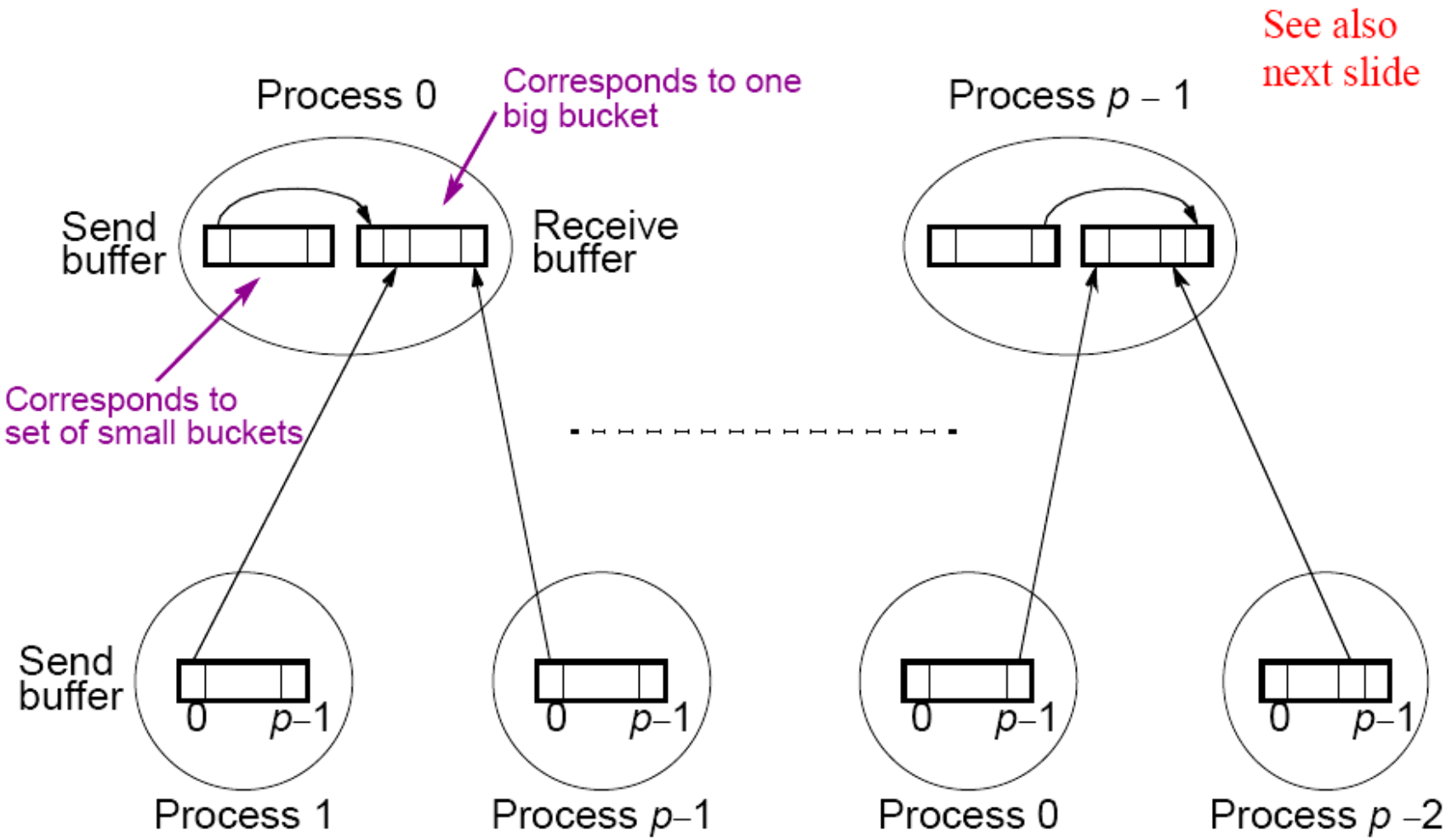
III. Each process receives $p-1$ **small buckets. It combines them with its own "small" buckets and then sorts all of them together. to make one list of approximately** ***n/p*** **numbers.**

Παράλληλος Αλγόριθμος Bucketsort



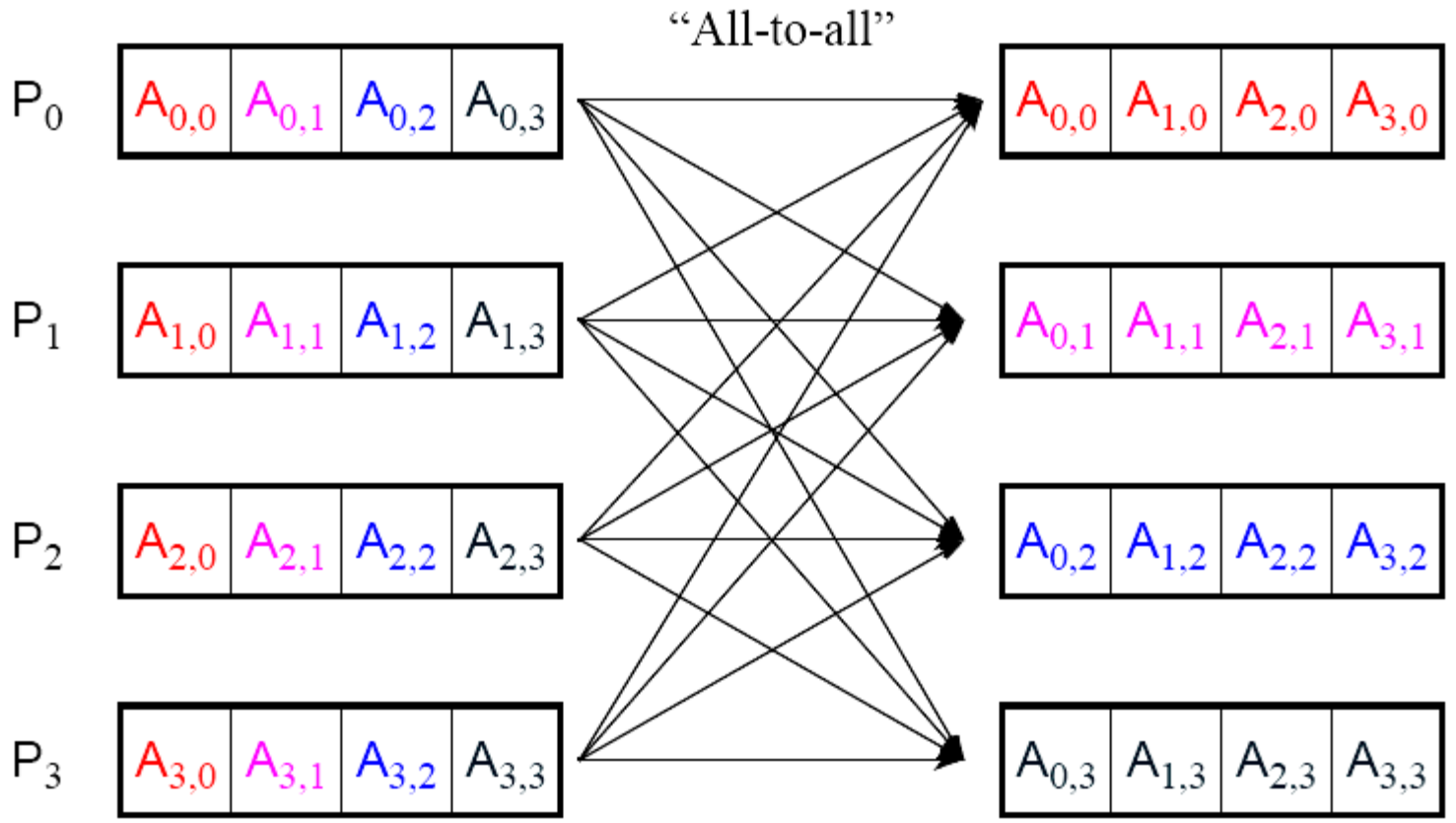
"all-to-all" communication

Sends data from each process to every other process



“all-to-all” communication

Transfers rows of an array to columns: transposes a matrix.



Παράλληλος Αλγόριθμος Bucketsort: Ανάλυση Πολυπλοκότητας

Step I. $t_{comp} = O(n/p)$ $t_{comm} = O(1)$

Assuming uniform distribution, at the end of Step I each small bucket will have about n/p^2 numbers.

Step II. **Each process sends $p - 1$ "small" buckets to other processes. If these communications cannot be overlapped and individual sends are used then**

$$t_{comp} = O(1)$$

$$t_{comm} = O(p(p-1)(t_{startup} + n/p^2\, t_{data}))$$

Step III.

$$t_{comp} = O(n/p \log (n/p))$$

$$t_{comm} = O(1)$$

Therefore, the overall runtime for parallel bucketsort is:

$$t_p = O(n/p \log (n/p) + p^2 t_{startup} + n\, t_{data})))$$

Παράλληλος Αλγόριθμος Bucketsort: Ανάλυση Πολυπλοκότητας

We can break the all-to-all communication into p rounds. In each round each process is sending out one message and receiving one message as follows:

round 0	$P_0 \rightarrow P_0$	$P_1 \rightarrow P_1$	$P_2 \rightarrow P_2$	$P_3 \rightarrow P_3$
round 1	$P_0 \rightarrow P_1$	$P_1 \rightarrow P_2$	$P_2 \rightarrow P_3$	$P_3 \rightarrow P_0$
round 2	$P_0 \rightarrow P_2$	$P_1 \rightarrow P_3$	$P_2 \rightarrow P_0$	$P_3 \rightarrow P_1$
round 3	$P_0 \rightarrow P_3$	$P_1 \rightarrow P_0$	$P_2 \rightarrow P_1$	$P_3 \rightarrow P_2$

If the processes are on nodes connected via a completely connected network and the network interface is capable of sending/receiving at the same time then each round can be done simultaneously. Then, Step II needs communication time

$$t_{comm} = O(p(t_{startup} + n/p^2\, t_{data}))$$

And the overall runtime for parallel bucketsort becomes:

$$t_p = O(n/p \log (n/p) + pt_{startup} + n/p\, t_{data})))$$