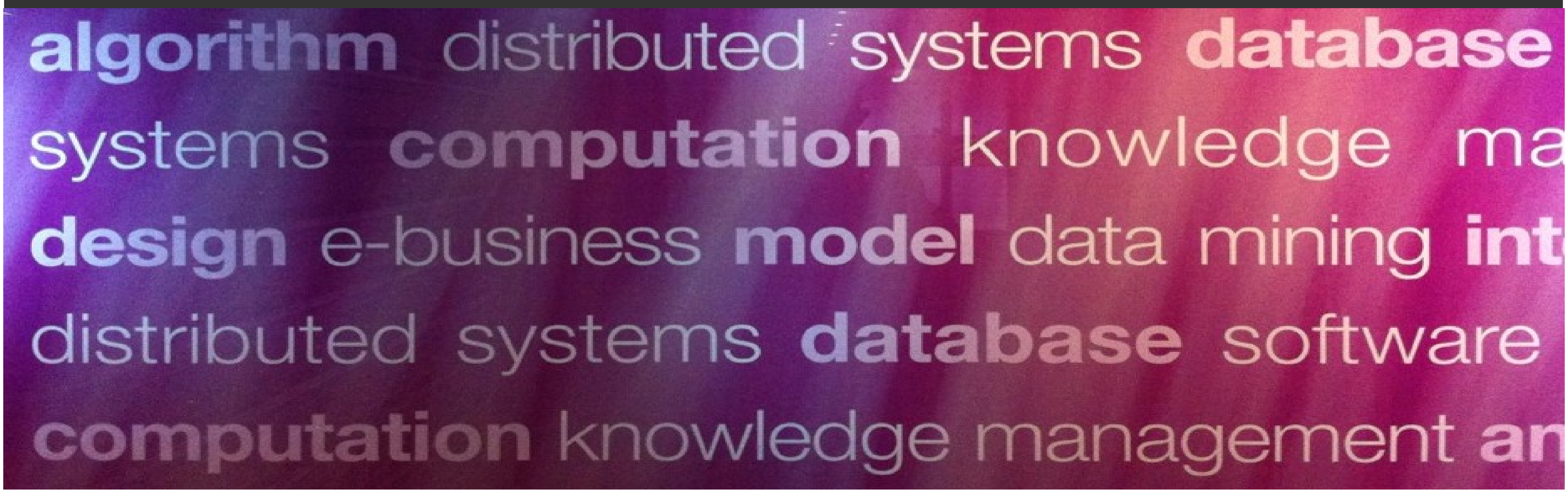
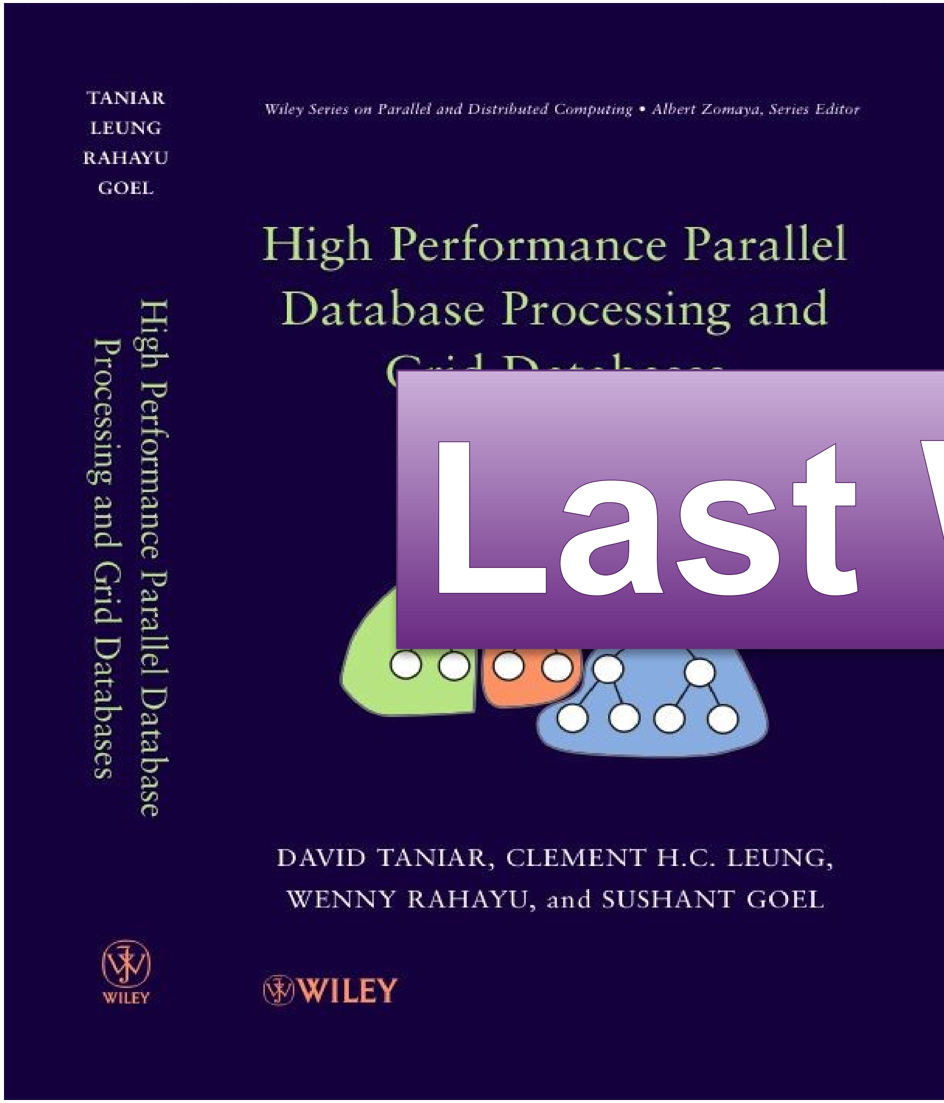
MONASH University

Information Technology

# FIT5202 (Volume IV – Sort and Group By)

Week 4a – Parallel Sort






# Chapter 5 Parallel Join

**Last Week**

5.1 Join Operations
5.2 Serial Join Algorithms
5.3 Parallel Join Algorithms
5.4 Cost Models
5.5 Parallel Join Optimization
5.6 Summary
5.7 Bibliographical Notes
5.8 Exercises

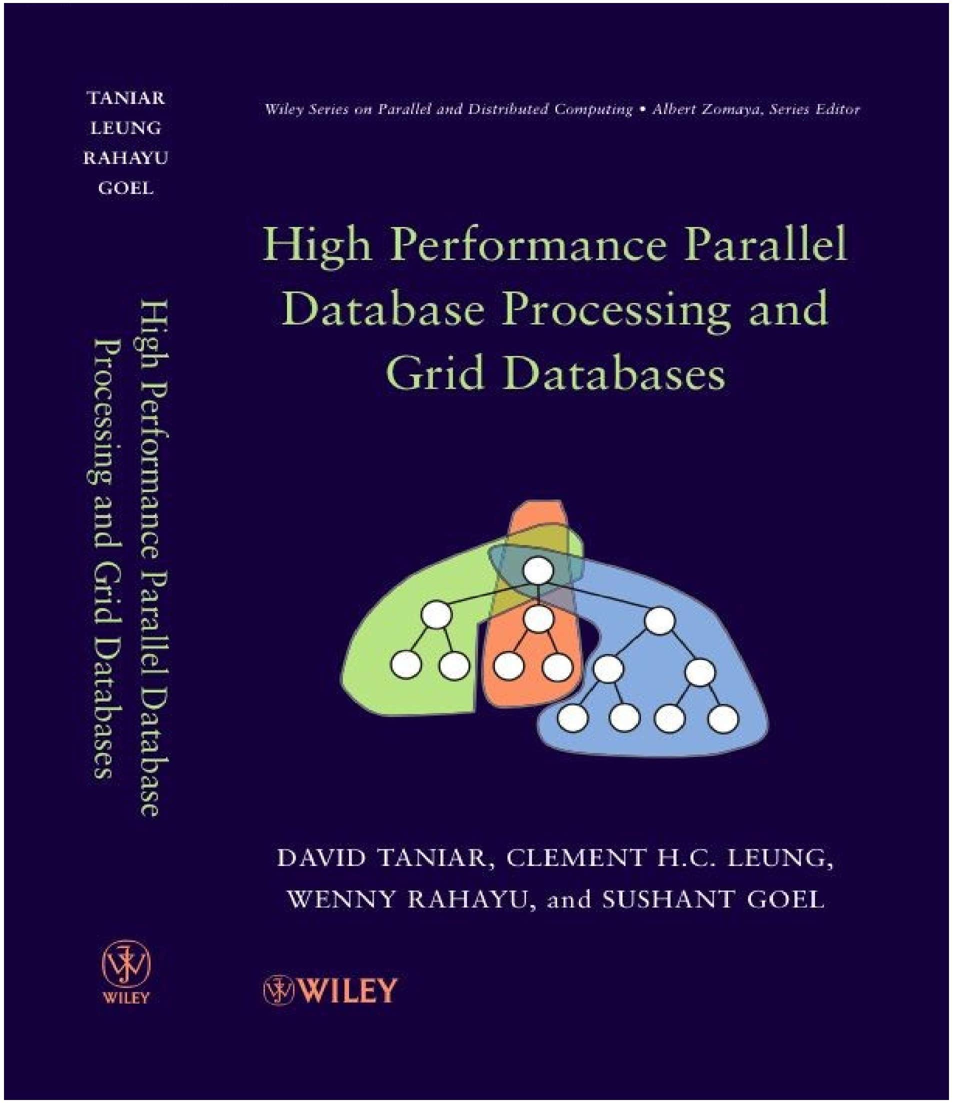
# Revision

- **Exercise 1 (FLUX Quiz)**
  - Parallel Join algorithms for Inner Join consists of two major phases: Data Partitioning, and Local Join
  - A. TRUE
  - B. FALSE

# Revision

- **Exercise 2 (FLUX Quiz)**
  - Parallel Join algorithms for Outer Join queries are:

  - A. ROJA and DOJA
  - B. DER
  - C. OJSO
  - E. only A and B
  - F. A. B and C are correct.




# Chapter 4 Parallel Sort and GroupBy

4.1 Sorting, Duplicate Removal and Aggregate
4.2 Serial External Sorting Method
4.3 Algorithms for Parallel External Sort
4.4 Parallel Algorithms for GroupBy Queries
4.5 Cost Models for Parallel Sort
4.6 Cost Models for Parallel GroupBy
4.7 Summary
4.8 Bibliographical Notes
4.9 Exercises

# 4.1. Sorting, and Serial Sorting

- Serial Sorting – **INTERNAL**
  - The data to be sorted fits entirely into the main memory

- Serial Sorting - **EXTERNAL**
  - The data to be sorted DOES NOT fit entirely into the main memory

# 4.1. Internal Serial Sorting (cont'd)

- **Bubble Sort**
  - Based on swapping
  - It compares the first two elements, and if the first is greater than the second, it swaps them.
  - It continues doing this for each pair of adjacent elements to the end of the data set.
  - It then starts again with the first two elements, repeating until no swaps have occurred on the last pass.
  - Example: 6 5 3 1 8 7 2 4

# 4.1. Internal Serial Sorting (cont'd)

## Bubble Sort

6 5 3 1 8 7 2 4
5 6 3 1 8 7 2 4
5 3 6 1 8 7 2 4
5 3 1 6 8 7 2 4
5 3 1 6 8 7 2 4
5 3 1 6 7 8 2 4
5 3 1 6 7 2 8 4
5 3 1 6 7 2 4 8

5 3 1 6 7 2 4 8
3 5 1 6 7 2 4 8
3 1 5 6 7 2 4 8
3 1 5 6 7 2 4 8
3 1 5 6 7 2 4 8
3 1 5 6 2 7 4 8
3 1 5 6 2 4 7 8

3 1 5 6 2 4 7 8
1 3 5 6 2 4 7 8
1 3 5 6 2 4 7 8
1 3 5 6 2 4 7 8
1 3 5 2 6 4 7 8
1 3 5 2 4 6 7 8

1 3 5 2 4 6 7 8
1 3 5 2 4 6 7 8
1 3 5 2 4 6 7 8
1 3 2 5 4 6 7 8
1 3 2 4 5 6 7 8

1 3 2 4 5 6 7 8
1 3 2 4 5 6 7 8
1 2 3 4 5 6 7 8
1 2 3 4 5 6 7 8

1 2 3 4 5 6 7 8
1 2 3 4 5 6 7 8
1 2 3 4 5 6 7 8
1 2 3 4 5 6 7 8
1 2 3 4 5 6 7 8

**Finished**

# 4.1. Internal Serial Sorting (cont'd)

- **Insertion Sort**
  - Based on inserting a new value
  - It works by taking elements from the list one by one and inserting them in their correct position into a new sorted list. In arrays, the new list and the remaining elements can share the array's space, but insertion is expensive, requiring shifting all following elements over by one.
  - Example: 6 5 3 1 8 7 2 4

| | | |
|---|---|---|
| **6** 5 3 1 8 7 2 4 | Take out 6, and insert it in the previous list | **6** 5 3 1 8 7 2 4 |
| 6 **5** 3 1 8 7 2 4 | Take out 5, and insert it in the previous list | **5 6** 3 1 8 7 2 4 |
| 5 6 **3** 1 8 7 2 4 | Take out 3, and insert it in the previous list | **3 5 6** 1 8 7 2 4 |
| 3 5 6 **1** 8 7 2 4 | Take out 1, and insert it in the previous list | **1 3 5 6** 8 7 2 4 |
| 1 3 5 6 **8** 7 2 4 | Take out 8, and insert it in the previous list | **1 3 5 6 8** 7 2 4 |
| 1 3 5 6 8 **7** 2 4 | Take out 7, and insert it in the previous list | **1 3 5 6 7 8** 2 4 |
| 1 3 5 6 7 8 **2** 4 | Take out 2, and insert it in the previous list | **1 2 3 5 6 7 8** 4 |
| 1 2 3 5 6 7 8 **4** | Take out 4, and insert it in the previous list | **1 2 3 4 5 6 7 8** |

**Finished**

# 4.1. Internal Serial Sorting (cont'd)

- **Quick Sort**
  - Quick Sort is a divide and conquer algorithm which relies on a partition operation: to partition an array an element called a *pivot* is selected.
  - All elements smaller than the pivot are moved before it and all greater elements are moved after it.
  - The lesser and greater sublists are then recursively sorted.
  - The most complex issue in Quick Sort is choosing a good pivot element; consistently poor choices of pivots can result in drastically slower performance
  - Example: 6 5 3 1 8 7 2 4

# 4.1. Internal Serial Sorting (cont'd)

- **Quick Sort**
  - Quick Sort is a divide and conquer algorithm which relies on a partition operation: to partition an array an element called a *pivot* is selected.
  - All elements smaller than the pivot are moved before it and all greater elements are moved after it.
  - The lesser and greater sublists are then recursively sorted.
  - The most complex issue in Quick Sort is choosing a good pivot element; consistently poor choices of pivots can result in drastically slower performance
  - Example: 6 5 3 1 8 7 2 4

**Homework: Work out step-by-step this Quick Sort example**

# 4.2. Serial External Sorting

- Sorting is expressed by the ORDER BY clause in SQL
- Duplicate remove is identified by the keyword DISTINCT in SQL

**Query 4.1:**

```
Select *
From STUDENT
Order By Sdegree;
```

**Query 4.3:**

```
Select Distinct Sdegree
From STUDENT;
```

# 4.2. Serial External Sorting (cont'd)

- External sorting assumes that the data does not fit into main memory
- Most common external sorting is sort-merge
- Break the file up into unsorted subfiles, sort the subfiles, and then merge the subfiles into larger and larger sorted subfiles until the entire file is sorted

```
Algorithm: Serial External Sorting

// Sort phase - Pass 0
1. Read B pages at a time into memory
2. Sort them, and Write out a sub-file
3. Repeat steps 1-2 until all pages have been processed

// Merge phase - Pass i = 1, 2, ...
4. While the number of sub-files at end of previous pass
   is > 1
5. While there are sub-files to be merged from
   previous pass
6.    Choose B-1 sorted sub-files from the previous pass
7.    Read each sub-file into an input buffer page
      at a time
8.    Merge these sub-files into one bigger sub-file
9.    Write to the output buffer one page at a time
```

**Figure 4.1** External sorting algorithm based on sort-merge

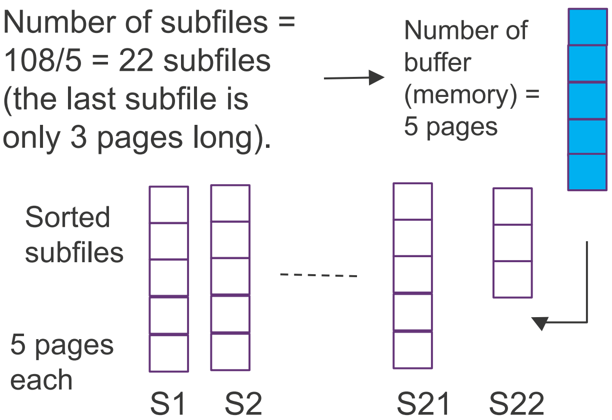
# 4.2. Serial External Sorting (cont'd)

- **Example**
  - File size to be sorted = 108 pages, number of buffer (or memory size) = 5 pages
  - Number of subfiles = 108/5 = 22 subfiles (the last subfile is only 3 pages long).
  - **Pass 0** (sorting phase): For each subfile, **read from disk**, **sort in main-memory**, and **write to disk** (Note: sorting the data in main-memory can use any fast in-memory sorting method, like Quick Sort)
  - Merging phase: **We use *B*-1 buffers (4 buffers) for input and 1 buffer for output**
  - **Pass 1**: Read 4 sorted subfiles and perform 4-way merging (apply a need *k*-way algorithm). Repeat the 4-way merging until all subfiles are processed. Result = 6 subfiles with 20 pages each (except the last one which has 8 pages)
  - **Pass 2**: Repeat 4-way merging of the 6 subfiles like pass 1 above. Result = 2 subfiles
  - **Pass 3**: Merge the last 2 subfiles
  - Summary: 108 pages and 5 buffer pages require 4 passes

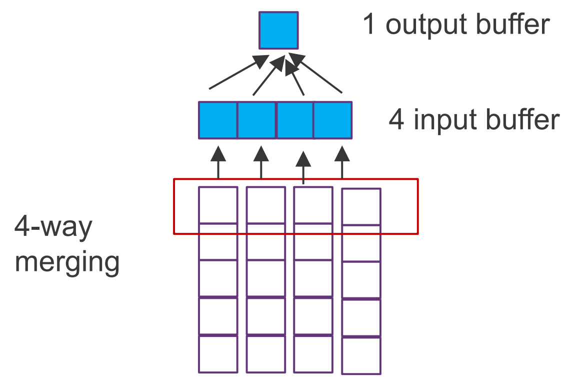
## **Pass 0** (sorting phase):

Number of subfiles = 108/5 = 22 subfiles (the last subfile is only 3 pages long).

Number of buffer (memory) = 5 pages




## **Pass 1** (Merging phase):




S1-S4
S5-S8
S9-S12
S13-S16
S17-S20

4-way merging → SS1, SS2, SS3, SS4, SS5 — 4 X 5 pages = 20 pages each

S21 & S22 — 2-way merging → SS6 — 5 + 3 pages = 8 pages

## **Pass 2** (Merging phase):

SS1-SS4 — 4-way merging → SSS1 — 4 X 20 pages = 80 pages

SS5 & SS6 — 2-way merging → SSS2 — 20 + 8 pages = 28 pages

## **Pass 3** (Merging phase):

SSS1 & SSS2 — 2-way merging → SSSS1 (sorted array)

# 4.2. Serial External Sorting (cont'd)

- **Exercise 3 (FLUX Quiz)**
  - There are 150 data pages to be sorted. The machine that we have has a limited memory, and can only take 8 pages at a time. How many passes will it take to sort the 150 data pages?
  - A. 2
  - B. 3
  - C. 4
  - D. 5

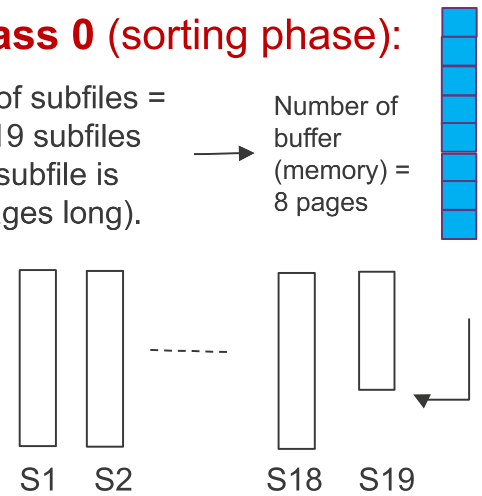
File size to be sorted = 150 pages, number of buffer (or memory size) = 8 pages

## **Pass 0** (sorting phase):

Number of subfiles = 150/8 = 19 subfiles (the last subfile is only 6 pages long).

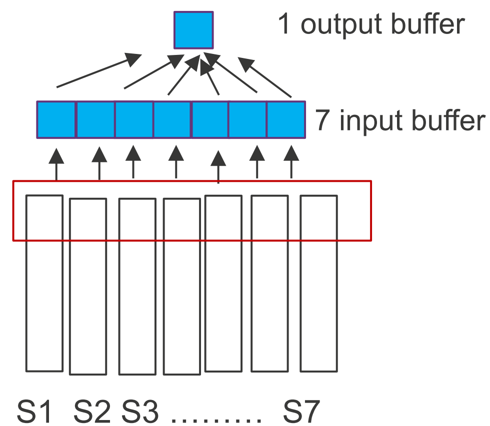
Sorted subfiles

8 pages each




## **Pass 1** (Merging phase):

7-way merging




| | | | |
|---|---|---|---|
| S1-S7<br>S8-S14 | 7-way merging → | SS1<br>SS2 | 7 X 8 pages = 56 pages each |
| S15-S18 & S19 | 5-way merging → | SS3 | 4 X 8 + 6 pages = 38 pages |

## **Pass 2** (Merging phase):

SS1 & SS2 & SS3 $\xrightarrow{\text{3-way merging}}$ SSS1 (sorted array)

# 4.2. Serial External Sorting (cont'd)

- **Example**
  - Buffer size plays an important role in external sort

**Table 4.1** Number of passes in serial external sorting as number of buffer increases

| R | B = 3 | B = 5 | B = 9 | B = 17 | B = 129 | B = 257 |
|---|---|---|---|---|---|---|
| **100** | 7 | 4 | 3 | 2 | 1 | 1 |
| **1,000** | 10 | 5 | 4 | 3 | 2 | 2 |
| **10,000** | 13 | 7 | 5 | 4 | 2 | 2 |
| **100,000** | 17 | 9 | 6 | 5 | 3 | 3 |
| **1 million** | 20 | 10 | 7 | 5 | 3 | 3 |
| **10 million** | 23 | 12 | 8 | 6 | 4 | 3 |
| **100 million** | 26 | 14 | 9 | 7 | 4 | 4 |
| **1 billion** | 30 | 15 | 10 | 8 | 5 | 4 |

# 4.3. Parallel External Sort

5 different Algorithms

- Parallel Merge-All Sort
- Parallel Binary-Merge Sort

Without data redistribution

- Parallel Redistribution Binary-Merge Sort
- Parallel Redistribution Merge-All Sort
- Parallel Partitioned Sort

With data redistribution

# 4.3. Parallel External Sort (cont'd)

- **Parallel Merge-All Sort**
  - A traditional approach
  - Two phases: local sort and final merge
  - Load balanced in local sort
  - Problems with merging:
    - Heavy load on one processor
    - Network contention

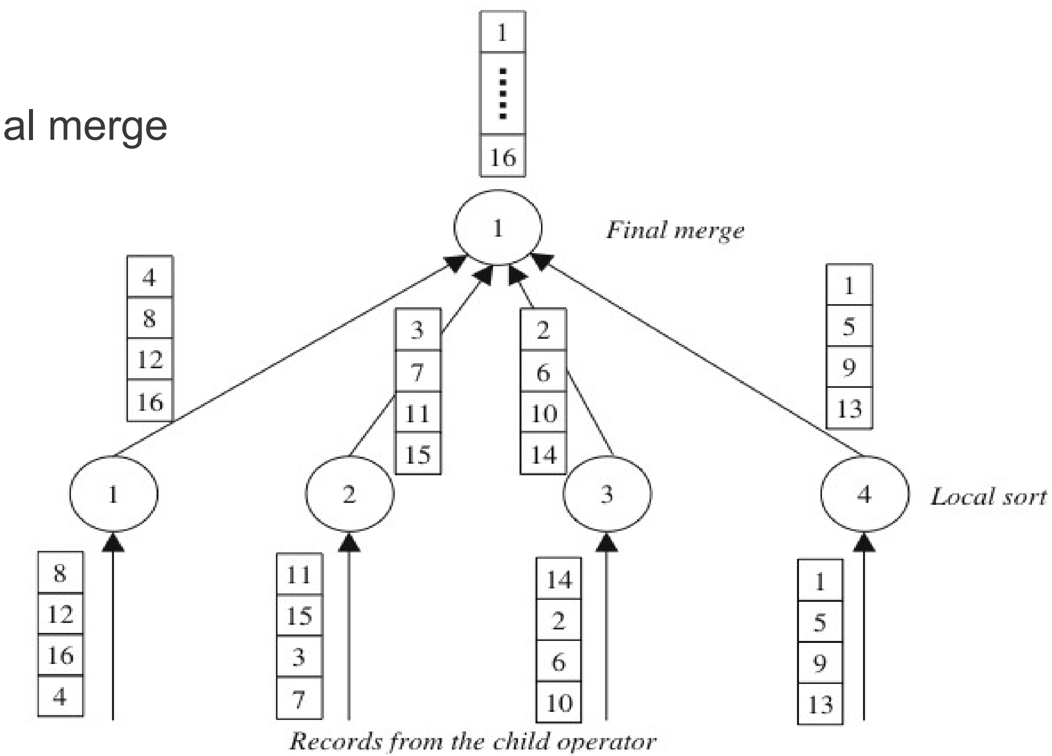



**Figure 4.3** Parallel merge-all sort

# 4.3. Parallel External Sort (cont'd)

- **Parallel Binary-Merge Sort**
  - Local sort similar to traditional method
  - Merging in pairs only
  - Merging work is now spread to pipeline of processors, but merging is still heavy

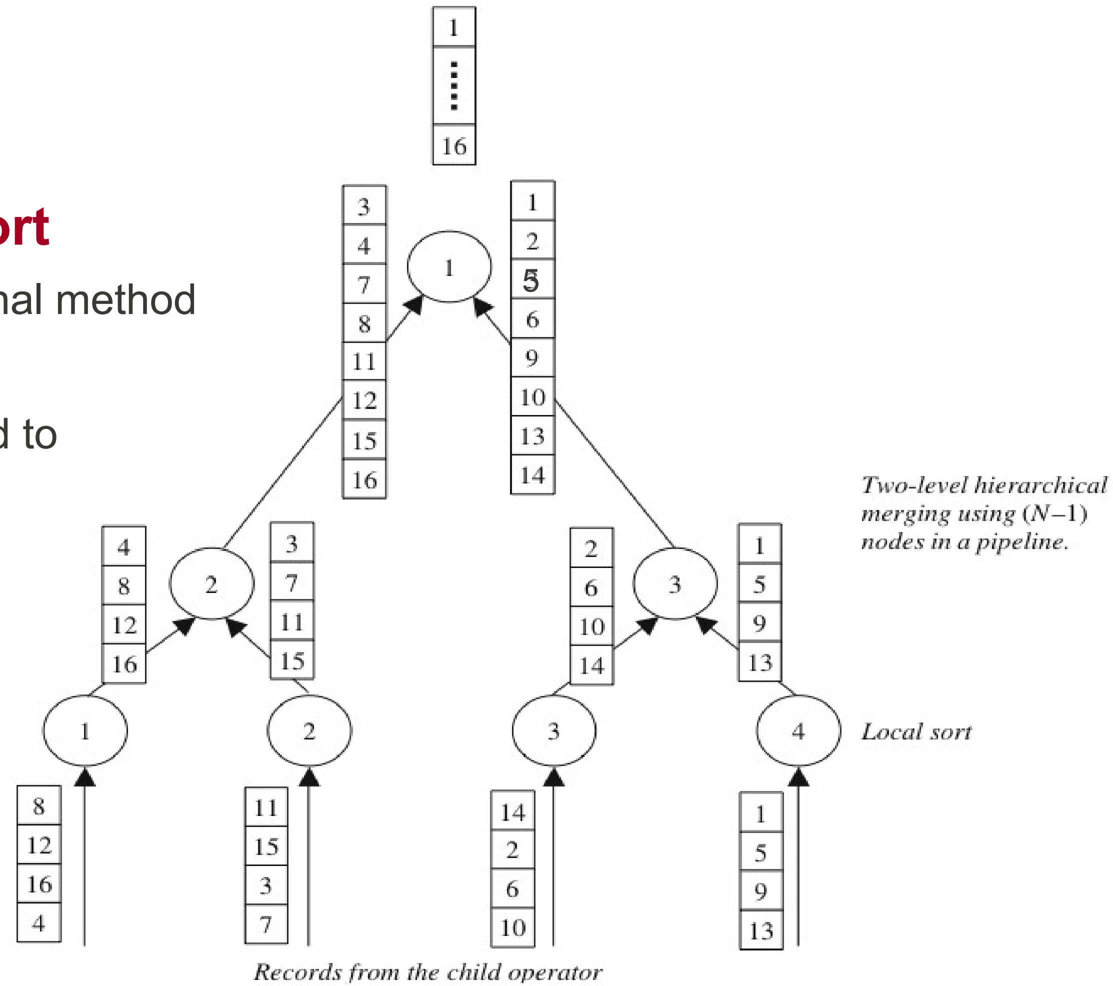
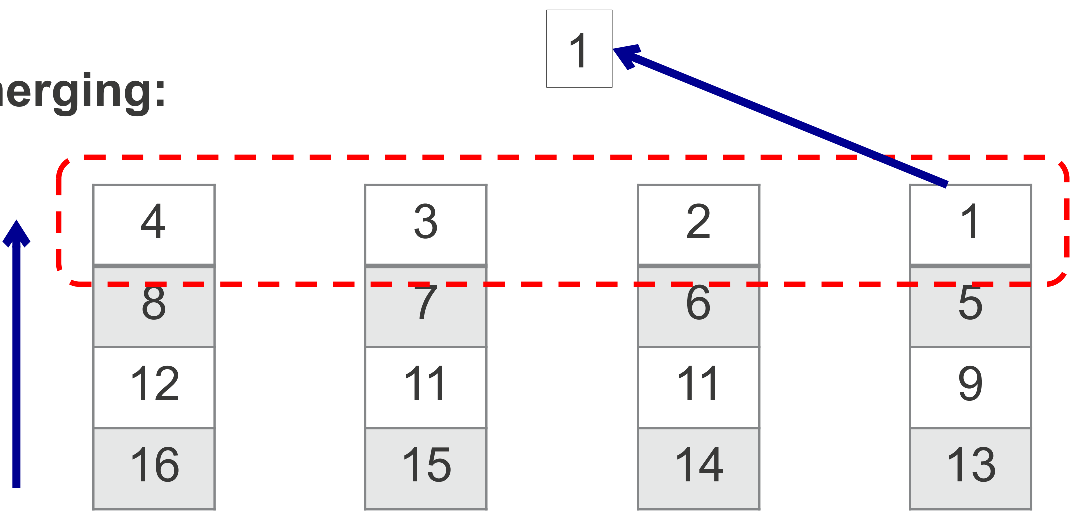



**Figure 4.4** Parallel binary-merge sort

# 4.3. Parallel External Sort (cont'd)

- **Parallel Binary-Merge Sort**
  - Binary merging vs. *k*-way merging
  - In *k*-way merging, the **searching for the smallest value among *k* partitions** is done at the same time
  - In binary merging, it is pairwise, but can be time consuming if the list is long
  - System requirements: ***k*-way merging requires *k* files open simultaneously**, but the pipeline process in binary merging requires extra overheads

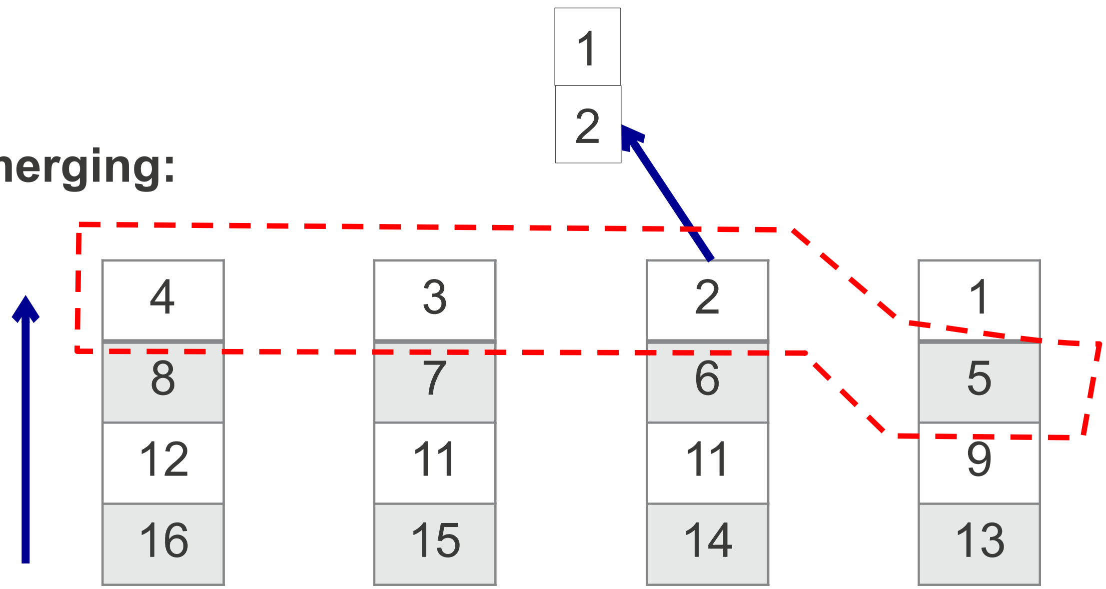
***k*-way merging:**




Searching for the smallest value among the *k* values

# 4.3. Parallel External Sort (cont'd)

***k*-way merging:**




Searching for the smallest value among the *k* values

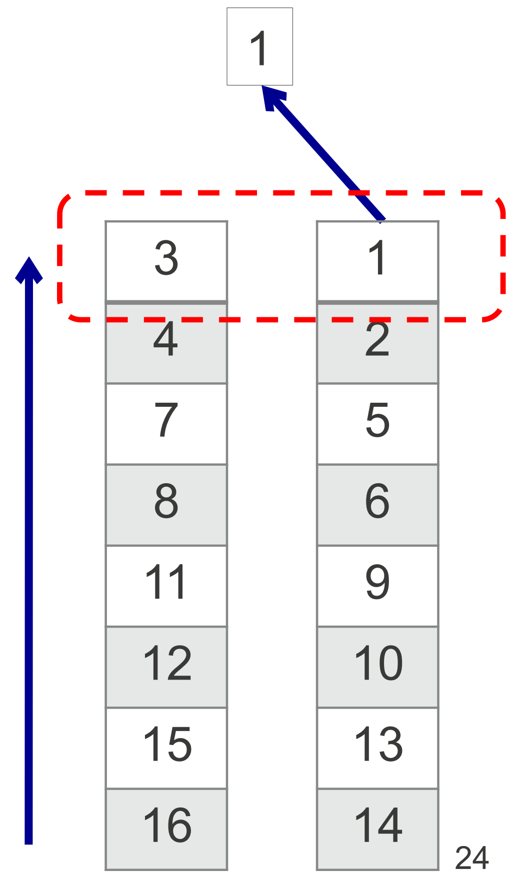
and so on…

# 4.3. Parallel External Sort (cont’d)

- **Parallel Binary-Merge Sort (Binary Merging step)**
  - Binary merging vs. *k*-way merging
  - In **binary merging**, it is pairwise, but can be time consuming if the list is long
  - System requirements: the pipeline process in binary merging requires extra overheads

**Binary merging:**

Compare two values only, but lists are longer



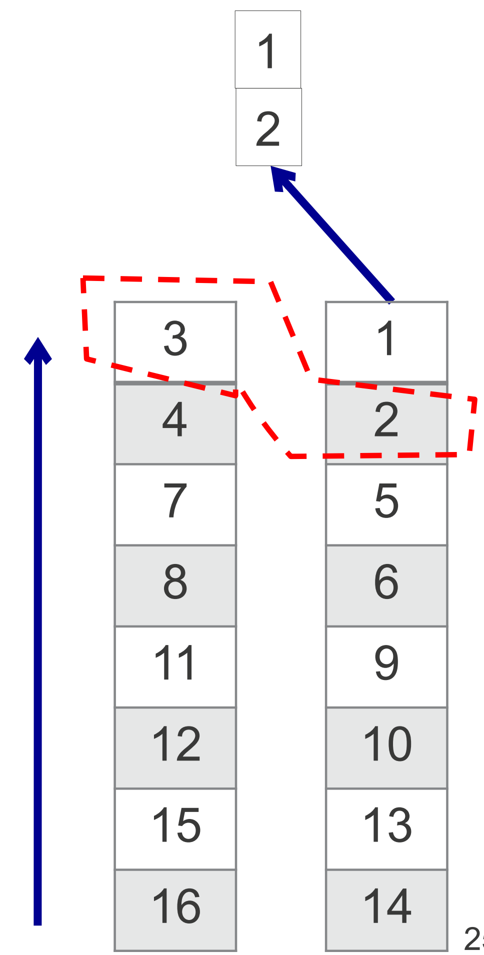
# 4.3. Parallel External Sort (cont'd)

- **Parallel Binary-Merge Sort (Binary Merging step)**
  - Binary merging vs. *k*-way merging
  - In **binary merging**, it is pairwise, but can be time consuming if the list is long
  - System requirements: the pipeline process in binary merging requires extra overheads

**Binary merging:**

Compare two values only, but lists are longer

And so on…



## Parallel Redistribution Binary-Merge Sort

- Parallelism at all levels in the pipeline hierarchy
- Step 1: local sort
- Step 2: redistribute the results of local sort
- Step 3: merge using the same pool of processors

- Benefit: merging becomes lighter than without redistribution
- Problem: height of the tree

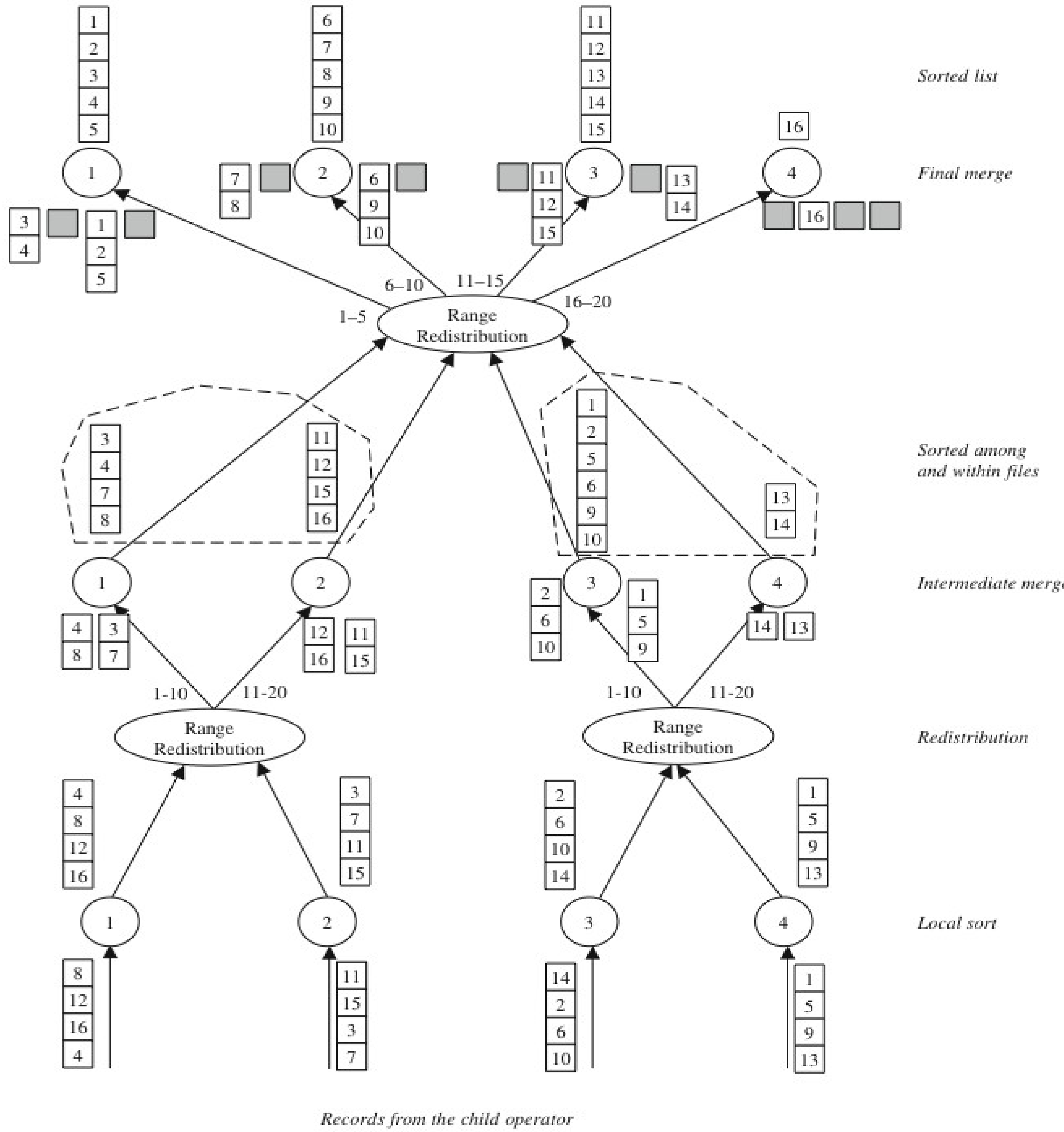



**Figure 4.6** Parallel redistribution binary-merge sort

## Parallel Redistribution Merge-All Sort

- Reduce the height of the tree, and still maintain parallelism
- Like parallel merge-all sort, but with redistribution
- The advantage is true parallelism in merging
- Skew problem in the merging

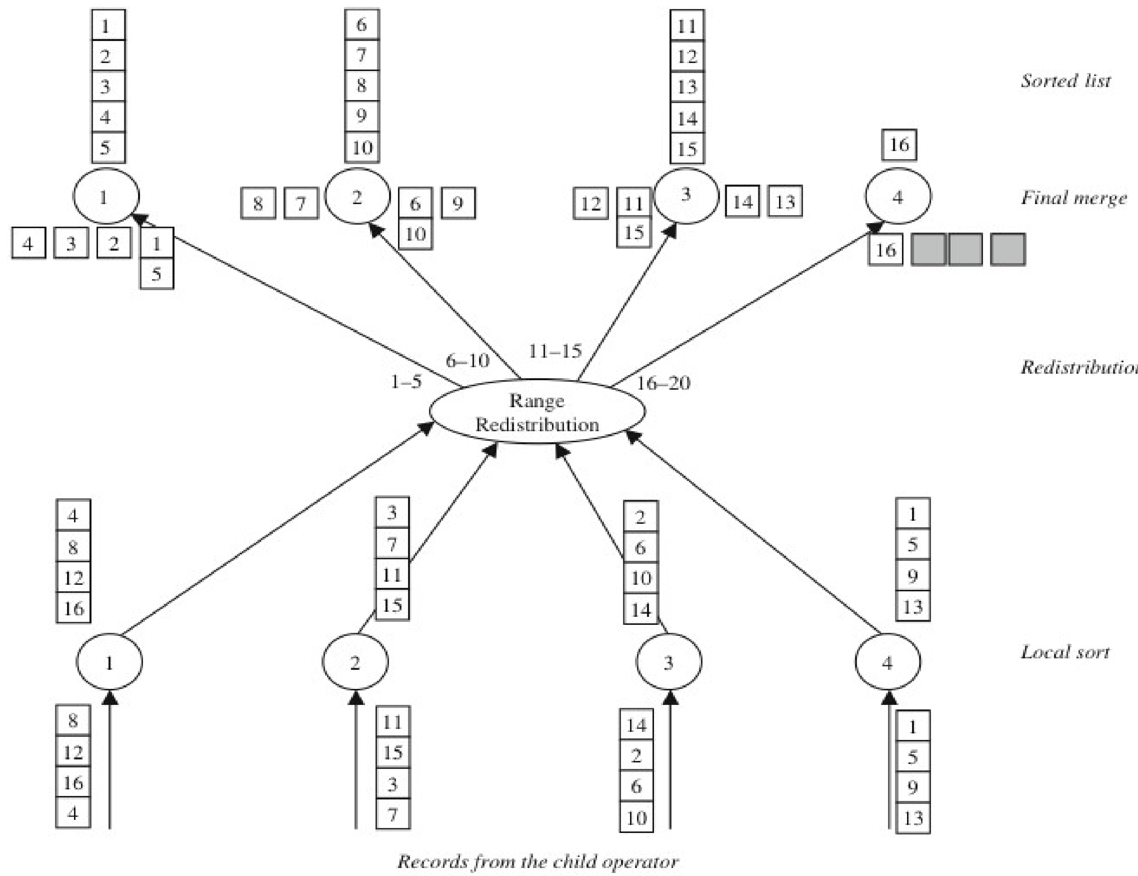



**Figure 4.7** Parallel redistribution merge-all sort

## Parallel Partitioned Sort

- Two stages: Partitioning stage and Independent local work
- Partitioning (or range redistribution) may raise load skew
- Local sort is done after the partitioning, not before
- No merging is necessary
- Main problem: **Skew** produced by the partitioning

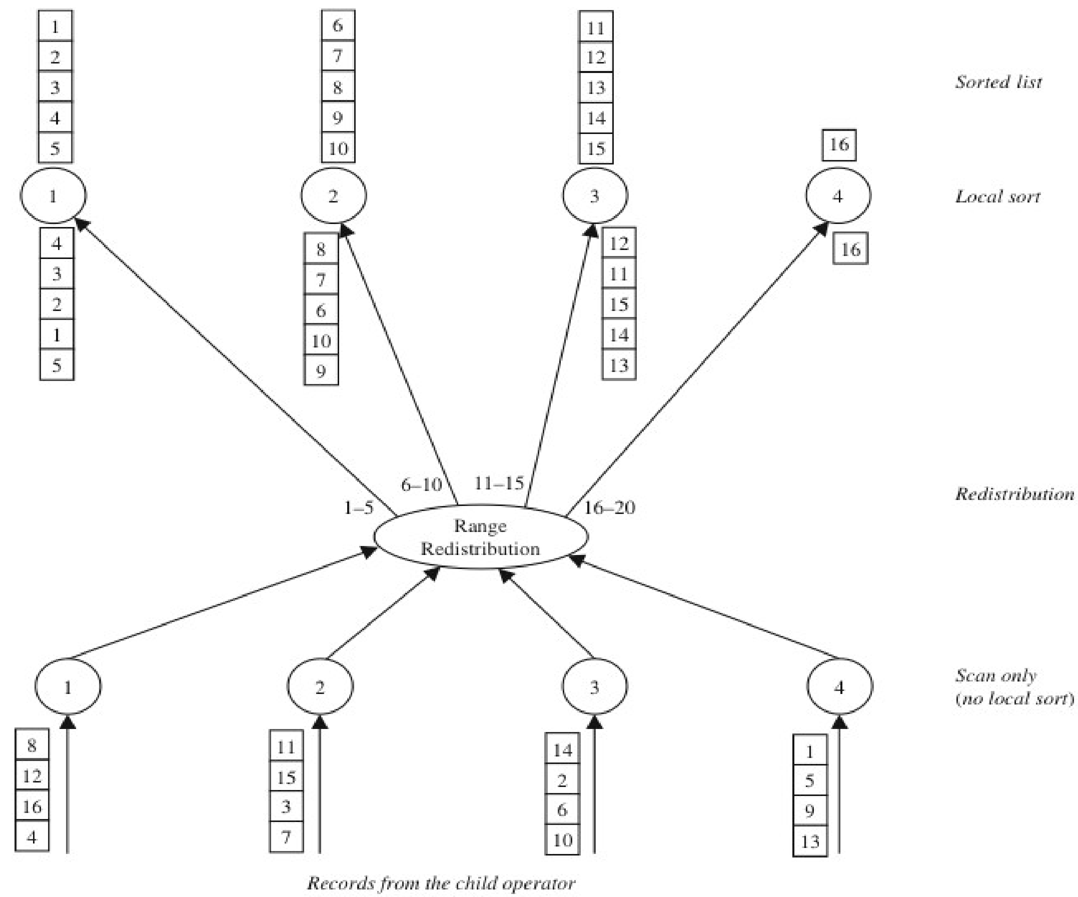



**Figure 4.8** Parallel partitioned sort

# 4.2. Serial External Sorting (cont'd)

- **Exercise 4 (Home Work)**
  - Given a data set $D$ = {55; 30; 68; 39; 1; 4; 49; 90; 34; 76; 82; 56; 31; 25; 78; 56; 38; 32; 88; 9; 44; 98; 11; 70; 66; 89; 99; 22; 23; 26} and four processors, show step by step how the **Parallel Partitioned Sort** works.

# 4.2. Serial External Sorting (cont'd)

- **Exercise 5 (Difficult)**
  - Given the same dataset as in the previous question, and 4 processors, show how **Load Balancing** is achieved in the **Parallel Partitioned Sort**.