# Introduction to Parallel Computing

George Karypis

Sorting

# Outline

- Background
- Sorting Networks
- Quicksort
- Bucket-Sort & Sample-Sort

# Background

- Input Specification
  - Each processor has n/p elements
  - A ordering of the processors
- Output Specification
  - Each processor will get n/p consecutive elements of the final sorted array.
  - The “chunk” is determined by the processor ordering.
- Variations
  - Unequal number of elements on output.
    - In general, this is not a good idea and it may require a shift to obtain the equal size distribution.

# Basic Operation: Compare-Split Operation

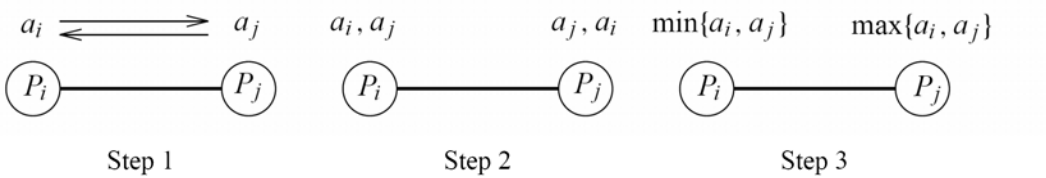



**Figure 9.1** A parallel compare-exchange operation. Processes $P_i$ and $P_j$ send their elements to each other. Process $P_i$ keeps $\min\{a_i, a_j\}$, and $P_j$ keeps $\max\{a_i, a_j\}$.

Single element per processor

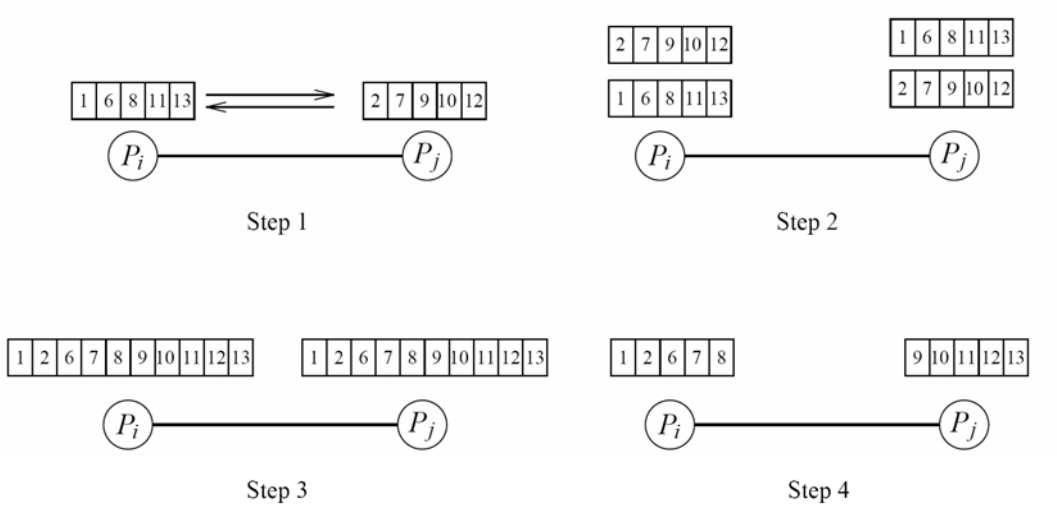



**Figure 9.2** A compare-split operation. Each process sends its block of size $n/p$ to the other process. Each process merges the received block with its own block and retains only the appropriate half of the merged block. In this example, process $P_i$ retains the smaller elements and process $P_j$ retains the larger elements.

Multiple elements per processor

# Sorting Networks

- Sorting is one of the fundamental problems in Computer Science
- For a long time researchers have focused on the problem of "how fast can we sort *n* elements"?
  - Serial
    - nlog(n) lower-bound for comparison-based sorting
  - Parallel
    - O(1), O(log(n)), O(???)
- Sorting networks
  - Custom-made hardware for sorting!
    - Hardware & algorithm
    - Mostly of theoretical interest but fun to study!

# Elements of Sorting Networks

- Key Idea:
  - Perform many comparisons in parallel.
- Key Elements:
  - Comparators:
    - Consist of two-input, two-output wires
    - Take two elements on the input wires and outputs them in sorted order in the output wires.
  - Network architecture:
    - The arrangement of the comparators into interconnected comparator columns
      - similar to multi-stage networks
- Many sorting networks have been developed.
  - Bitonic sorting network
    - $\Theta(\log^2(n))$ columns of comparators.

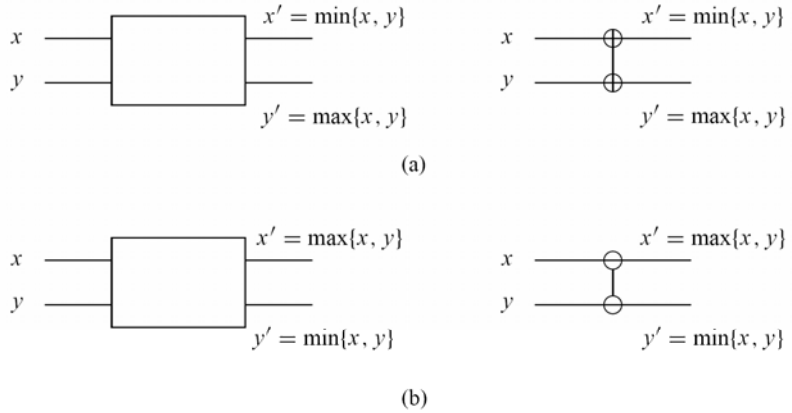



**Figure 9.3** A schematic representation of comparators: (a) an increasing comparator, and (b) a decreasing comparator.

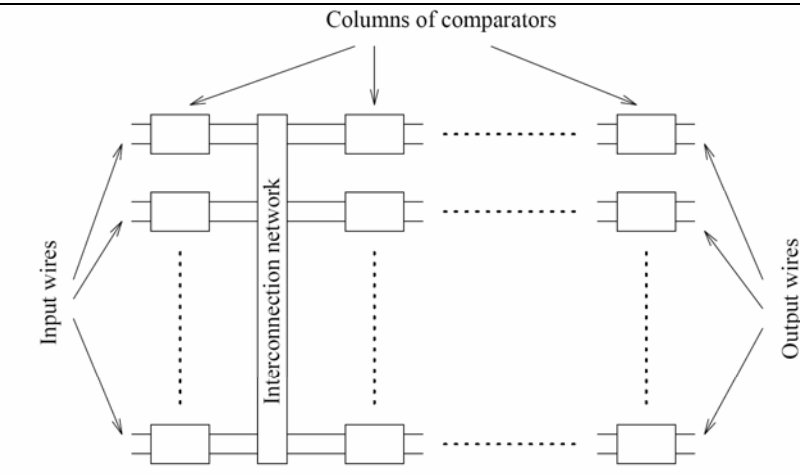
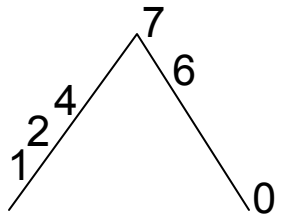



**Figure 9.4** A typical sorting network. Every sorting network is made up of a series of columns, and each column contains a number of comparators connected in parallel.

# Bitonic Sequence

A ***bitonic sequence*** is a sequence of elements $\langle a_0, a_1, \ldots, a_{n-1} \rangle$ with the property that either (1) there exists an index $i$, $0 \leq i \leq n-1$, such that $\langle a_0, \ldots, a_i \rangle$ is monotonically increasing and $\langle a_{i+1}, \ldots, a_{n-1} \rangle$ is monotonically decreasing, or (2) there exists a cyclic shift of indices so that (1) is satisfied. For example, $\langle 1, 2, 4, 7, 6, 0 \rangle$ is a bitonic sequence, because it first increases and then decreases. Similarly, $\langle 8, 9, 2, 1, 0, 4 \rangle$ is another bitonic sequence, because it is a cyclic shift of $\langle 0, 4, 8, 9, 2, 1 \rangle$.

Bitonic sequences are graphically represented by lines as follows:



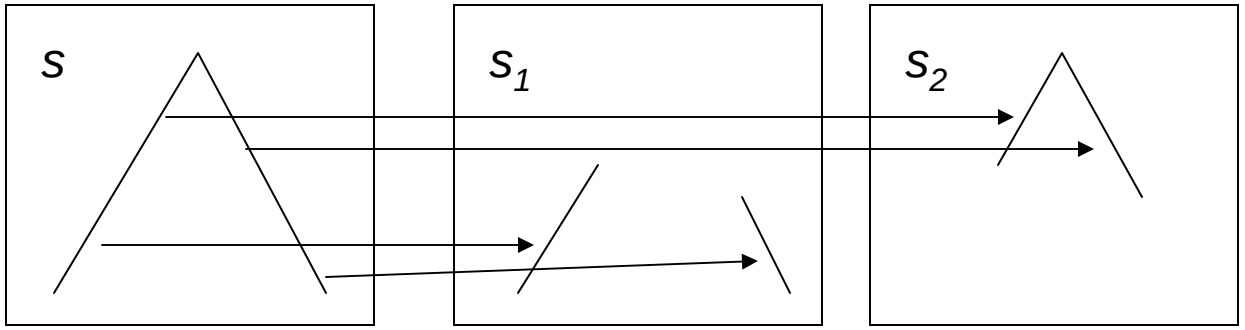
# Why Bitonic Sequences?

- A bitonic sequence can be "easily" sorted in increasing/decreasing order.

Let $s = \langle a_0, a_1, \ldots, a_{n-1} \rangle$ be a bitonic sequence such that $a_0 \leq a_1 \leq \ldots \leq a_{n/2-1}$ and $a_{n/2} \geq a_{n/2+1} \geq \ldots \geq a_{n-1}$. Consider the following subsequences of $s$:

$$\begin{aligned} s_1 &= \langle \min\{a_0, a_{n/2}\}, \min\{a_1, a_{n/2+1}\}, \ldots, \min\{a_{n/2-1}, a_{n-1}\} \rangle \\ s_2 &= \langle \max\{a_0, a_{n/2}\}, \max\{a_1, a_{n/2+1}\}, \ldots, \max\{a_{n/2-1}, a_{n-1}\} \rangle \end{aligned} \quad (9.1)$$

Bitonic Split




- Every element of $s_1$ will be less than or equal to every element of $s_2$
- Both $s_1$ and $s_2$ are bitonic sequences.
- So how can a bitonic sequence be sorted?

# An example

| | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 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.

# Bitonic Merging Network

- A comparator network that takes as input a bitonic sequence and performs a sequence of bitonic splits to sort it.
  - +BM[n]
    - A bitonic merging network for sorting in increasing order an *n*-element bitonic sequence.
  - -BM[n]
    - Similar sort in decreasing order.

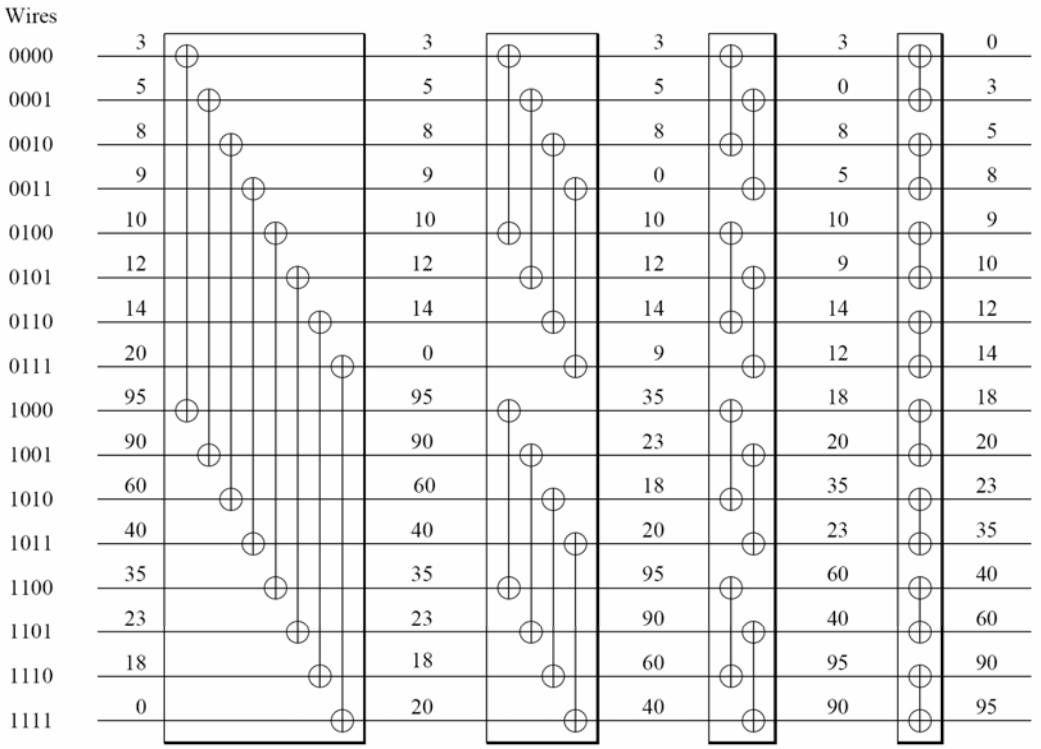



**Figure 9.6** A bitonic merging network for $n = 16$. The input wires are numbered $0, 1 \ldots, n-1$, and the binary representation of these numbers is shown. Each column of comparators is drawn separately; the entire figure represents a $\oplus$BM[16] bitonic merging network. The network takes a bitonic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

# Are we done?

- Given a set of elements, how do we re-arrange them into a bitonic sequence?
- Key Idea:
  - Use successively larger bitonic networks to transform the set into a bitonic sequence.

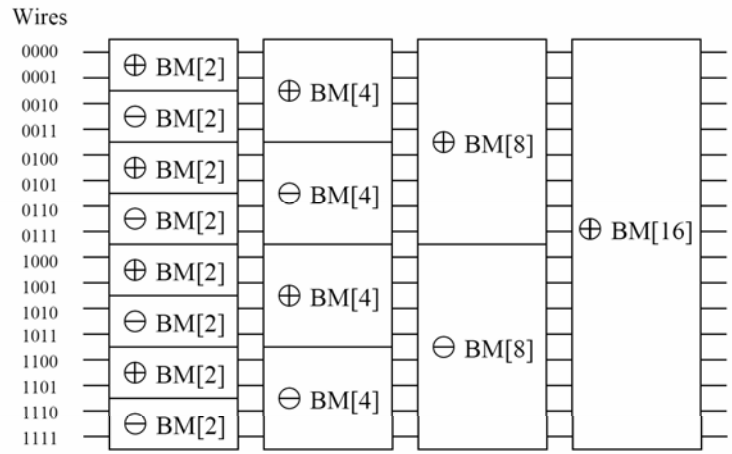



**Figure 9.7** A schematic representation of a network that converts an input sequence into a bitonic sequence. In this example, ⊕BM[k] and ⊖BM[k] denote bitonic merging networks of input size $k$ that use ⊕ and ⊖ comparators, respectively. The last merging network (⊕BM[16]) sorts the input. In this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

# An example

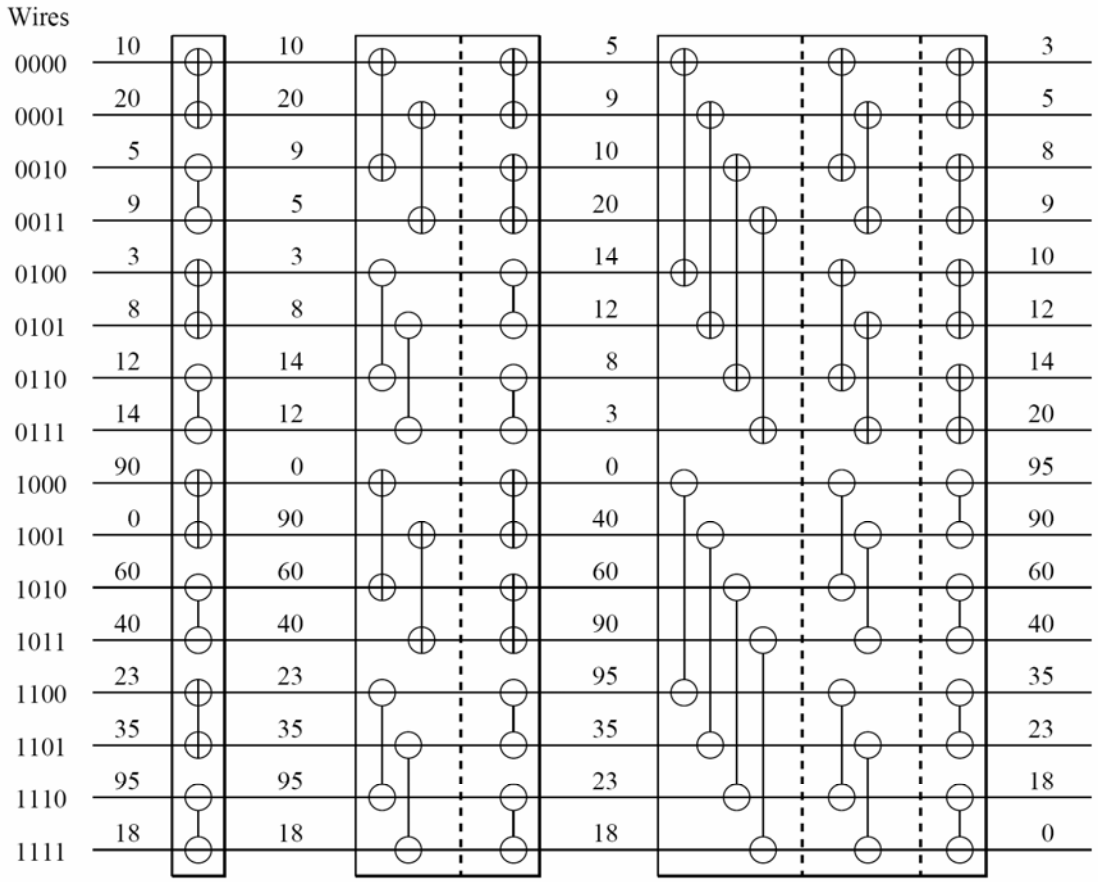
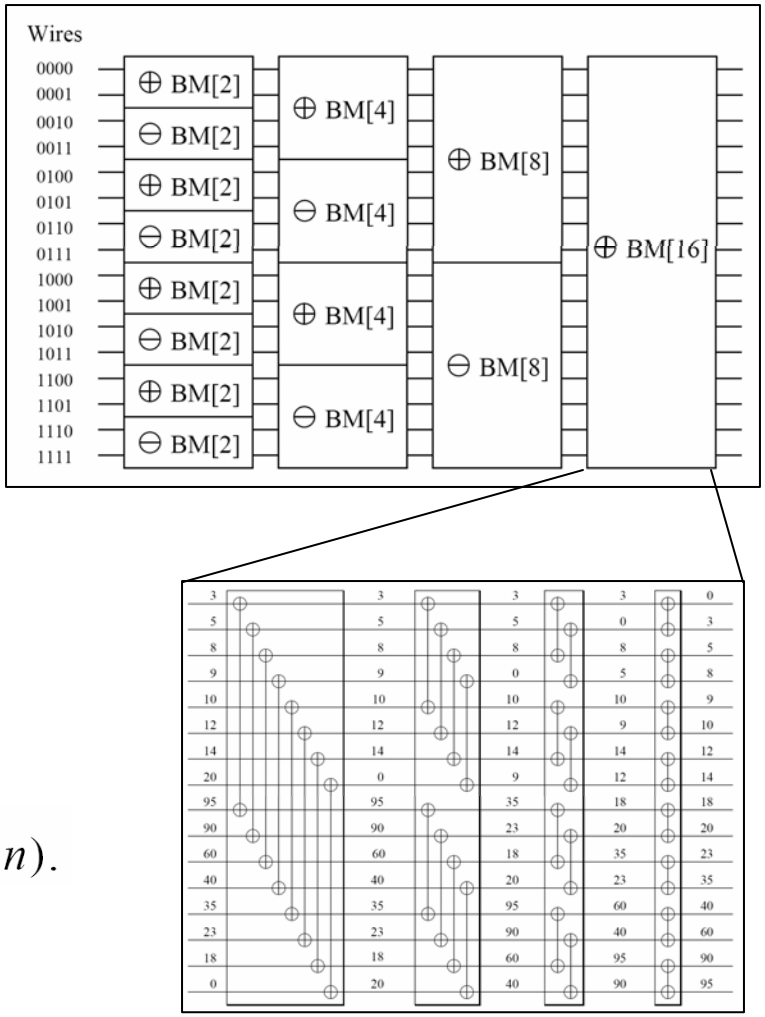



**Figure 9.8** The comparator network that transforms an input sequence of 16 unordered numbers into a bitonic sequence. In contrast to Figure 9.6, the columns of comparators in each bitonic merging network are drawn in a single box, separated by a dashed line.

# Complexity

- How many columns of comparators are required to sort $n=2^l$ elements?
  - i.e., depth $d(n)$ of the network?

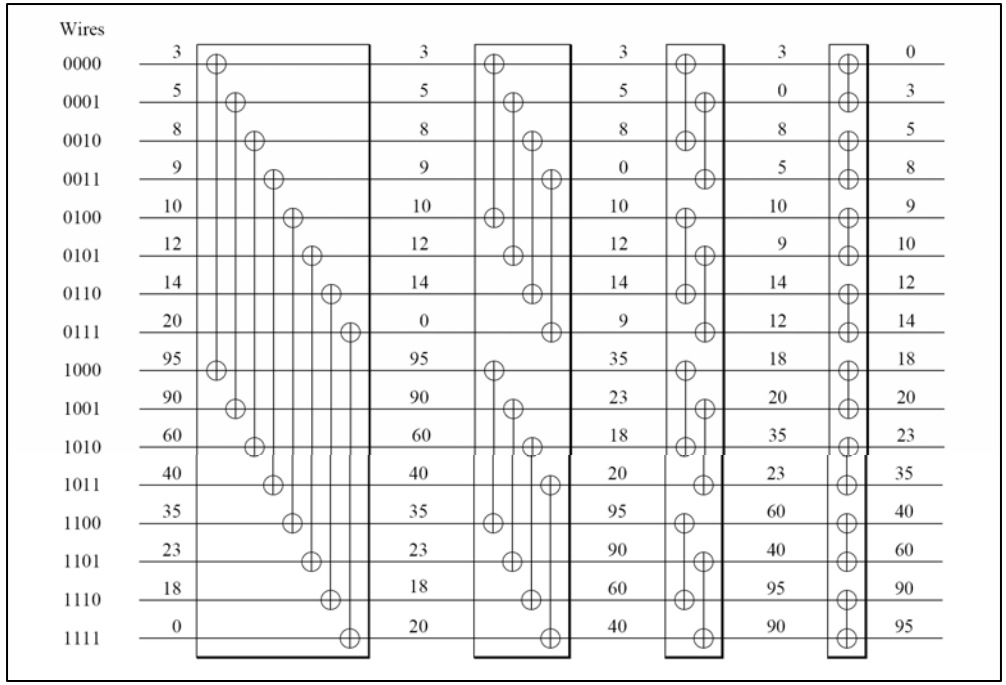



$$d(n) = d(n/2) + \log n$$

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

# Bitonic Sort on a Hypercube

- One-element-per-processor case
  - How do we map the algorithm onto a hypercube?
    - What is the comparator?
    - How do the wires get mapped?




What can you say about the pairs of wires that are inputs to the various comparators?

# Illustration

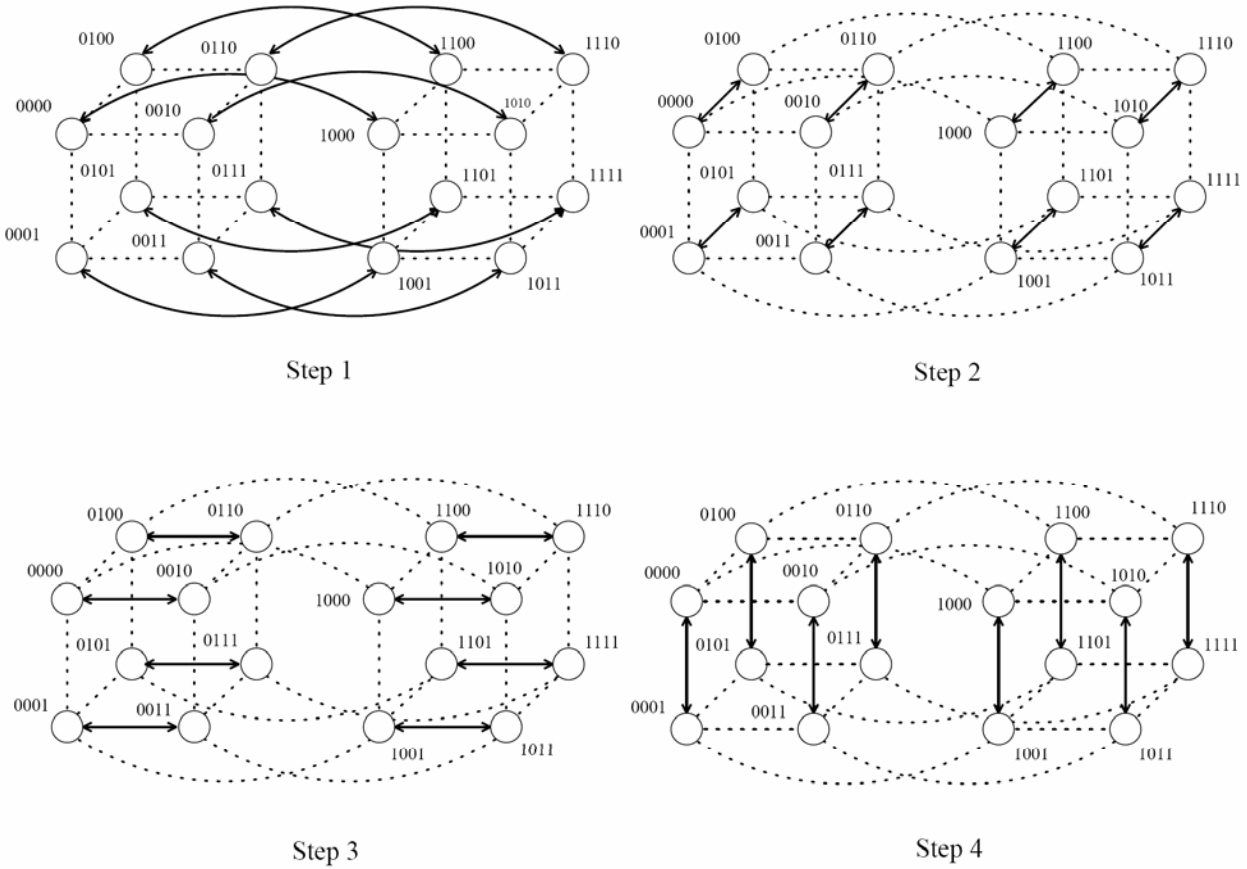



**Figure 9.9** Communication during the last stage of bitonic sort. Each wire is mapped to a hypercube process; each connection represents a compare-exchange between processes.

# Communication Pattern

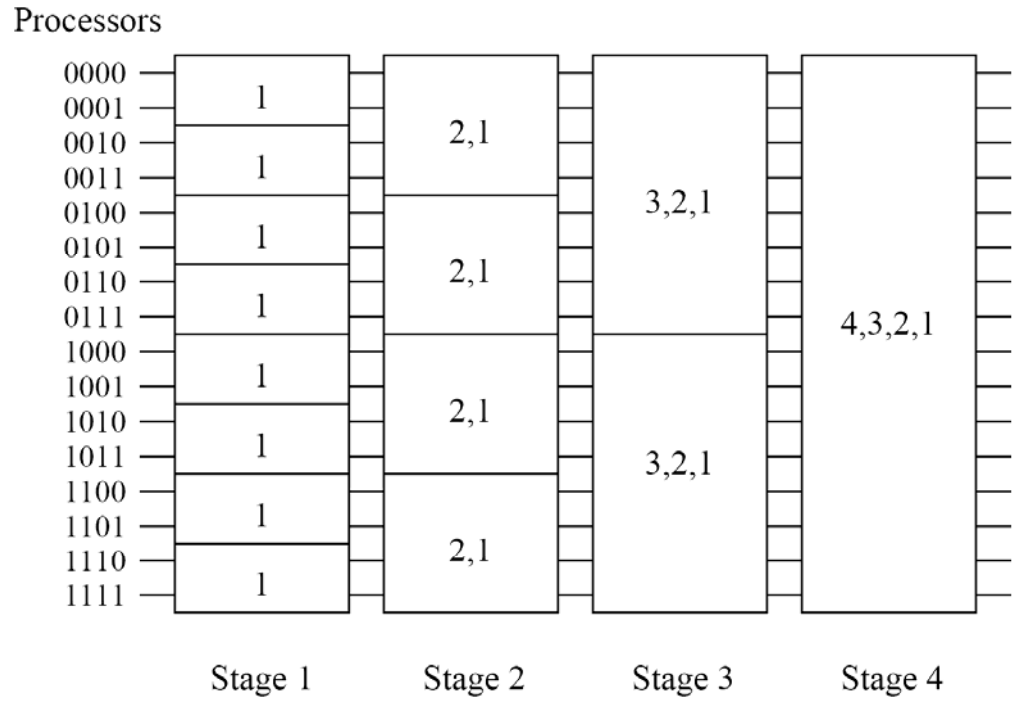



**Figure 9.10** Communication characteristics of bitonic sort on a hypercube. During each stage of the algorithm, processes communicate along the dimensions shown.

# Algorithm

```
1.  procedure BITONIC_SORT(label, d)
2.  begin
3.      for i := 0 to d − 1 do
4.          for j := i downto 0 do
5.              if (i + 1)st bit of label ≠ jth bit of label then
6.                  comp_exchange_max(j);
7.              else
8.                  comp_exchange_min(j);
9.  end BITONIC_SORT
```

**Algorithm 9.1** Parallel formulation of bitonic sort on a hypercube with $n = 2^d$ processes. In this algorithm, *label* is the process's label and $d$ is the dimension of the hypercube.

Complexity?

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

# Bitonic Sort on a Mesh

- One-element-per-processor case
  - How do the wires get mapped?

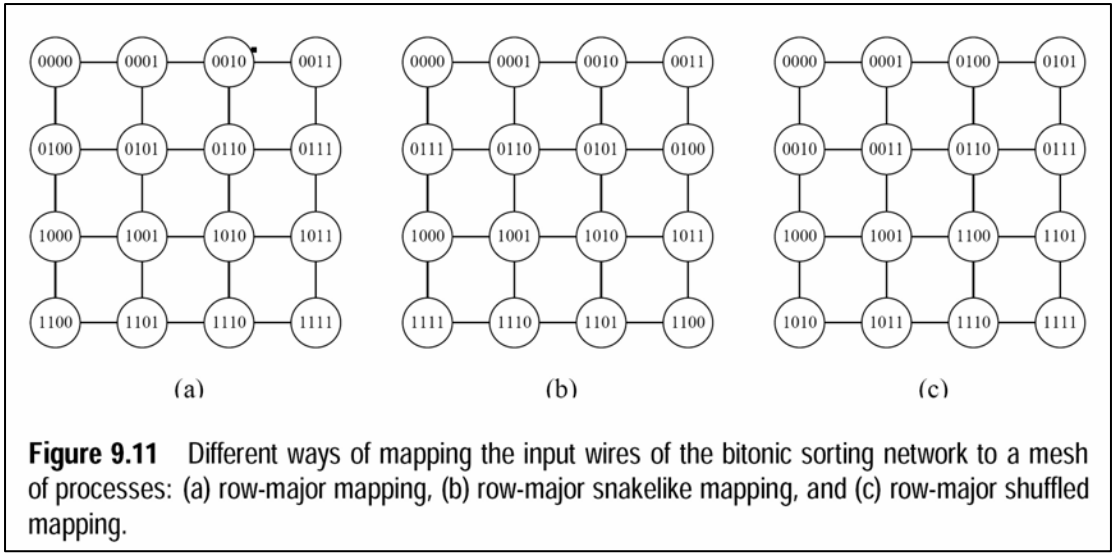
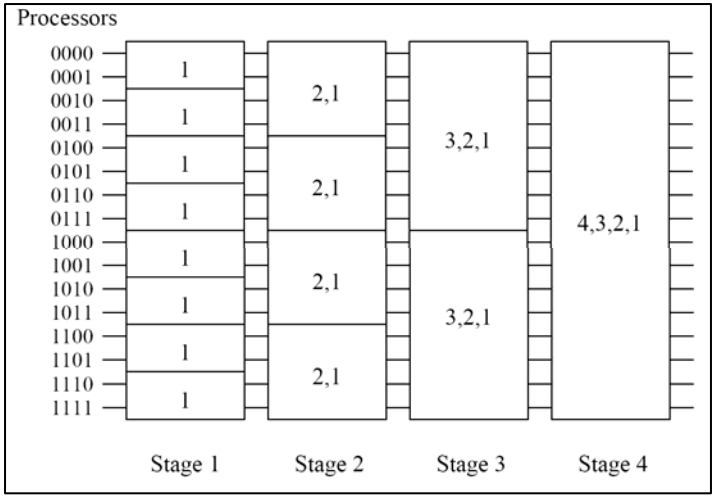



**Figure 9.11** Different ways of mapping the input wires of the bitonic sorting network to a mesh of processes: (a) row-major mapping, (b) row-major snakelike mapping, and (c) row-major shuffled mapping.

Which one is better?
Why?



# Row-Major Shuffled Mapping

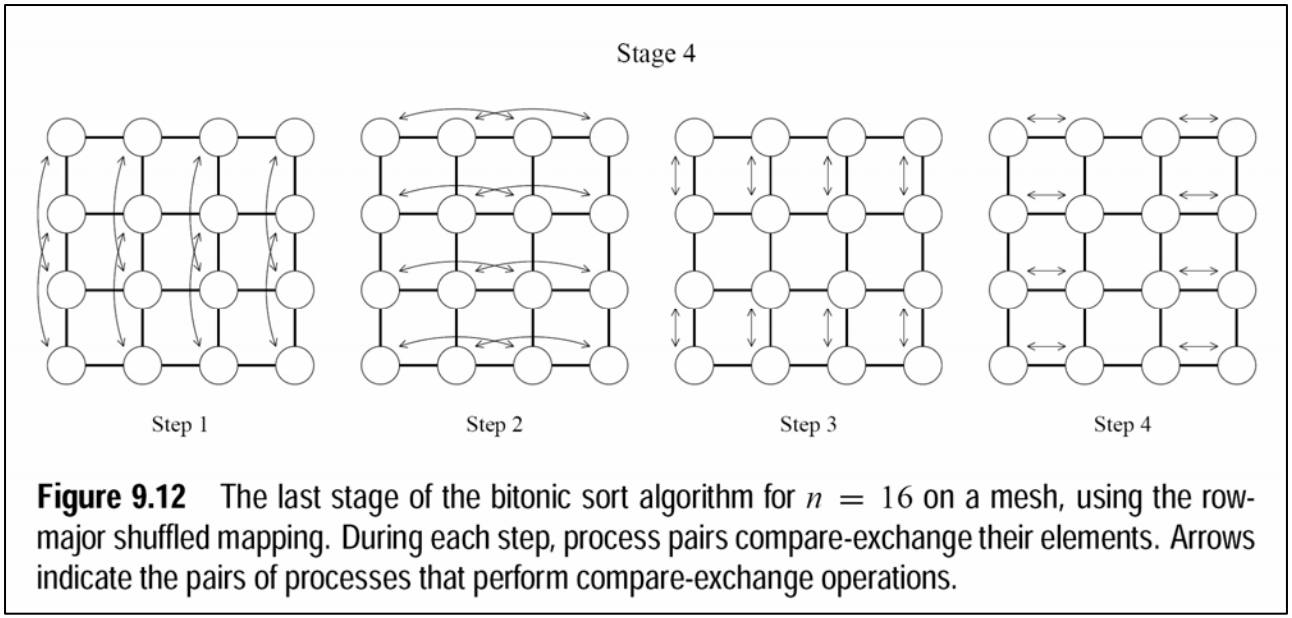



**Figure 9.12** The last stage of the bitonic sort algorithm for $n = 16$ on a mesh, using the row-major shuffled mapping. During each step, process pairs compare-exchange their elements. Arrows indicate the pairs of processes that perform compare-exchange operations.

Complexity?

that differ in the least-significant bit) are neighbors. In general, wires that differ in the $i^{\text{th}}$ least-significant bit are mapped onto mesh processes that are $2^{\lfloor (i-1)/2 \rfloor}$ communication links away. The compare-exchange steps of the last stage of bitonic sort for the row-major

$$\sum_{i=1}^{\log n} \sum_{j=1}^{i} 2^{\lfloor (j-1)/2 \rfloor} \approx 7\sqrt{n},$$

communication performed by each process

Can we do better?

What is the lowest bound of sorting on a mesh?

# More than one element per processor

- Hypercube

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

- Mesh

$$T_P = \overbrace{\Theta\left(\frac{n}{p}\log\frac{n}{p}\right)}^{\text{local sort}} + \overbrace{\Theta\left(\frac{n}{p}\log^2 p\right)}^{\text{comparisons}} + \overbrace{\Theta\left(\frac{n}{\sqrt{p}}\right)}^{\text{communication}}$$

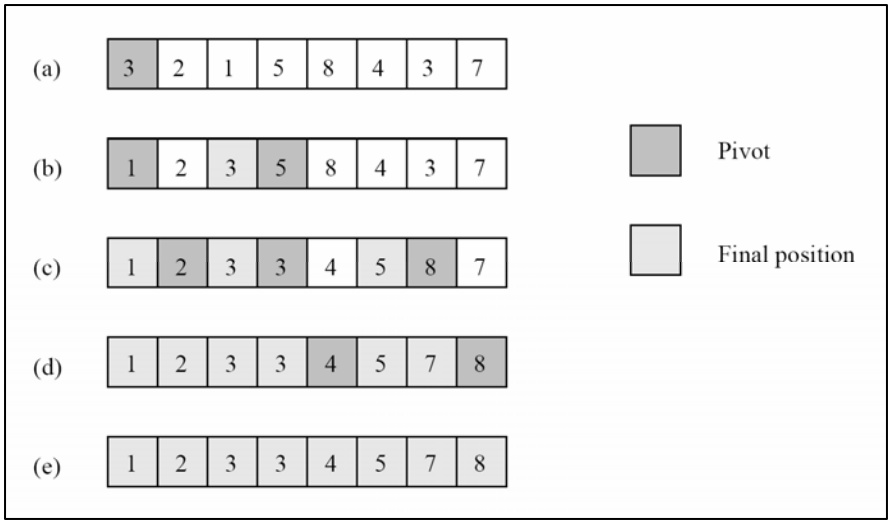
# Bitonic Sort Summary

**Table 9.1** The performance of parallel formulations of bitonic sort for $n$ elements on $p$ processes.

| Architecture | Maximum Number of Processes for $E = \Theta(1)$ | Corresponding Parallel Run Time | Isoefficiency Function |
|---|---|---|---|
| Hypercube | $\Theta(2^{\sqrt{\log n}})$ | $\Theta(n/(2^{\sqrt{\log n}}) \log n)$ | $\Theta(p^{\log p} \log^2 p)$ |
| Mesh | $\Theta(\log^2 n)$ | $\Theta(n/\log n)$ | $\Theta(2^{\sqrt{p}} \sqrt{p})$ |
| Ring | $\Theta(\log n)$ | $\Theta(n)$ | $\Theta(2^p p)$ |

# Quicksort

```
1.   procedure QUICKSORT (A, q, r)
2.   begin
3.       if q < r then
4.       begin
5.           x := A[q];
6.           s := q;
7.           for i := q + 1 to r do
8.               if A[i] ≤ x then
9.               begin
10.                  s := s + 1;
11.                  swap(A[s], A[i]);
12.              end if
13.          swap(A[q], A[s]);
14.          QUICKSORT (A, q, s);
15.          QUICKSORT (A, s + 1, r);
16.      end if
17.  end QUICKSORT
```

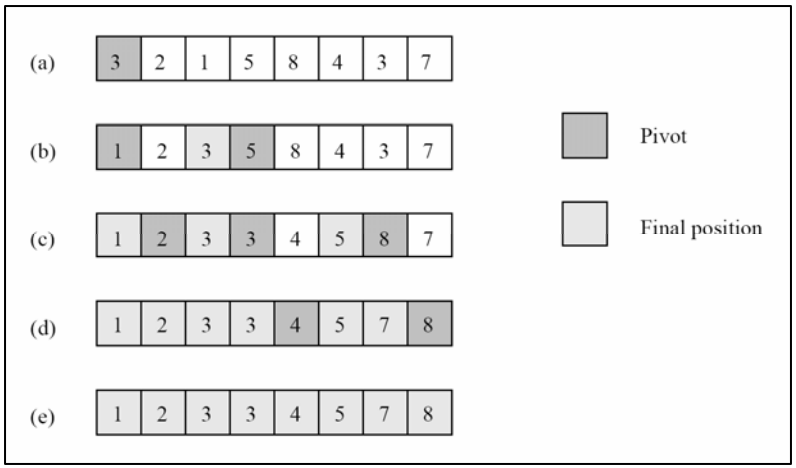


# Parallel Formulation

- How about recursive decomposition?
  - Is it a good idea?
    - We need to do the partitioning of the array around a pivot element in parallel.
- What is the lower bound of parallel quicksort?
  - What will it take to achieve this lower bound?

# Optimal for CRCW PRAM

- One element per processor
- Arbitrary resolution of the concurrent writes.
- Views the sorting as a two-step process:
  - (i) Constructing a binary tree of pivot elements
  - (ii) Obtaining the sorted sequence by performing an inorder traversal of this binary tree.



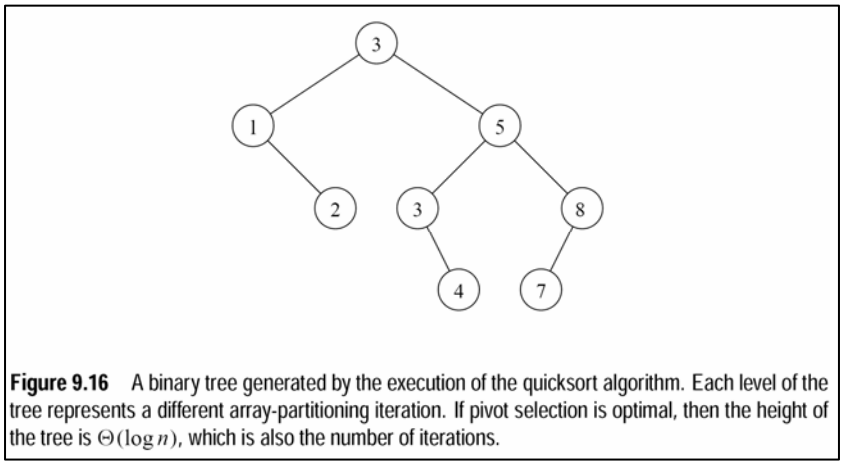




**Figure 9.16** A binary tree generated by the execution of the quicksort algorithm. Each level of the tree represents a different array-partitioning iteration. If pivot selection is optimal, then the height of the tree is $\Theta(\log n)$, which is also the number of iterations.

# Building the Binary Tree

```
1.  procedure BUILD_TREE (A[1 ... n])
2.  begin
3.      for each process i do
4.      begin
5.          root := i;
6.          parent_i := root;
7.          leftchild[i] := rightchild[i] := n + 1;
8.      end for
9.      repeat for each process i ≠ root do
10.     begin
11.         if (A[i] < A[parent_i]) or
                (A[i] = A[parent_i] and i <parent_i) then
12.         begin
13.             leftchild[parent_i] := i;
14.             if i = leftchild[parent_i] then exit
15.             else parent_i := leftchild[parent_i];
16.         end for
17.         else
18.         begin
19.             rightchild[parent_i] := i;
20.             if i = rightchild[parent_i] then exit
21.             else parent_i := rightchild[parent_i];
22.         end else
23.     end repeat
24. end BUILD_TREE
```

**Algorithm 9.6** The binary tree construction procedure for the CRCW PRAM parallel quicksort formulation.

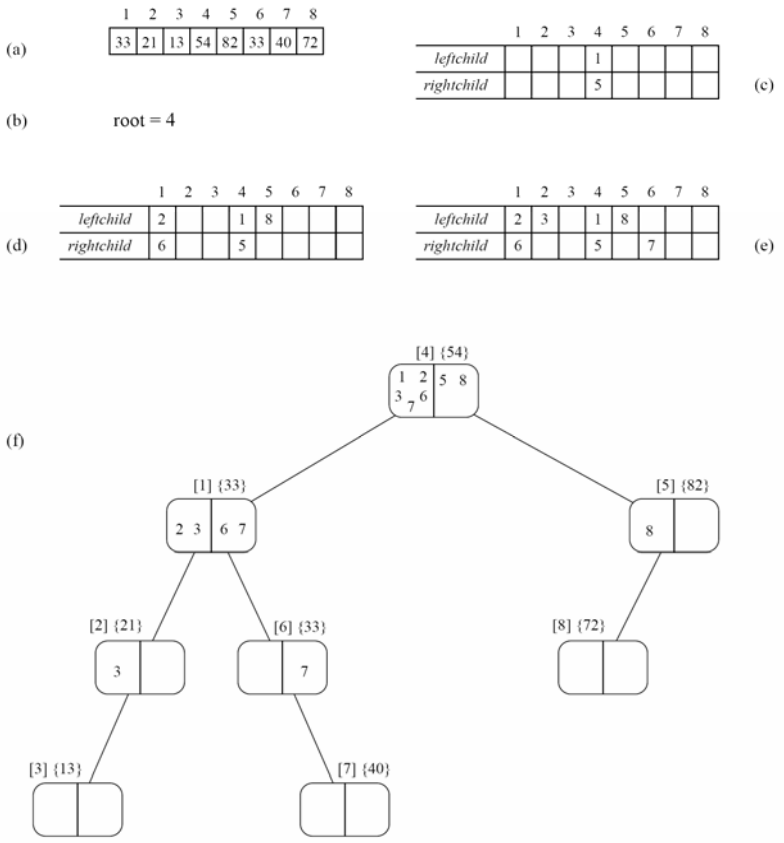



**Figure 9.17** The execution of the PRAM algorithm on the array shown in (a). The arrays *leftchild* and *rightchild* are shown in (c), (d), and (e) as the algorithm progresses. Figure (f) shows the binary tree constructed by the algorithm. Each node is labeled by the process (in square brackets), and the element is stored at that process (in curly brackets). The element is the pivot. In each node, processes with smaller elements than the pivot are grouped on the left side of the node, and those with larger elements are grouped on the right side. These two groups form the two partitions of the original array. For each partition, a pivot element is selected at random from the two groups that form the children of the node.

Complexity?

$$\Theta(\log n)$$

# Practical Quicksort

- Shared-memory
  - Data resides on a shared array.
  - During a partitioning each processor is responsible for a certain portion.
- Array Partitioning:
  - Select & Broadcast pivot.
  - Local re-arrangement.
    - Is this required?
  - Global re-arrangement.

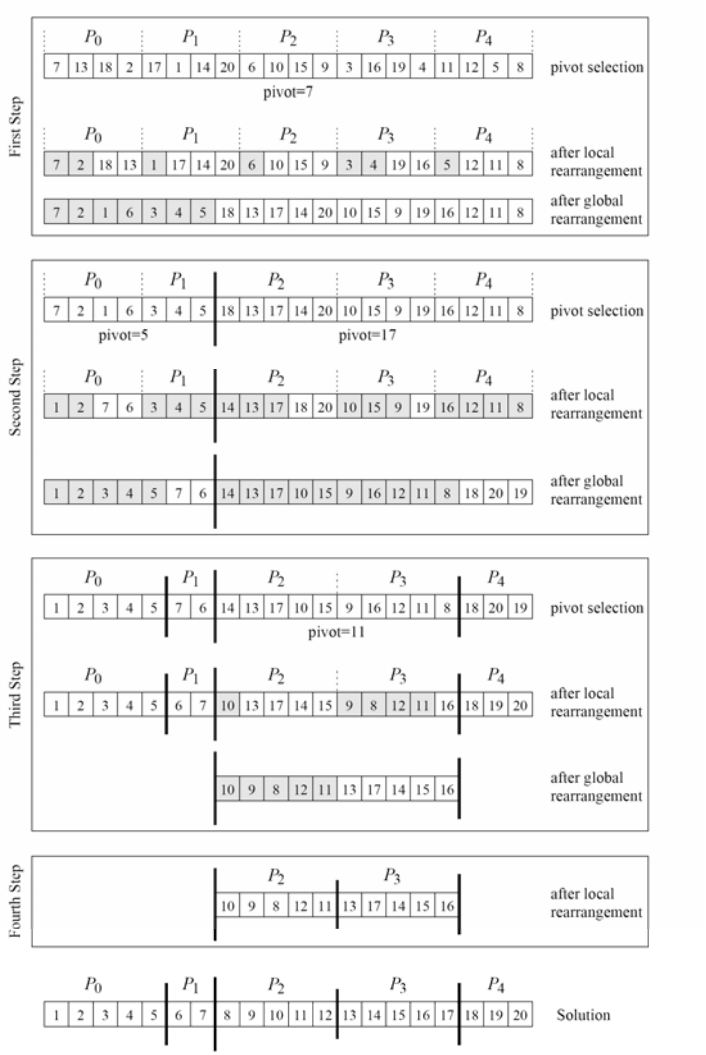



**Figure 9.18** An example of the execution of an efficient shared-address-space quicksort algorithm.

# Efficient Global Rearrangement

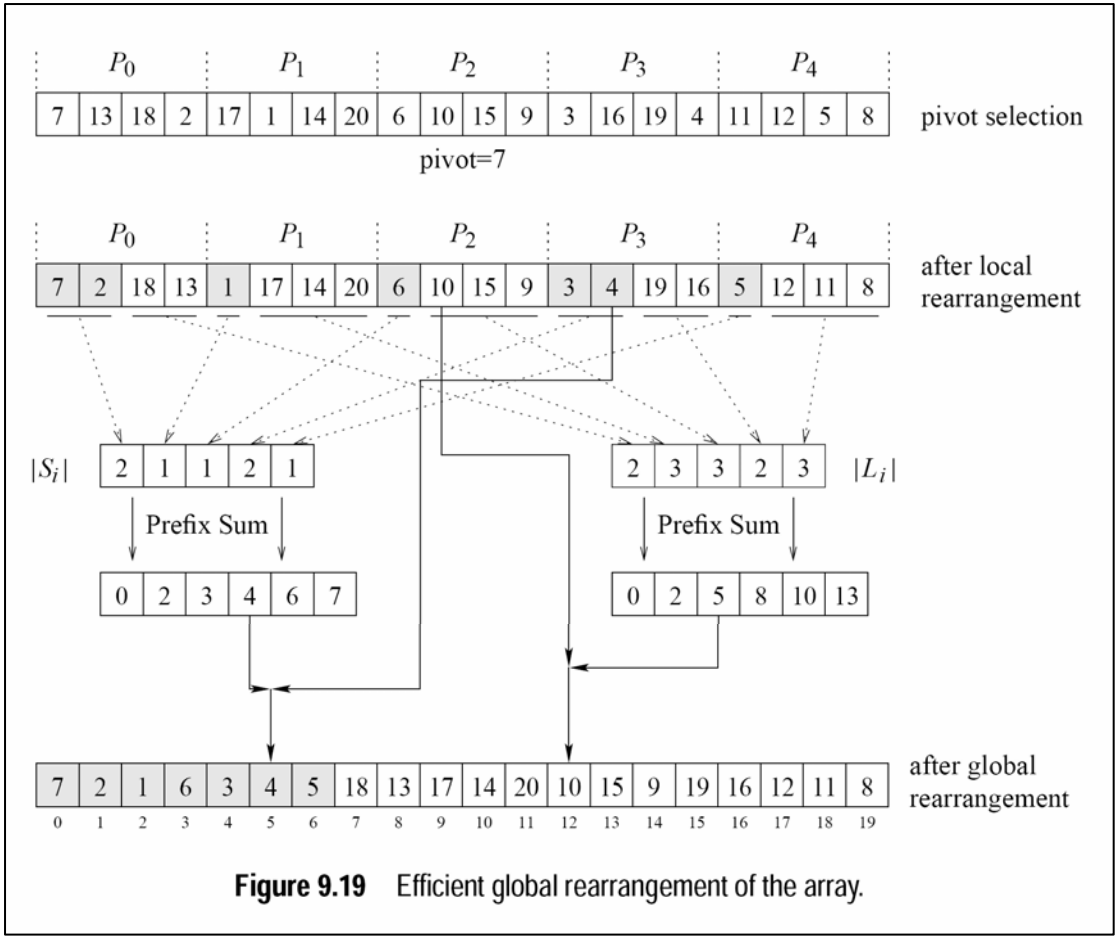



**Figure 9.19** Efficient global rearrangement of the array.

# Practical Quicksort

- Complexity

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

$$\text{overall isoefficiency of } \Theta(p\log^2 p).$$

Complexity for message-passing is similar assuming that the all-to-all personalized communication is not cross-bisection bandwidth limited.

# A word on Pivot Selection

- Selecting pivots that lead to balanced partitions is importance
  - height of the tree
  - effective utilization of processors

# Sample Sort

- Generalization of bucket sort with data-driven sampling
  - n/p elements per-processor.
  - Each processor sorts is local elements.
  - Each processor selects *p-1* equally spaced elements from its own list.
  - The combined *p(p-1)* set of elements are sorted and *p-1* equally spaced elements are selected from that list.
  - Each processor splits its own list according to these splitters into *p* buckets.
  - Each processor sends its *ith* bucket to the *ith* processor.
  - Each processor merges the elements that it receives.
  - Done.

# Sample Sort Illustration

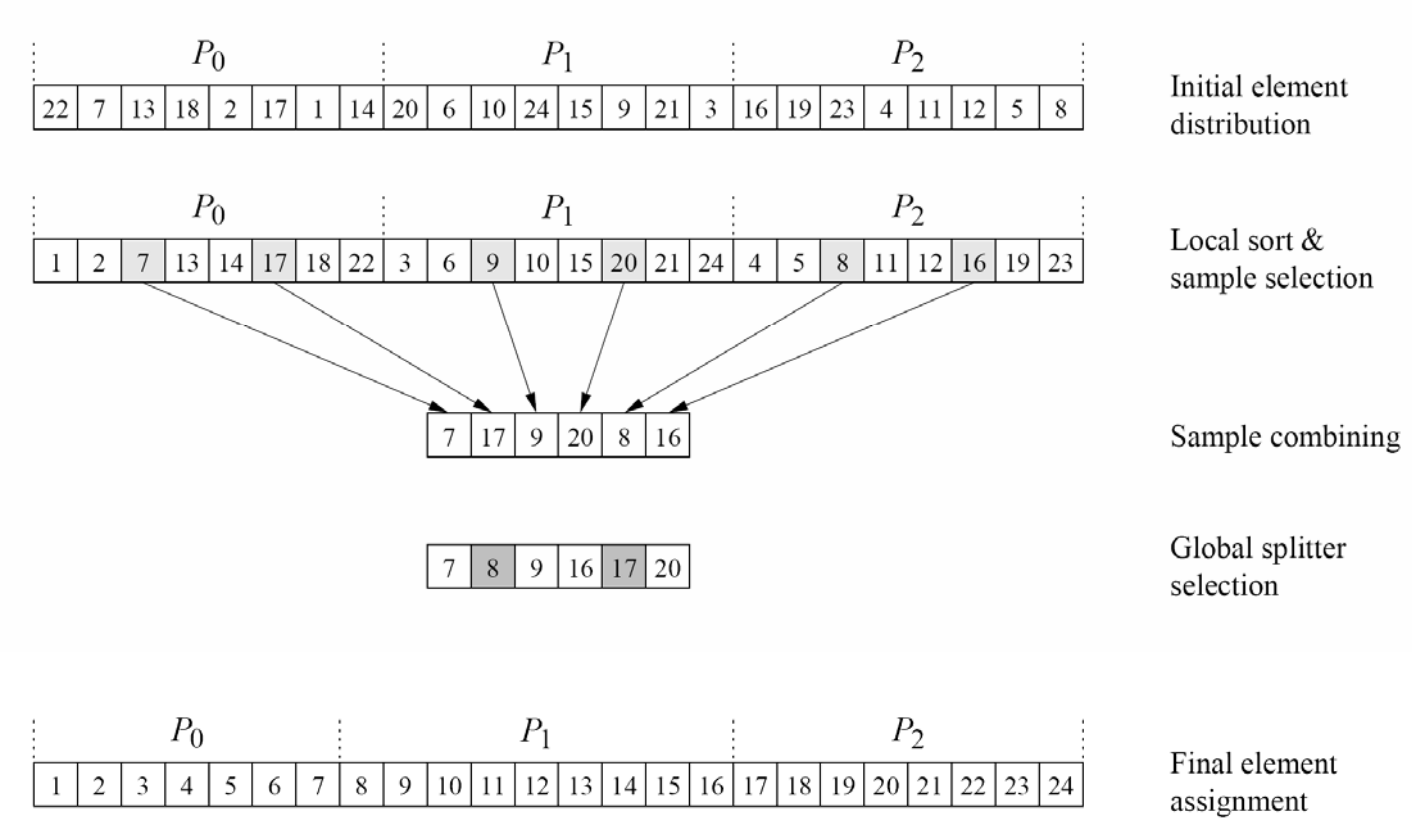



**Figure 9.20** An example of the execution of sample sort on an array with 24 elements on three processes.

# Sample Sort Complexity

$$T_P = \overbrace{\Theta\left(\frac{n}{p}\log\frac{n}{p}\right)}^{\text{local sort}} + \overbrace{\Theta\left(p^2\log p\right)}^{\text{sort sample}} + \overbrace{\Theta\left(p\log\frac{n}{p}\right)}^{\text{block partition}} + \overbrace{\Theta(n/p) + O(p\log p).}^{\text{communication}}$$

Assumes a serial sort

In this case, the isoefficiency function is $\Theta(p^3 \log p)$. If bitonic sort is used to sort the $p(p-1)$ sample elements, then the time for sorting the sample would be $\Theta(p \log p)$, and the isoefficiency will be reduced to $\Theta(p^2 \log p)$ (Problem 9.30).