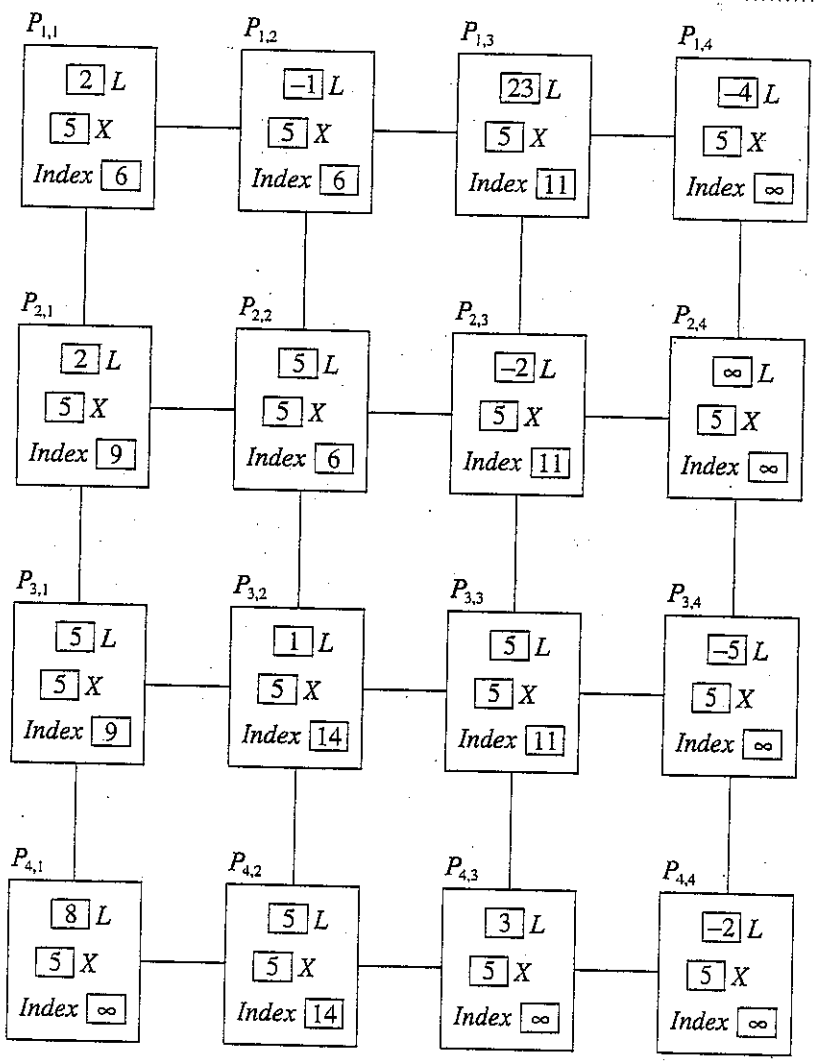
Figure 5.15 State of the distributed variables *L*, *X*, and *Index* upon completion of the searching algorithm



for $1 \le i \le n$ **do in parallel**
  $C[i] := A[i]*B[i]$
end in parallel

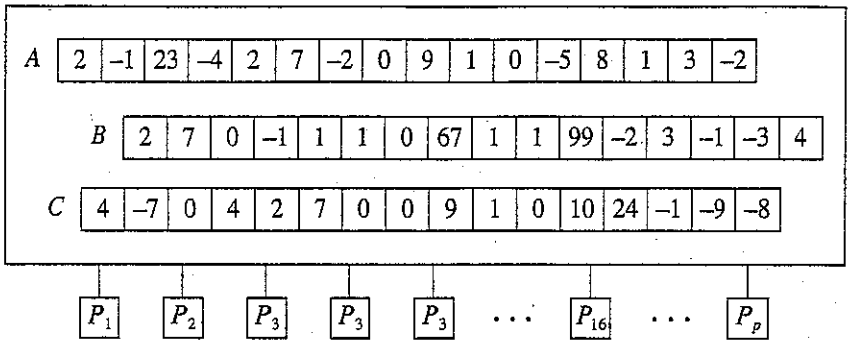


Figure 5.18 Component multiplication on a PRAM

Figure 5.19 Component multiplication on $M_{q,q}$, $q = 4$, $p(n) = n = 16$

for $P_{i,j}$, $1 \le i,j \le q$ **do in parallel**
  $C := A*B$
end in parallel

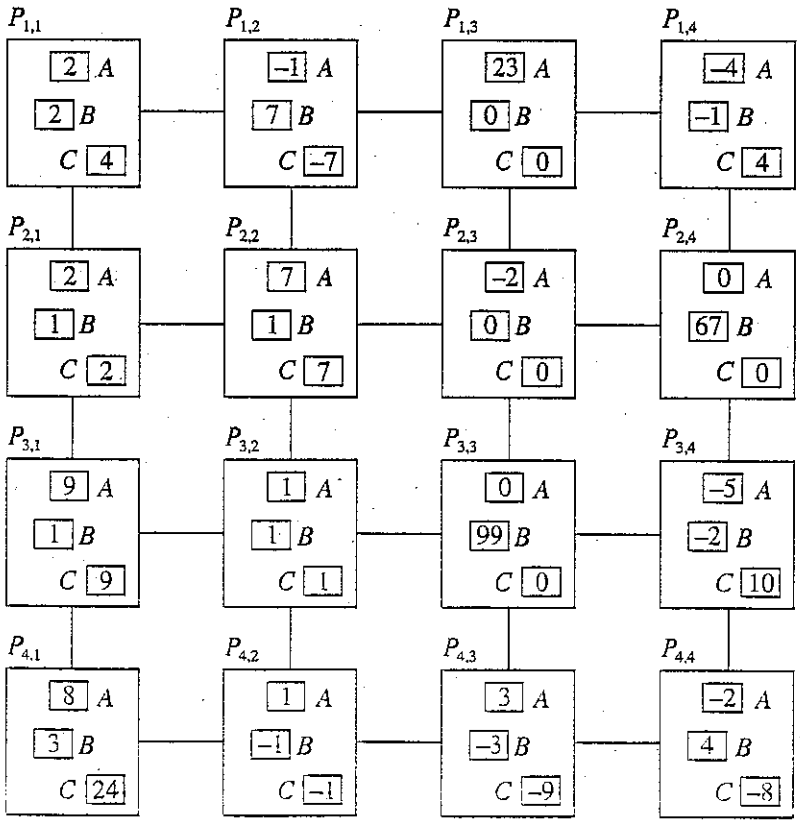
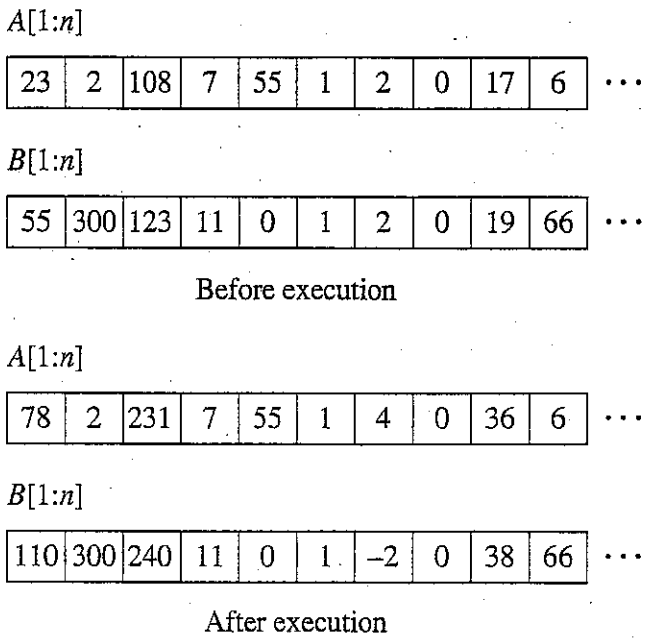


Figure 5.20 Action of **in parallel** statement for sample arrays

```
for 1 ≤ i ≤ 2k – 1 .and. odd(i) do in parallel
   A[i] := A[i] + B[i]
   B[i] := 2*B[i]
   if i mod 4 = 3 then
      B[i] := B[i] – 6
   endif
end in parallel
```




```
for 1 ≤ i ≤ n do in parallel
   if i ≤ 10 then
      A[i] := 2*A[i]
   else
      A[i] := A[i] + 1
   endif
end in parallel
```

MIMD PRAM

```
for 1 ≤ i ≤ 10 do in parallel
   A[i] := 2*A[i]
end in parallel
for 11 ≤ i ≤ n do in parallel
   A[i] := A[i] + 1
end in parallel
```

SIMD PRAM

for P_i, $1 \le i \le 2k-1$ **.and. odd**(i) **do in parallel**
  $A := A + B$
  $B := 2*B$
  if i **mod** $4 = 3$ **then**
    $B := B - 6$
  endif
end in parallel

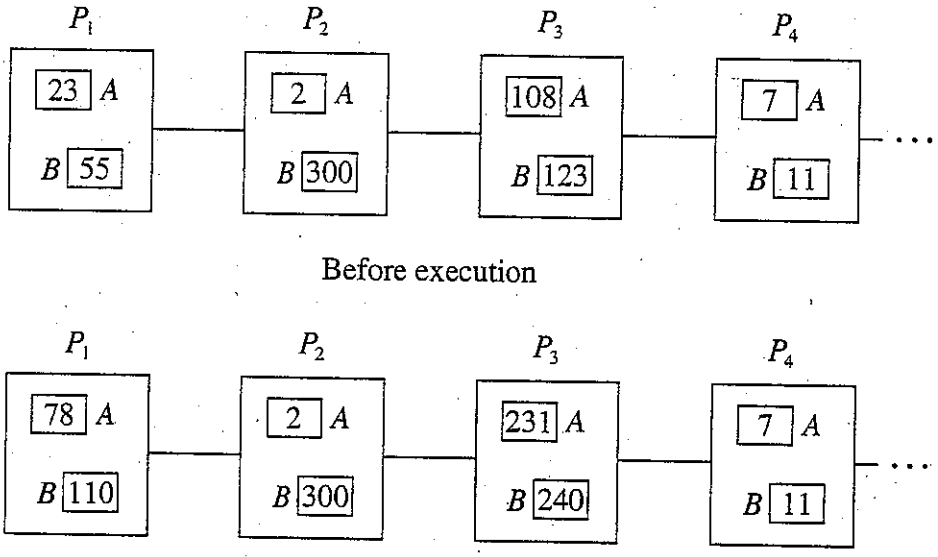
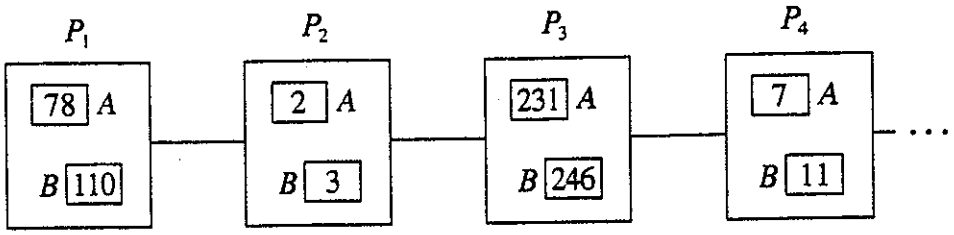


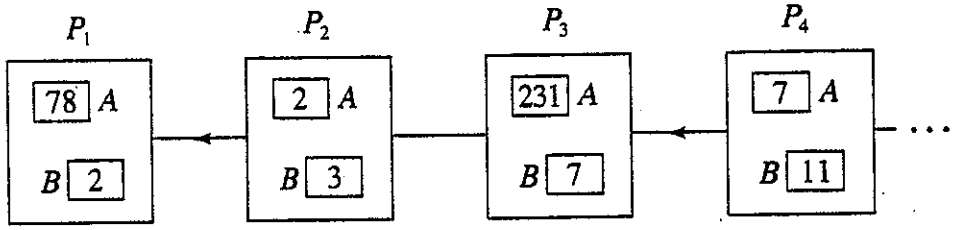
Figure 5.21 Action of **in parallel** statement on the one-dimensional mesh M_p

Interprocessor Communication Statement

for P_i, $2 \le i \le 2k$ **.and. even**(i) **do in parallel**
  $\mathbf{P}_{i-1}$:$B \Leftarrow \mathbf{P}_i$:$A$ {communicate left from A to B}
end in parallel



Before execution

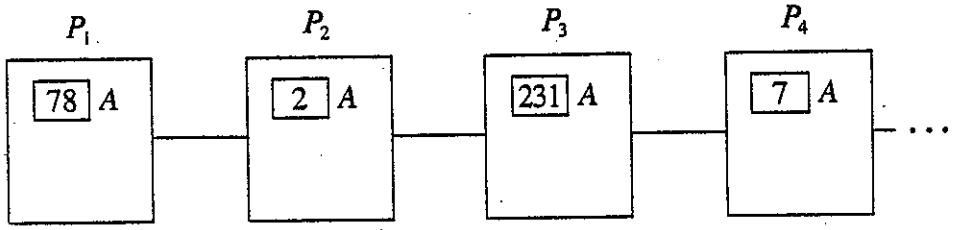


After execution

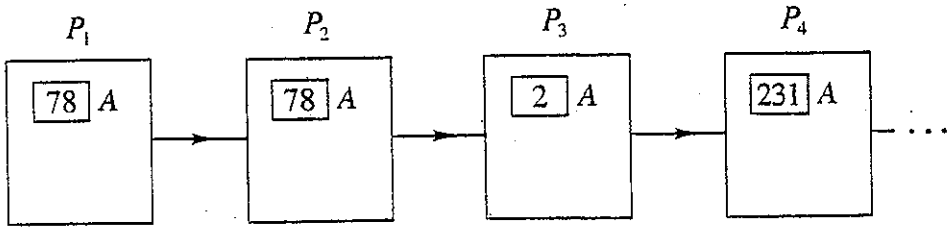
Figure 5.23 Action of parallel communication step

Figure 5.24 Action of parallel communication with processors being both source and target

for P_i, $1 \le i \le n-1$ **do in parallel**
  $\mathbf{P}_{i+1}$:$A \Leftarrow \mathbf{P}_i$:$A$ {propagate A to the right} // SAR //
end in parallel



Before execution

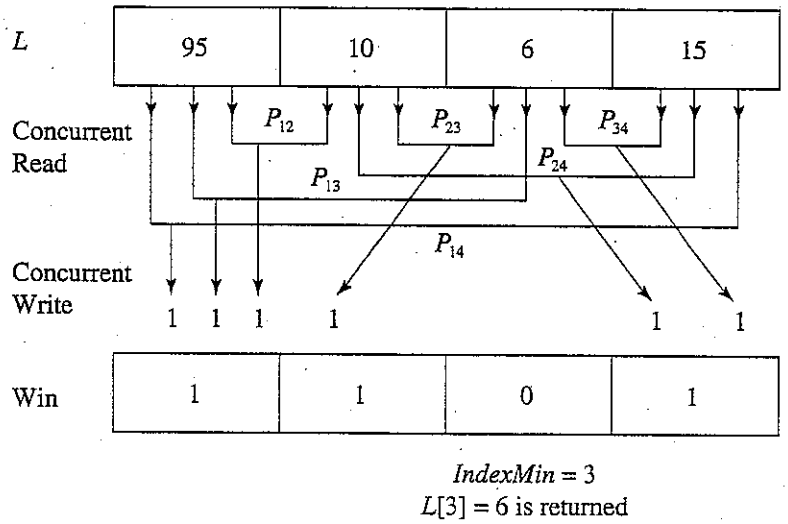


After execution

```
function MinCRCW(L[1:n])
Model: CRCW PRAM with p = (n^2 − n)/2 processors
Input: L[1:n] (a list of size n)
Output: the minimum value of a list element in L
  for 1 ≤ i ≤ n do in parallel
    Win[i] := 0
  end in parallel
  for 1 ≤ i, j ≤ n .and. i < j do in parallel
    {P_{i,j} reads and compares L[i] and L[j]}
    if L[i] > L[j] then
      Win[i] := 1    {processors P_{i,j} concurrently write 1 to Win[i]}
    else
      Win[j] := 1    {processors P_{i,j} concurrently write 1 to Win[j]}
    endif
  end in parallel
  for 1 ≤ i ≤ n do in parallel
    if Win[i] = 0 then IndexMin := i endif
  end in parallel
  return(L[IndexMin])
end MinCRCW
```

The action of *MinCRCW* is illustrated for a sample list of size 4 in Figure 5.26.

Figure 5.26 Action of *MinCRCW* for the sample input list [*L*: 95, 10, 6, 15]. *MinCRCW* uses six processors: P_{12}, P_{13}, P_{14}, P_{23}, P_{24}, P_{34}.



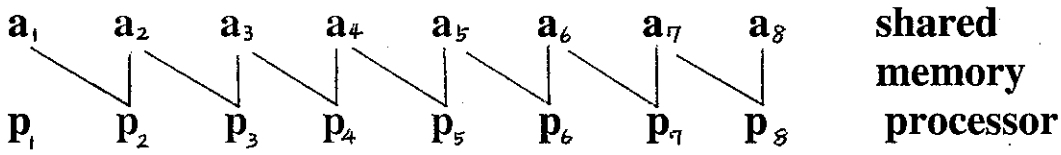
$$W(n) = 1, \qquad \text{// worst-case time complexity //}$$

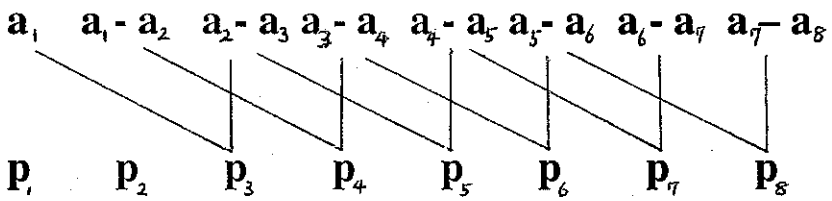
$$S(n) = n - 1. \qquad \text{// speed-up}$$

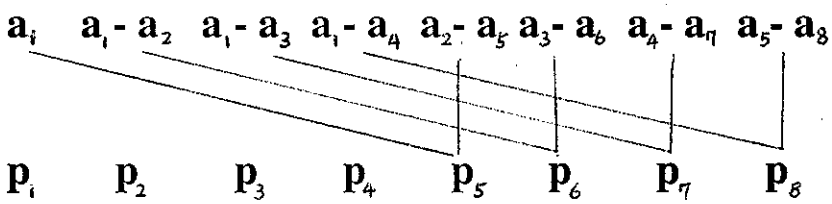
$$C(n) = \frac{n^2 - n}{2}, \quad \text{and} \quad E(n) = \frac{2}{n}, \qquad \text{// C = cost, E = efficiency //}$$

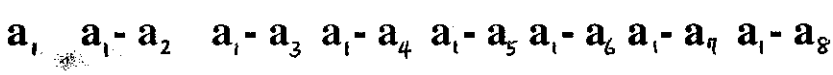
16.1 Prefix Sum (Partial Sum)

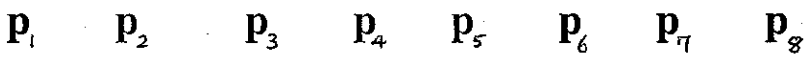
(1) • EREW SM SIMD







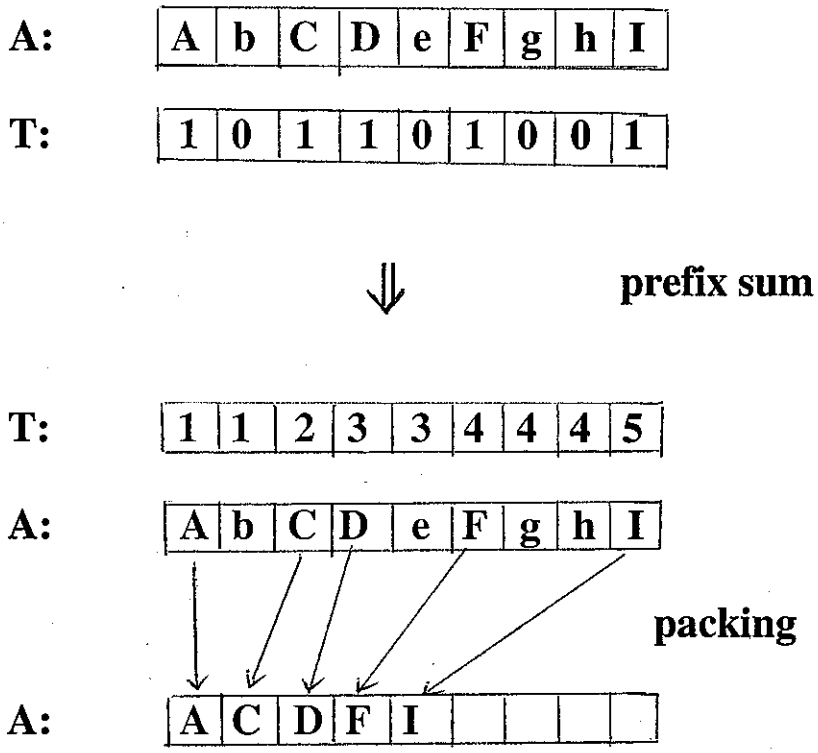




$T(n) = \lceil \log n \rceil$

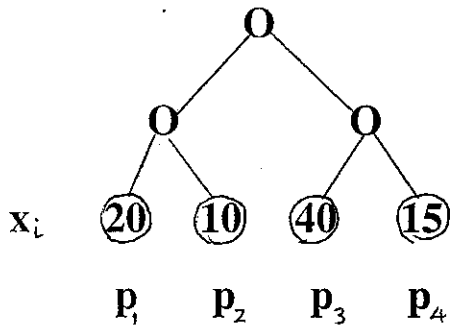
$P(n) = p\ (p/2)$

Example Packing



Book — Top down
4 — Bottom up

(2) • **Partial Sum on a Tree**



root:

if input is received from left child then send it to right child
if input is received from right child then discard it

intermediate:

if input is received from left and right children then
send sum to parent and send left input to right child
if input is received from the parent then
send it to both children

leaf:

$s_i := x_i$
send the value x_i to the parent
if input is received from parent then add it to s_i

Ex.

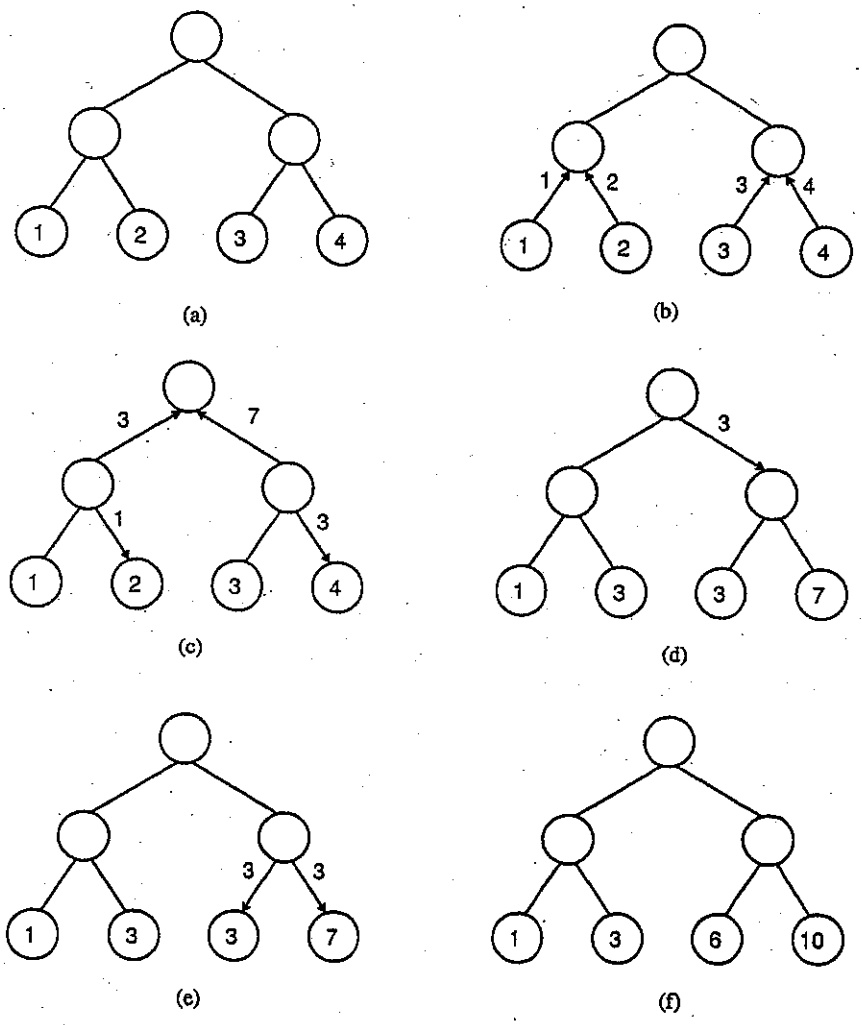


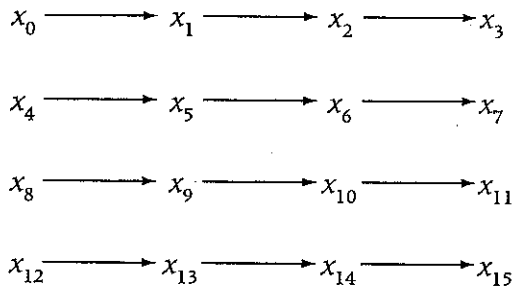
Figure Computing prefix sums on tree of processors.

$T(n) = 2 \cdot \log n$

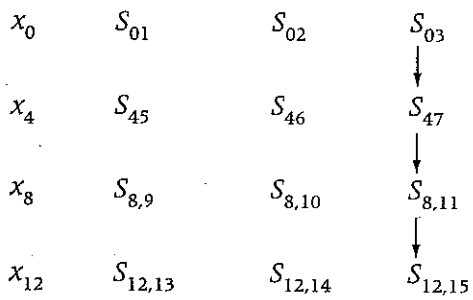
$P(n) = 2n - 1$

$C(n) = O(n \cdot \log n)$ // $W(n)$

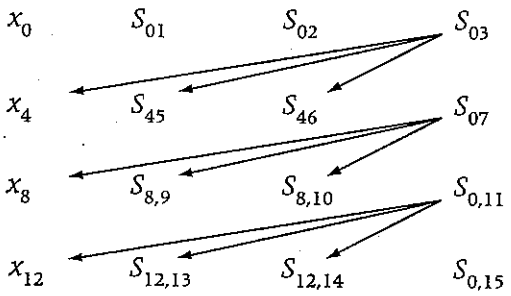
(3) • Partial Sum on a 2-D Mesh



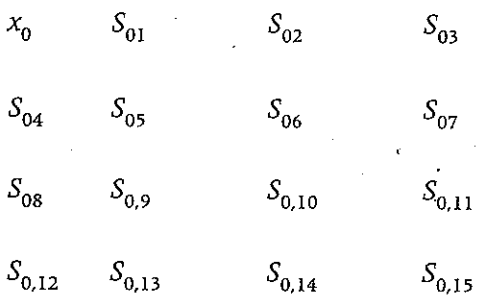
(a) Initial contents of *Prefix* and direction for performing the summing in phase 1.



(b) Contents of *Prefix after* phase 1 and direction for performing the summing in phase 2.



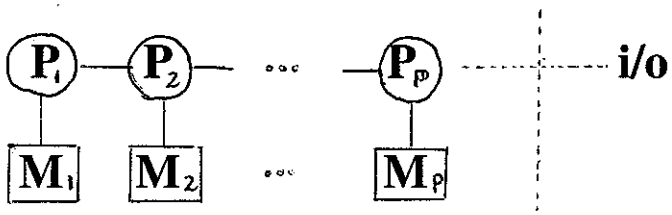
(c) Contents of *Prefix* after phase 2. Arrow from processor $P_{i-1,q-1}$ to P_{ij} indicates that P_{ij}:*Prefix* should be replaced with P_{ij}:*Prefix* $\oplus$ $P_{i-1,q-1}$ *Prefix* in phase 3. This can be accomplished in q parallel steps.



(d) Contents of *Prefix* upon completion.

Find Sum

(1). MC^1 (one-dimensional array)



- send (neighbor, message)
- receive (neighbor, message)

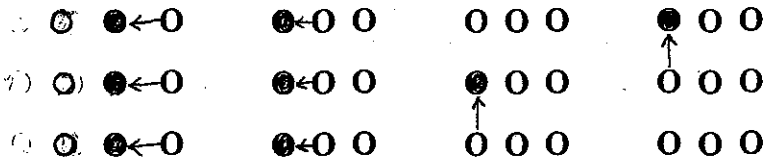
Let N = kp.
Each local memory stores k numbers.
Each process obtains its partial sum -- (k-1)
P_1 sends S_1 to P_2, P_2 adds S_1 to S_2, and sends it to P_3... -- (p-1)

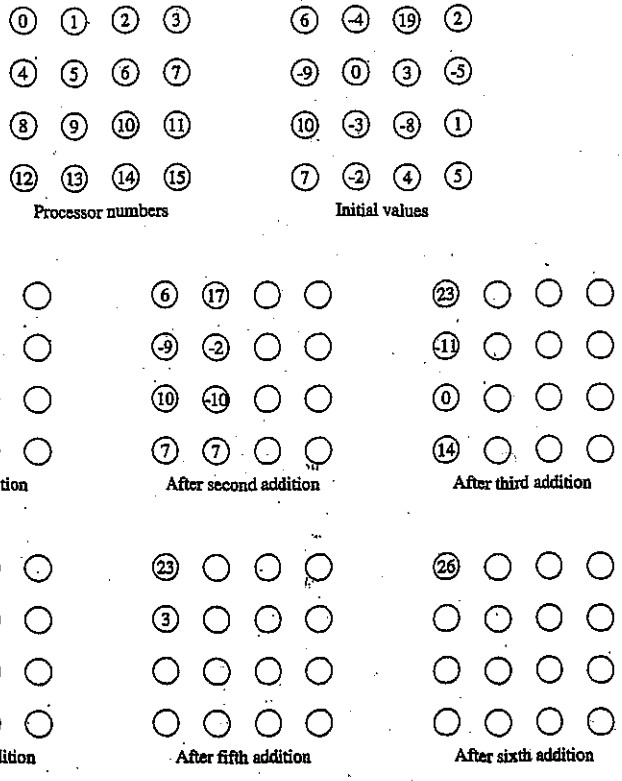
```
{each process Pi computes the sum of its local numbers A[1..k]}
      sum:= 0;
      for i:= 1 to k do
          sum:= sum + A[i];
{Process Pi send its local sum to P2}
      if INDEX = 1 then send(RIGHT,sum)
      else
{every remaining Pi wait to receive an external result from Pi-1}
      begin
          receive (LEFT, LEFTSUM)
          SUM:= SUM + LEFTSUM
          {each Pi except Pp sends SUM to Pi+1}
          if INDEX < p then send (RIGHT, SUM)
      end
```

$T(n) = k_1 N / p + k_2 (p - 1)$

(2) MC^2 (2-D Mesh)

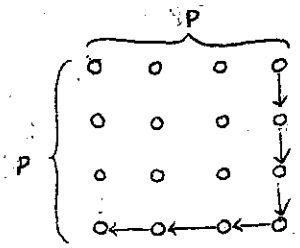
Idea.





Finding sum of 16 values on a processor array organized as a 2-D mesh.

Note. n values, p processors ($p = \sqrt{n}$)



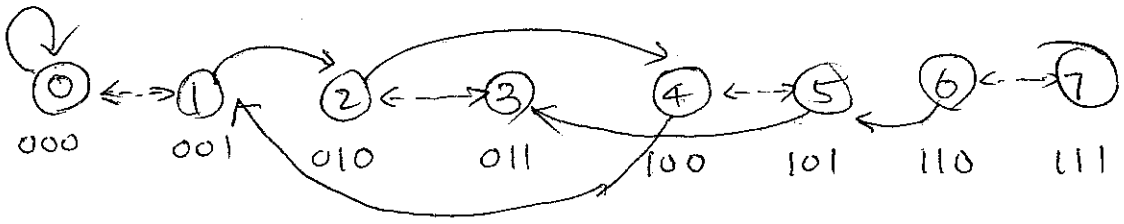
communication step: $2(p-1)$ lower-bound

$$T(n) = O(n/p + p)$$

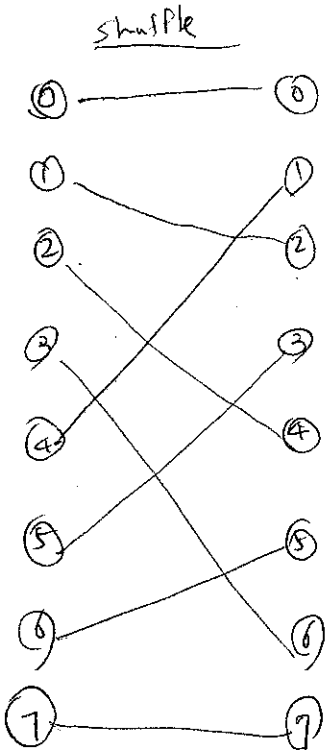
$$= O(\sqrt{n})$$

$$p(n) = \sqrt{n}$$

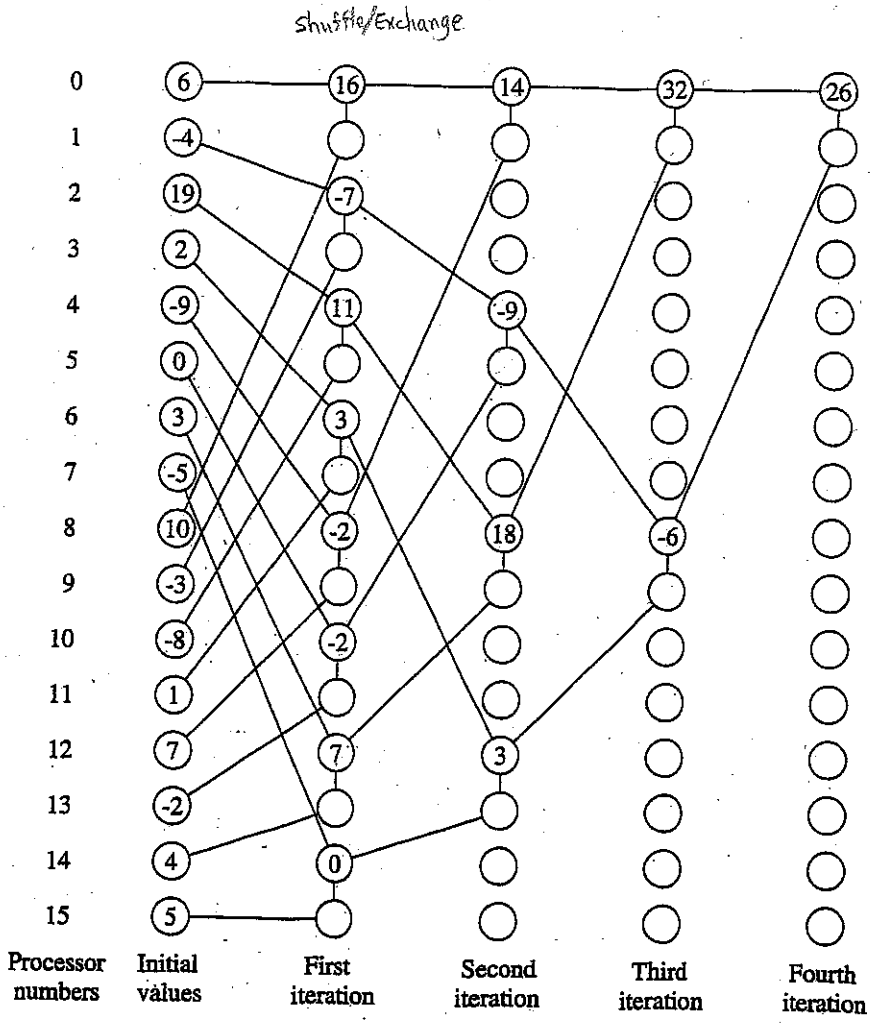
Shuffle-Exchange Network (Perfect-Shuffle)



Note ⟷ shufle
⟷ exchange



(3). Shuffle-Exchange



Finding sum of 16 values on the shuffle-exchange SIMD model.

Complexity:

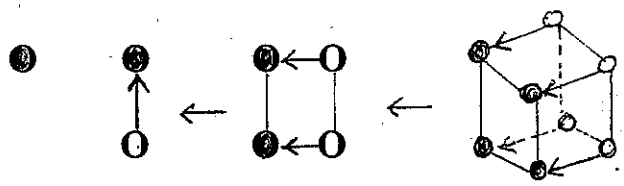
n/p for local sum

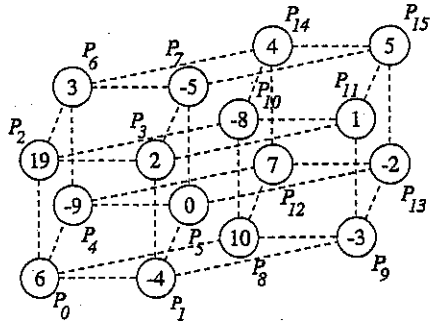
Log P iterations (each iteration = 1 shuffle + 1 exchange)

$$T(n) = O(n/p + \log P)$$

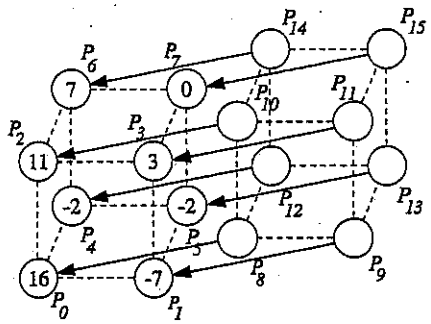
(4). Hypercube SIMD

Idea:

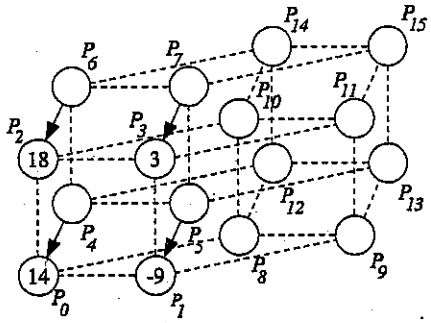




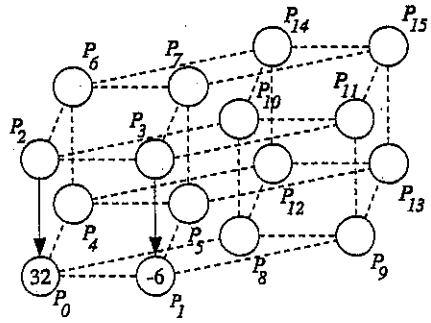
Values to be added



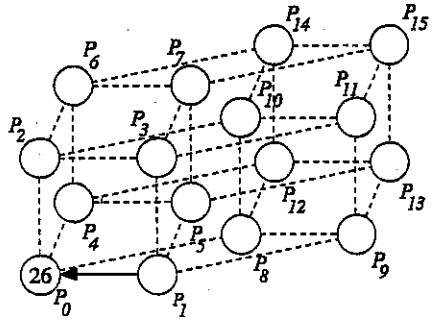
First iteration



Second iteration



Third iteration



Fourth iteration

FIGURE 6-2 Parallel summation on the hypercube SIMD model.

Complexity: n/p to find local sum
Log p steps

$$T(n) = O(n/p + \log p)$$

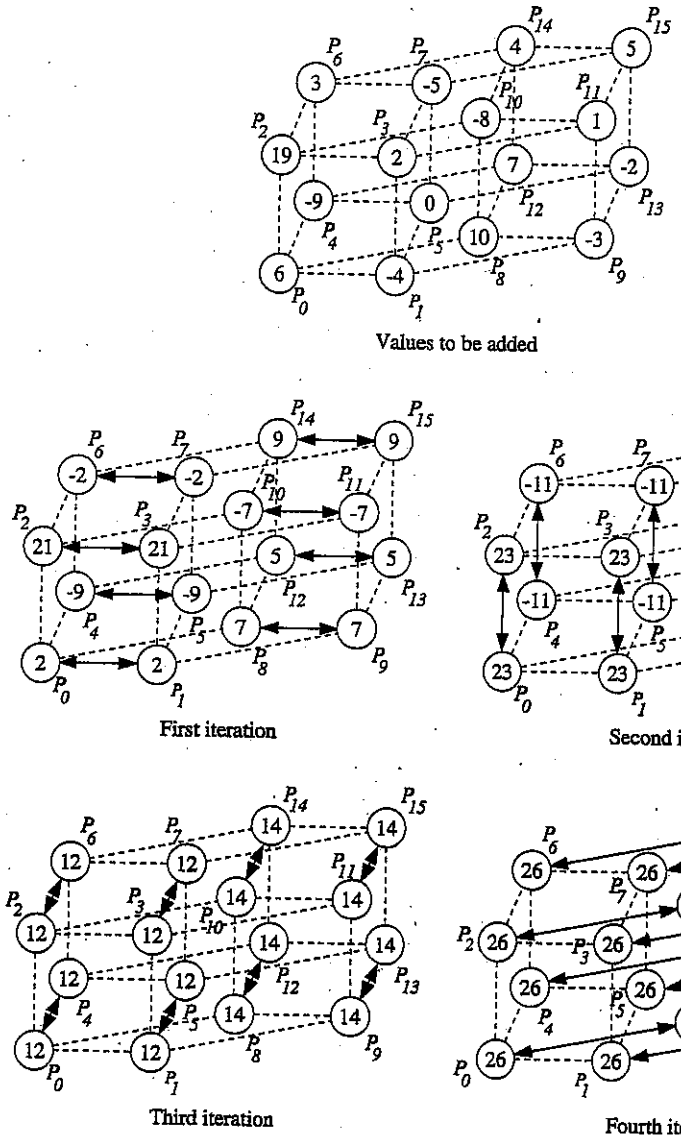
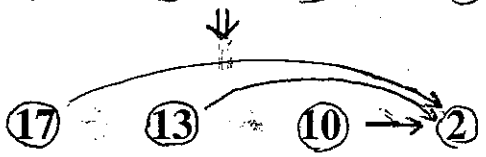


FIGURE 6-3 Another parallel summation algorithm for the hypercube SIMD model. After the processing elements have found the sum of their local values, they perform $\log p$ swap-and-accumulate steps, one for each dimension of the hypercube.

Suffix Sum

- **linked-list**

Ex. ④ → ③ → ⑧ → ②

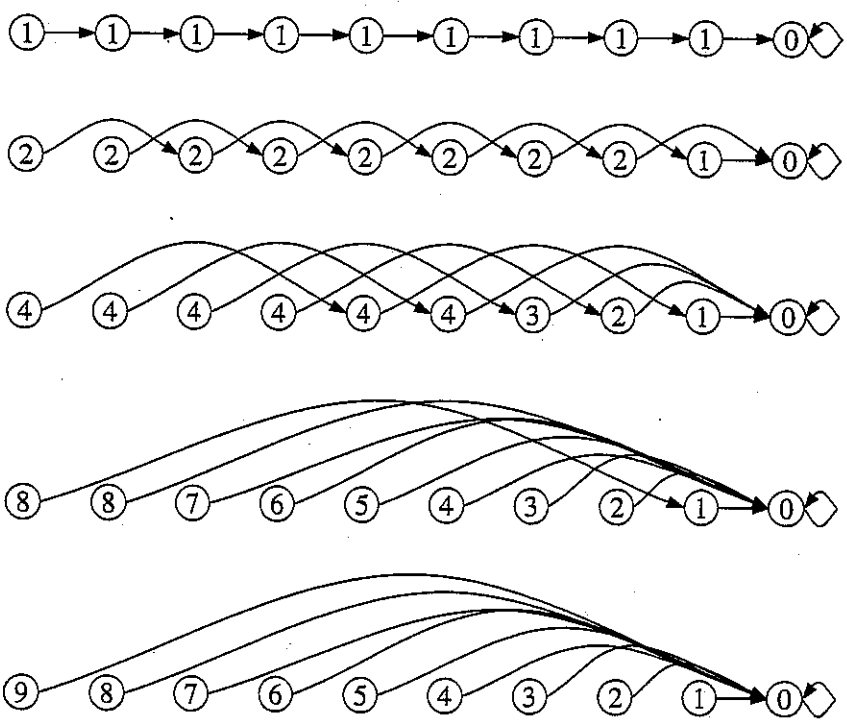


List Ranking Problem:

Find the position of each item on an n-element linked list.

Idea. The distance to the end of the list is cut in half through the instruction, next[i] ← next[next[i]].

EX.



T(n) = O(log n)
P(n) = n

Parallel Sorting

Sorting
- internal or external
- comparison or noncomparison

Note: The art of computer programming, Vol 3, Knuth

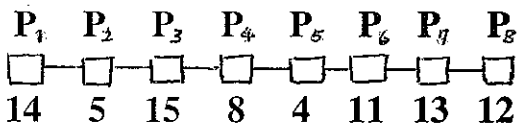
Table 14.1.1 Parallel Comparison Sort Algorithms

Method	Input size	Number of processors	Interconnection network	Time	Space
Enumeration sorts					
Preparata (1978)	n	$n \log n$		$O(\log n) + c \log n$	
Muller and Preparata (1975)	n	$O(n^2)$	Sorting network	$O(\log n)$	
Nassimi and Sahni (1979)	$n \times n$	$n \times n$	Mesh		
	n	$n^{1+1/k}$ $1 \leqslant k \leqslant \log n$	Perfect shuffle	$O(k \log n)$	
Ramakrishnan and Browne (1983)	n	n	Linear	$O(n)$	
Odd–even transposition sorts					
Baudet and Stevenson	n	$\log n$	Linear	$O(\log n)$ †	
(1978)	n	$(\log n/\log \log n)^2$	Mesh	$O((\log n/\log \log n)^2)$	
	n	$2^{\sqrt{\log n}}$	Perfect shuffle	$2^{\sqrt{\log n}}$	
Thompson and Kung	n	n	Linear	$O(n)$	
(1977)	$n \times n$	$n \times n$	Mesh	$O(n)$	
Bucket sorts					
Hirschberg (1978)	n numbers in the range of $\{0, \ldots, m-1\}$	n		$O(n)$	$O(mn)$
Orenstein *et al.* (1983)	n	$O(\log n)$	Linear	$O(n \log n)$	$O(n \log n)$
Mergesorts					
Batcher (1968)	n	$O(n \log^2 n)$	Sorting network	$O(\log^2 n)$	
Valiant (1975)	Two sorted files of size n and m	$\sqrt{nm}$		$2 \log \log n + O(i)$	
Orenstein *et al.* (1983)	n	$O(\log n)$	Linear	$O(n \log \log n)$	
Todd (1978)	n	$O(\log n)$		$O(n)$	$O(n)$

ODD-EVEN TRANSPOSITION SORT

- linear array (1-D mesh) (SIMD MC)

(1) p = n



odd-even exchange	14—5 15—8 4—11 13—12
even-odd exchange	5—14 15—4 8—11 13—12
odd-even exchange	5 14—4 15—8 11—12 13
even-odd exchange	5—4 14—8 15—11 12—13
odd-even exchange	4 5—8 14—11 15—12 13
even-odd exchange	4—5 8—11 14—12 15—13
odd-even exchange	4 5—8 11—12 14—13 15
even-odd exchange	4—5 8—11 12—13 14—15

Note. maximum distance to travel: (n – 1)

T(n) = O(n) lower-bound for this computer model
P(n) = n
C(n) = $O(n^2)$ optimal

(2) $p << n$

Ex. $S = \{8, 2, 5, 10, 1, 7, 3, 12, 6, 11, 4, 9\}$, $p = 4$

P_1	P_2	P_3	P_4
{8, 2, 5}	{10, 1, 7}	{3, 12, 6}	{11, 4, 9}
{2, 5, 8}	{1, 7, 10}	{3, 6, 12}	{4, 9, 11}
{1, 2, 5}	{7, 8, 10}	{3, 4, 6}	{9, 11, 12}
{1, 2, 5}	{3, 4, 6}	{7, 8, 10}	{9, 11, 12}
{1, 2, 3}	{4, 5, 6}	{7, 8, 9}	{10, 11, 12}
{1, 2, 3}	{4, 5, 6}	{7, 8, 9}	{10, 11, 12}

merge-split: Knuth (1973)
Baudet and Stevenson (1978)

Proc MERGE-SPLIT(s)

```
for i:= 1 to p do in parallel
    QUICKSORT(S_i)
endfor

for j:= 1 to ⌈p/2⌉ do
  for odd-numbered processors do in parallel
    MERGE(S_i, S_{i+1}, S_i')
    SPLIT
  endfor
  for even-numbered processors do in parallel
    MERGE(S_i, S_{i+1}, S_i')
    SPLIT
  endfor
endfor
```

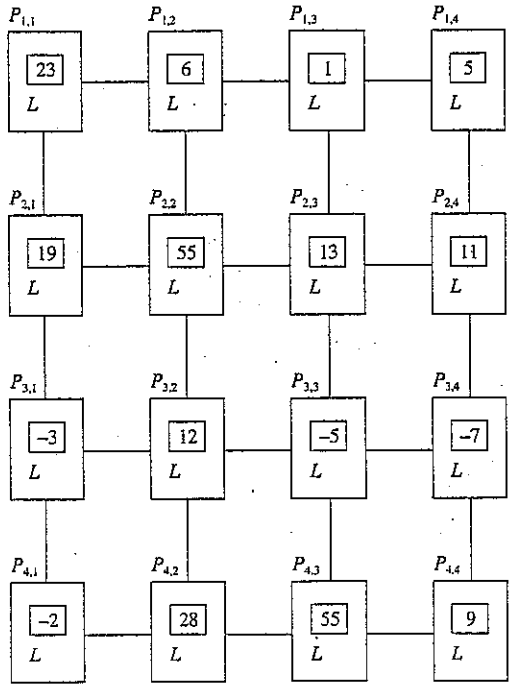
$T(n) = O(n/p \log (n/p)) + \lceil p/2 \rceil O(n/p)$

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

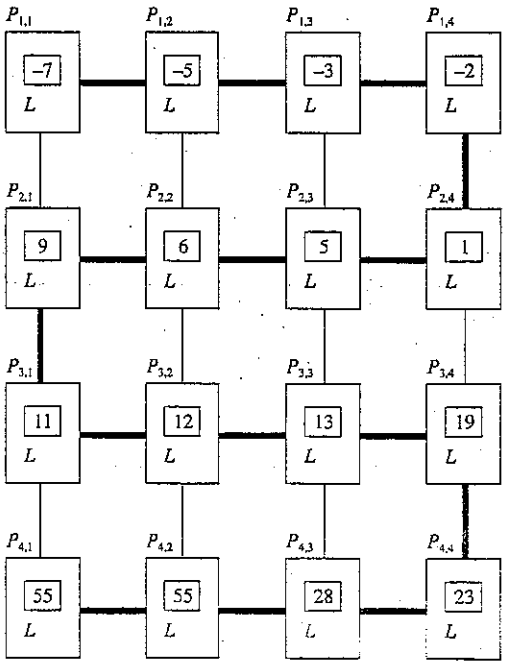
$C(n) = O(n \log n) + O(np)$ optimal when $p < \log n$

Sorting on 2-D Mesh

Figure 6.10 (a) The sample list 23,6,1,5,11,13, 55,19, −3,12, −5, −7,9,55,28, −2 of size 16 stored in the distributed variable L on the two-dimensional mesh $M_{4,4}$; (b) the sample list stored in the distributed variable L on the two dimensional mesh $M_{4,4}$ after snake ordering



(a)



(b)

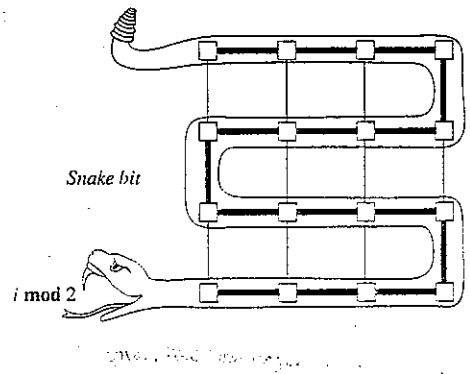


Figure 6.10 Continued

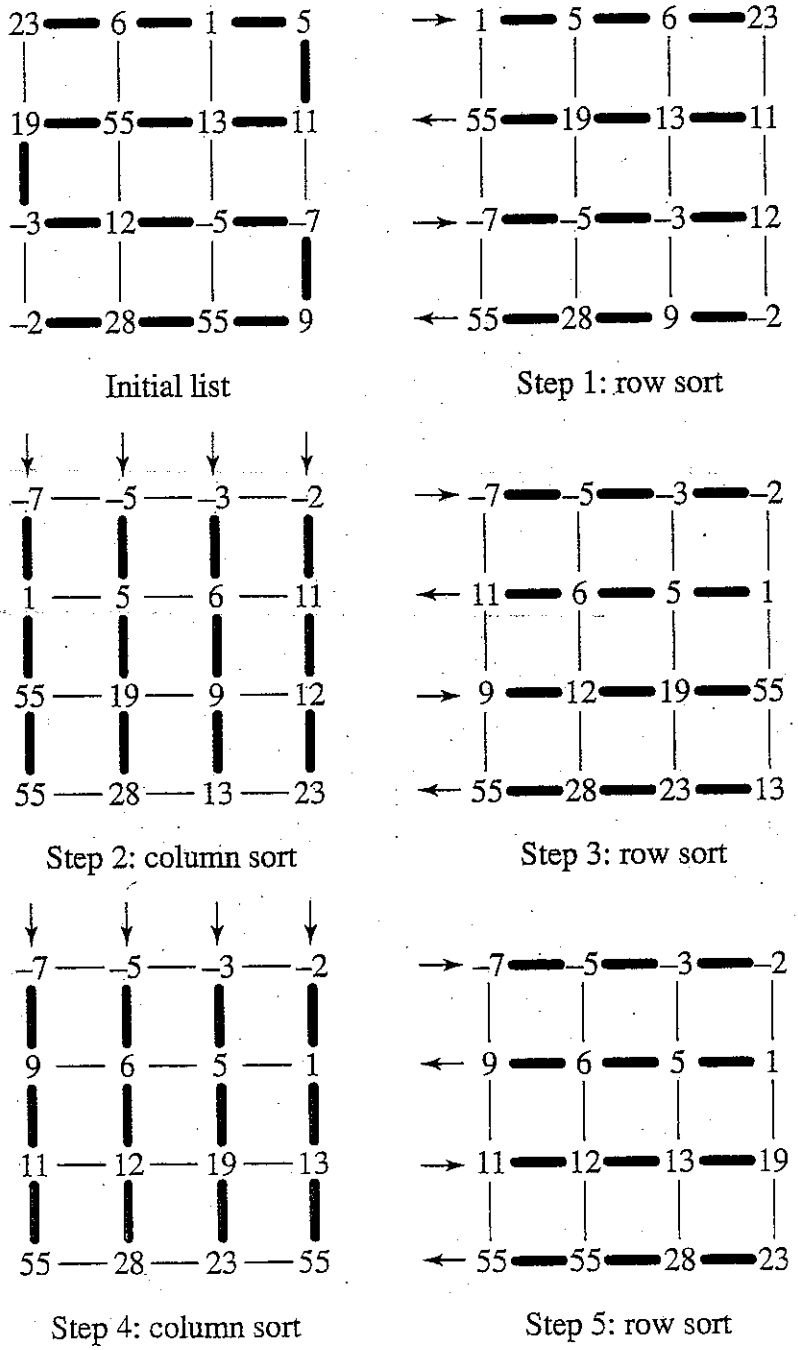


Figure 6.11 Action of *Sort2DMesh* for input list 23,6,1,5,11,13,55,19, – 3,12, – 5, – 7,9,55,28, – 2 stored in the distributed variable *L* on the two-dimensional mesh $M_{4,4}$

ENUMERATION SORTING

- Preparata (1978)

Each key is compared with all the others and the number of smaller keys determines the given key's final position.

Step 1. count acquisition
For each key the number of smaller keys is determined.

Step 2. rank computation
The final position of the key in the sorted sequence is determined as the sum of the counts obtained in Step 1.

Step 3. data rearrangement
Each key is placed in its final position according to its rank.

Example. (44, 7, 3, 21)

Step 1.

P_1	44:44	44:7	44:3	44:21	1 1 1 1
P_2	7:44	7:7	7:3	7:21	0 1 1 0
P_3	3:44	3:7	3:3	3:21	0 0 1 0
P_4	21:44	21:7	21:3	21:21	0 1 1 1

n^2 processors
O(1)

Step 2.

row 1 (44): 1 + 1 + 1 + 1 = 4
row 2 (7): 0 + 1 + 1 + 0 = 2
row 3 (3): 0 + 0 + 1 + 0 = 1
row 4 (21): 0 + 1 + 1 + 1 = 3

n^2 processors

O(1)	O(log n)
CRCW	CREW

Step 3.

(3 7 21 44)

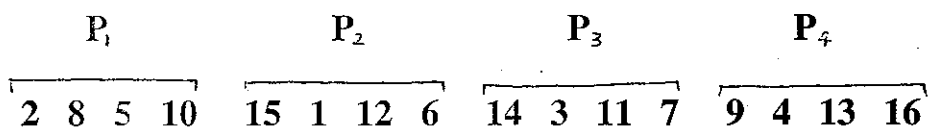
O(1): n processors

$$T(n) = \begin{cases} \text{O(1):} & \text{CRCW} \\ \text{O(log n):} & \text{CREW} \end{cases}$$

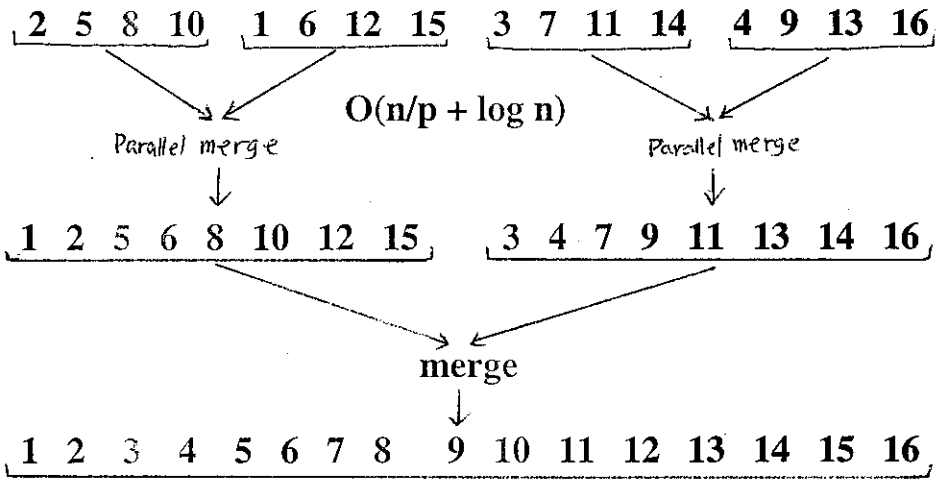
$$P(n) = n^2$$

CREW SORT

Ex $n = 16,\ p = 4$



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

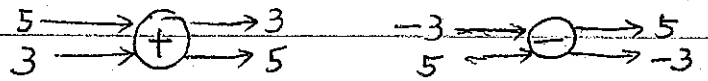


$$T(n) = O(n/p \log n/p) + O(n/p \log n + \log p \cdot \log n)$$

$$= O(n/p \log n + \log^2 n)$$

Bitonic Merge Sort (Batcher, 1968)

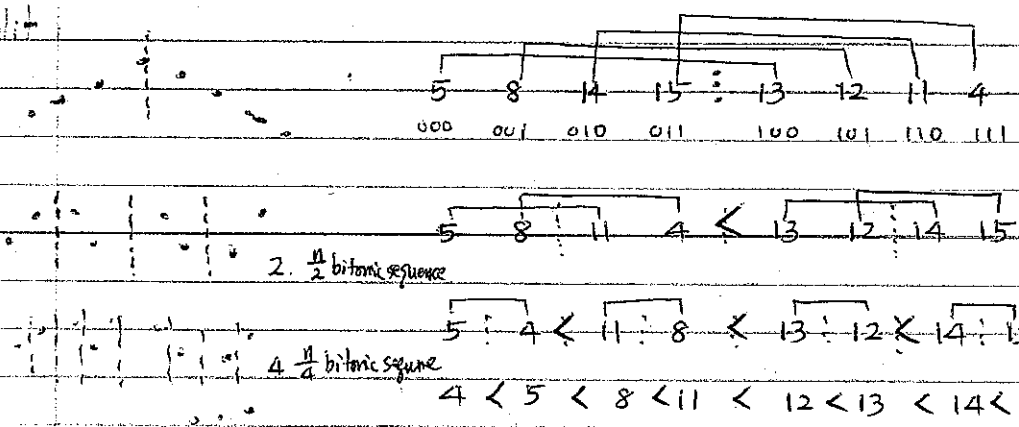
comparator (compare-exchange)



Bitonic sequence EX 5, 8, 14, 15, 13, 12, 11, 4

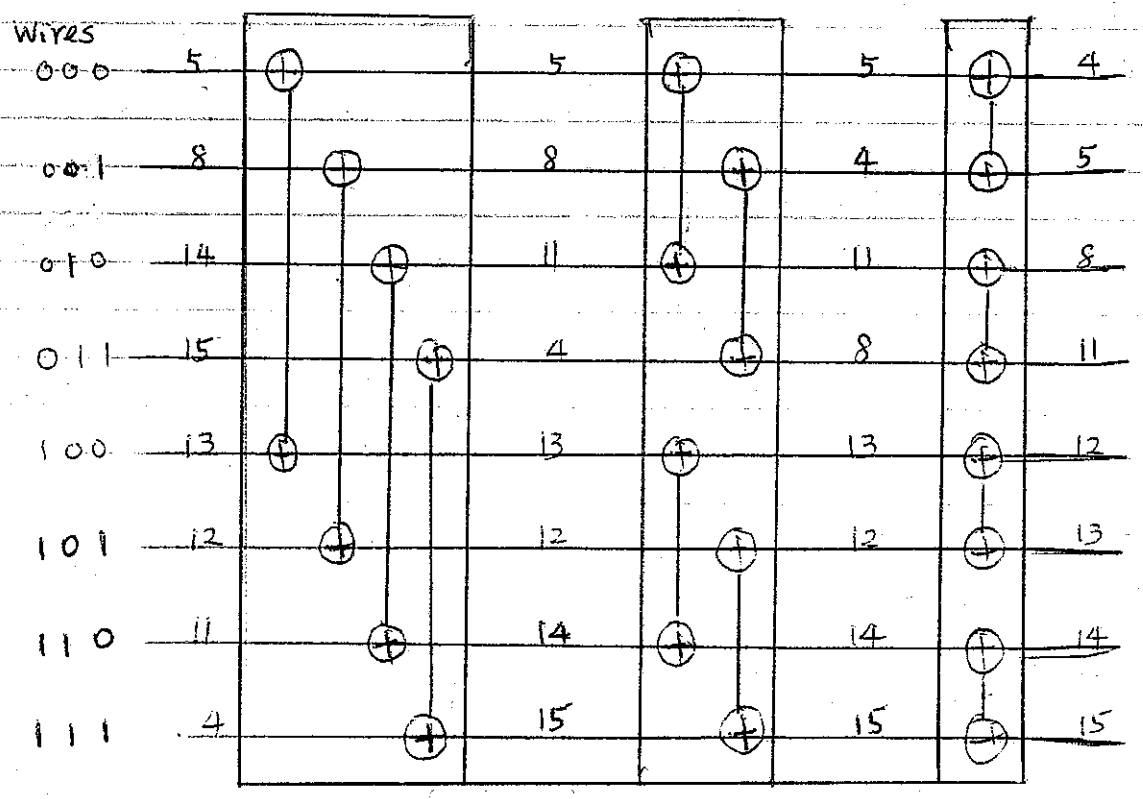
$n = 2^k$, $n=8, k=3$.

split



Note: Bitonic Merge compares elements whose indices differ in exactly 1 bit.

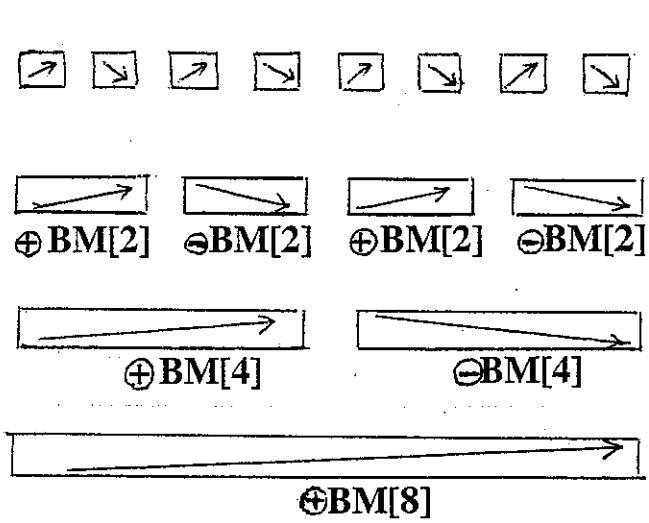
Bitonic Merge Network for n=8 (⊕ BM[8])



cf. Fig 6.22

Theorem. A list of $n = 2^K$ unsorted elements can be sorted by using a network of $2^{K-2}k(k+1)$ comparators in $O(\log^2 n)$. – polylogarithmic

Note. Bitonic sequence of 2^K elements (2^{K-1} in asc, 2^{K-1} in des) takes k steps.



total #steps: $1 + 2 + \ldots k = \frac{1}{2} k(k+1)$

each step needs $n/2 = 2^{K-1}$ comparators.

total #comparators = $2^{K-2}k(k+1)$

Time complexity (depth) = $\frac{1}{2} k(k+1) = \frac{1}{2} \log n (\log n + 1) = O(\log^2 n)$

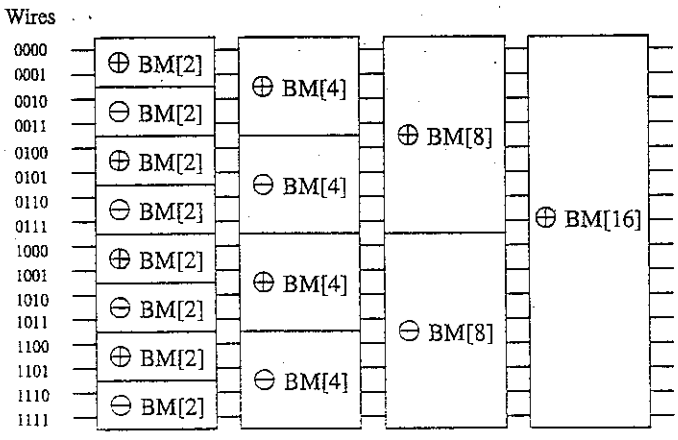
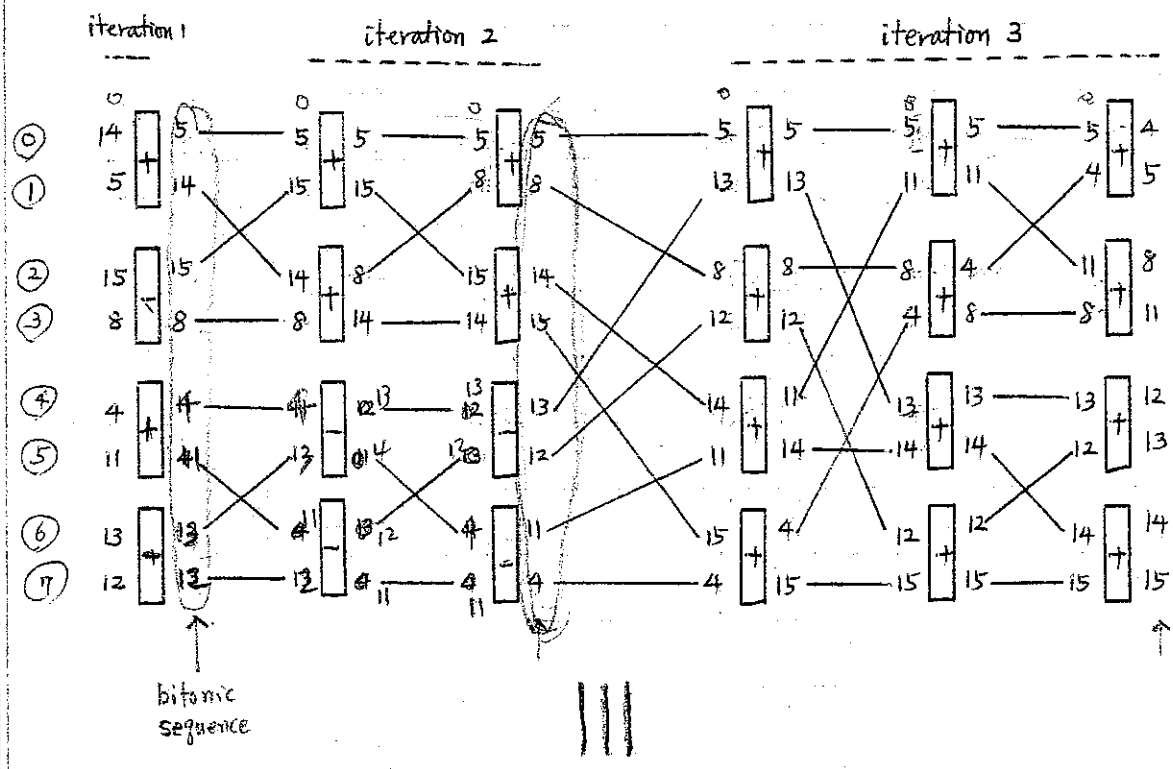
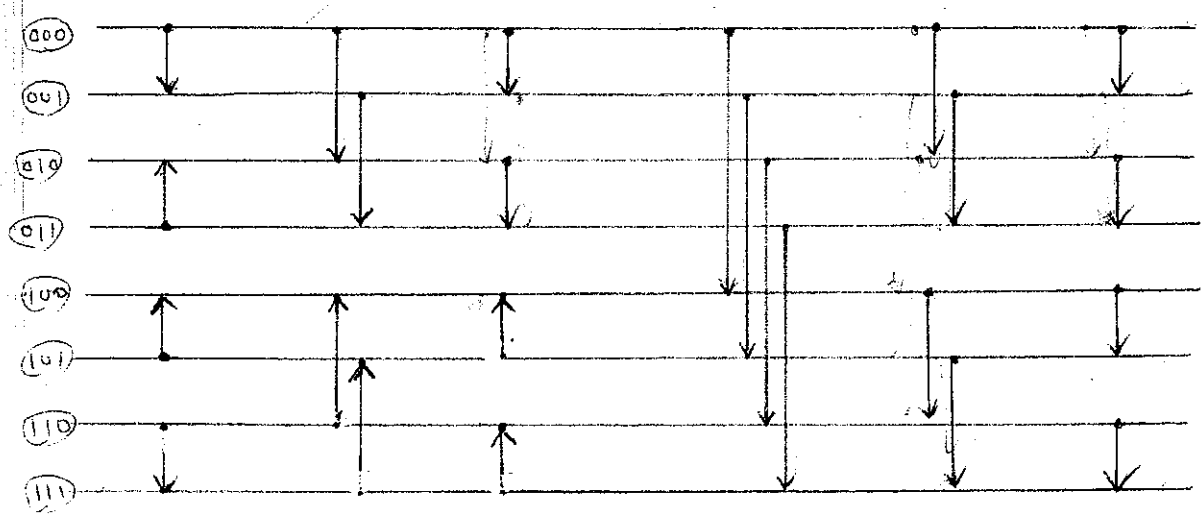


Figure 6.7 A schematic representation of a network that converts an input sequence into a bitonic sequence. In this example, ⊕BM[k] and ⊖BM[k] denote bitonic merging networks of input size k that use ⊕ and ⊖ comparators, respectively. The last merging network (⊕BM[16]) sorts the input. In this example, $n = 16$.

Example [14 5 15 8 4 11 13 12]

Bitonic Merge Sort





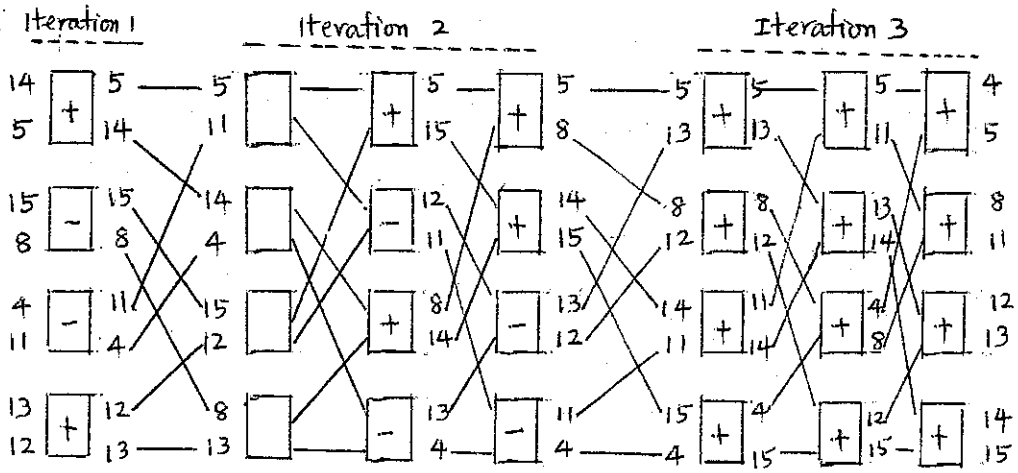
Sorting Network based on bitonic merge (Knuth 1973)

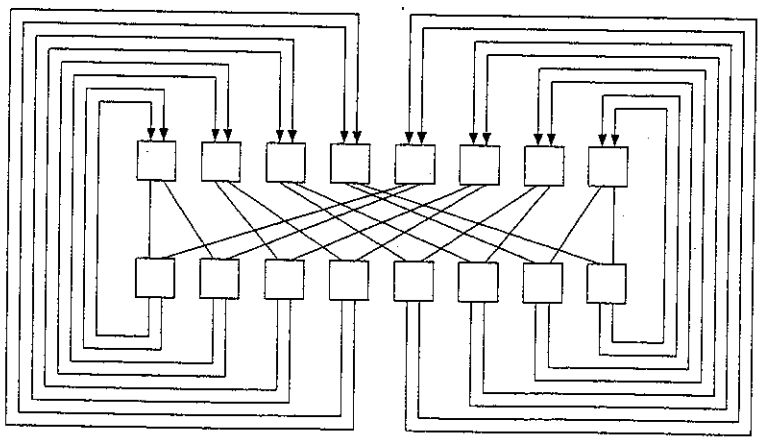
Bitonic Mergesort on Shuffle-Exchange Network

- Perfect Shuffle, Harold Stone (1971)

Theorem. A list of $n = 2^k$ unsorted elements can be sorted in time $\Theta(\log^2 n)$ with a network of $2^{k-1}[k(k-1) + 1]$ comparators using the shuffle-exchange network.

Ex [14, 5, 15, 8, 4, 11, 13, 12]





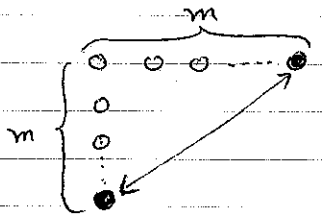
Sorting machine based upon perfect shuffle connection (Sedgewick 1983).

Bitonic Mergesort on 2-D Mesh

[Thompson and Kung] "Sorting on a mesh-connected parallel computer"
CACM 20(4), 1977. pp 263-271

Theorem 10.7 $n = m \times m = 2^K$, $T(n) = \Theta(\sqrt{n})$.

Proof



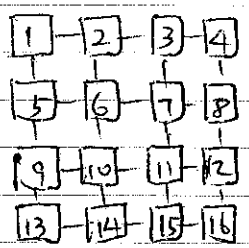
To exchange two diagonally opposite corner items, $4(n-1)$ data exchanges needed.

$\therefore T(n) = \Theta(\sqrt{n})$

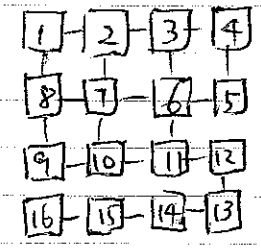
Proximity order of a mesh of size 16

0	1	14	15
3	2	13	12
4	11	8	11
5	6	9	10

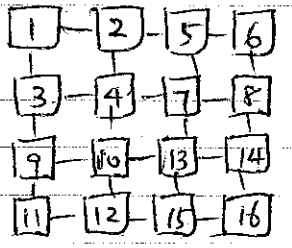
index function



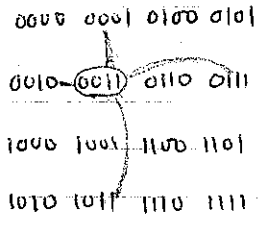
row-major



snake-like row-major



Shuffled row-major



one-bit difference neighbor 만 연결됨

Note: odd-even transposition sort can be efficiently implemented on snake-like row-major order on MC^2.

$\Theta(\log^2 n) + \Theta(\sqrt{n})$, $n = m^2$

(log n) (communication)

Adaptation of bitonic merge to the mesh using shuffled row-major order

[10, 9, 4, 15, 14, 2, 11, 12, 6, 1, 8, 3, 5, 13, 7, 0]

stage 1

10 → 9 14 → 2
4 ← 15 11 ← 12
6 → 1 5 → 13
8 ← 3 7 ← 0

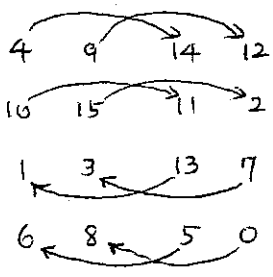
4 → 2 11 → 12
10 → 9 14 → 15
13 ← 8 5 ← 3
6 ← 7 1 ← 0

stage 2

9 ↓	10 ↓	2 ↑	14 ↑
15	4	12	11
1 ↓	6 ↓	5 ↑	13 ↑
8	3	7	0

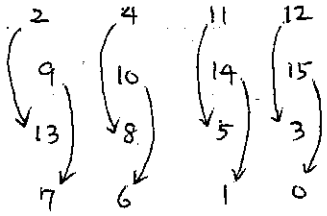
9 → 4 12 ← 14
15 → 10 2 ← 11
1 → 3 7 ← 13
8 → 6 5 ← 0

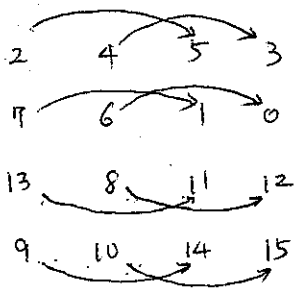
stage 3



4 ↓	9 ↓	14 ↓	12 ↓
10	2	11	15
13 ↑	7 ↑	1 ↑	3 ↑
6	8	5	0

stage 4





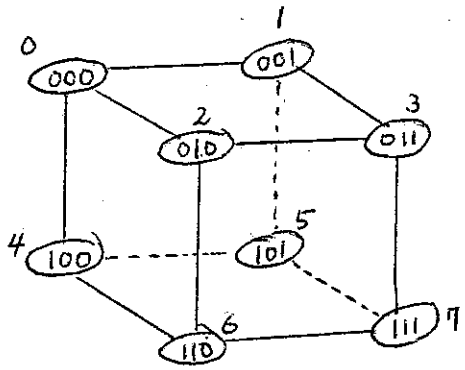
2 ↓	3 ↓	4 ↓	5 ↓
1	0	7	6
11 ↓	8 ↓	13 ↓	12 ↓
9	10	14	15

1 → 0 4 → 5
2 → 3 7 → 6
9 → 8 13 → 12
11 → 10 14 → 15

0	1	4	5
2	3	6	7
8	9	12	13
10	11	14	15

Bitonic Mergesort on Hypercube

Note. Bitonic merge always compares elements whose indices dffer in exactly one bit.



BITONIC MERGE SORT (HYPERCUBE PROCESSOR ARRAY):

```
  Global   d    {Distance between elements being compared}
  Local    a    {One of the elements to be sorted}
           t    {Element retrieved from adjacent processor}
begin
 for i ← 0 to m − 1 do
   for j ← i downto 0 do
     d ← 2^j
     for all P_k where 0 ≤ k ≤ 2^m − 1 do
       if k mod 2d < d then
         t ⇐ [k + d]a {Get value from adjacent processor}
         if k mod 2^(i+2) < 2^(i+1) then
           [k + d]a ⇐ max (t, a)   {Sort low to high...}
           a ← min (t, a)
         else
           [k + d]a ⇐ min (t, a)     {...or sort high to low}
           a ← max (t, a)
         endif
       endif
     endfor
   endfor
 endfor
end
```

T(n) = O(log²n)
P(n) = n

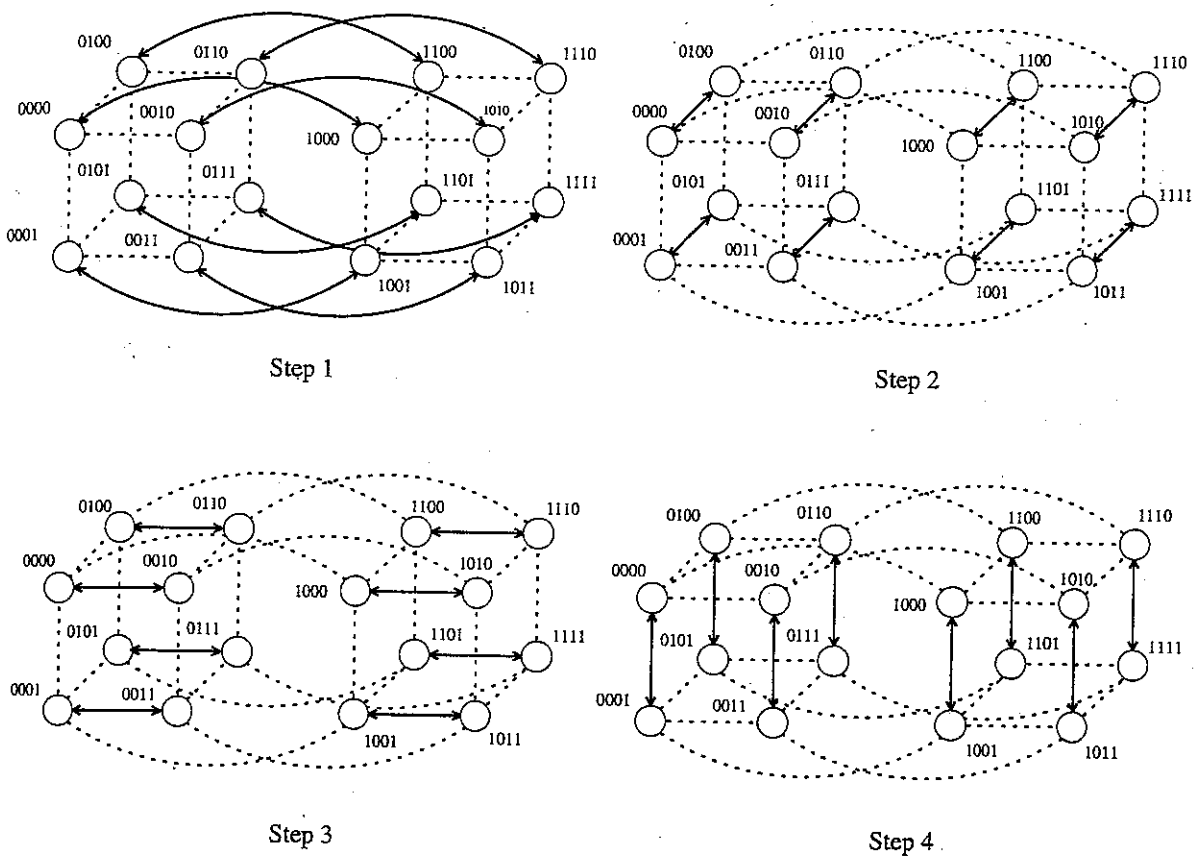
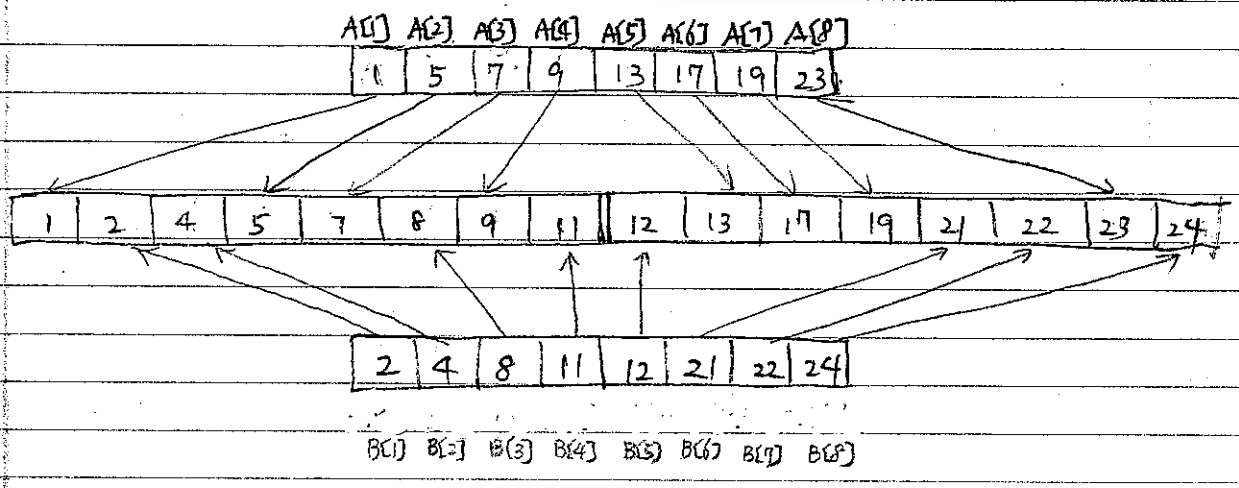


Figure 6.9 Communication during the last stage of bitonic sort. Each wire is mapped to a hypercube processor; each connection represents a compare-exchange between processors.

Merging two sorted list (2.3.5)

PRAM

Ex (Fig 2.16) $n=16$, $p=16$



Idea: Each process finds the position of its own element on the other list using binary search. Then add its own index and subtract $n/2$. That is the position of the element in the merged list.

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

$p(n) = n$

$C(n) = O(n \log n)$

Note: sequential algorithm : $C(n) = O(n)$

∴ not cost optimal