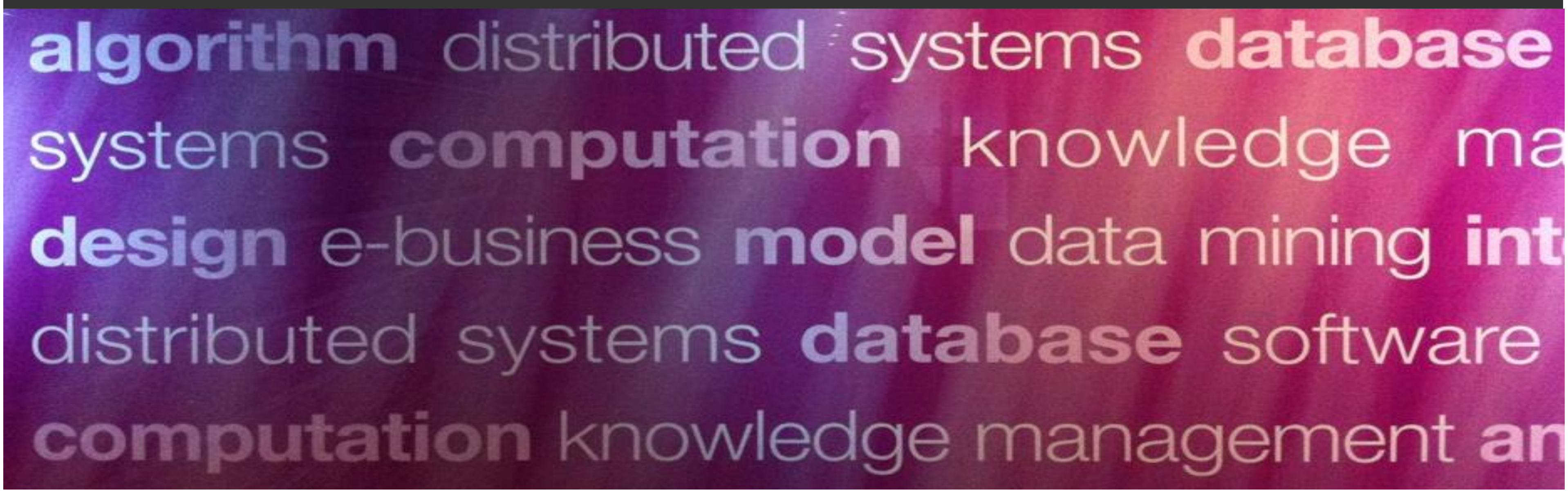
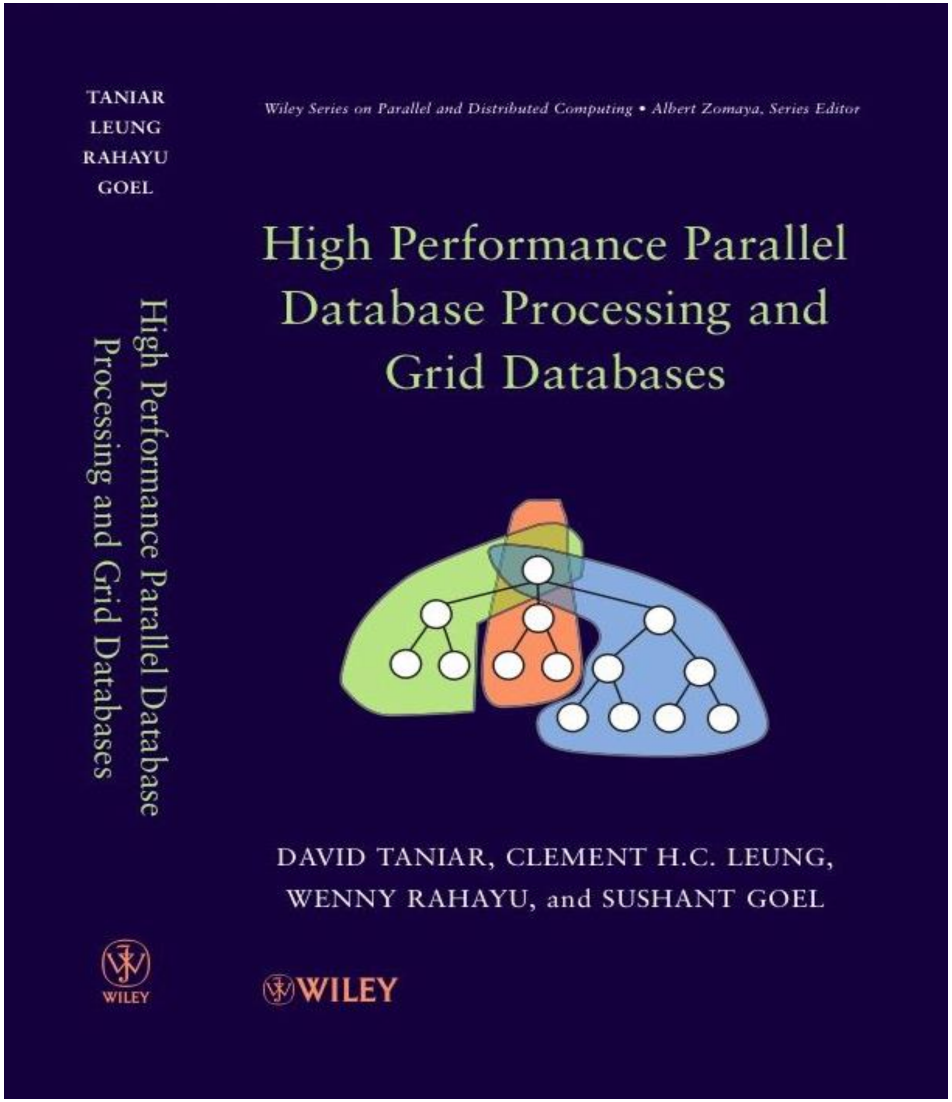
MONASH University

Information Technology

FIT5202 (Volume IV – Sort and Group By)

Week 4a – Parallel Sort





Chapter 4 Parallel Sort and GroupBy

4.1. Sorting, and Serial Sorting

- Serial Sorting – **INTERNAL**
  - The data to be sorted fits entirely into the main memory
  - Three types of algorithm
    - Bubble Sort
    - Insertion Sort
    - Quick Sort

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

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)

6 5 3 1 8 7 2 4

Bubble Sort

6 5 3 1 8 7 2 4	5 3 1 6 7 2 4 8	3 1 5 6 2 4 7 8	1 3 5 2 4 6 7 8	1 3 2 4 5 6 7 8
5 6 3 1 8 7 2 4	3 5 1 6 7 2 4 8	1 3 5 6 2 4 7 8	1 3 5 2 4 6 7 8	1 3 2 4 5 6 7 8
5 3 6 1 8 7 2 4	3 1 5 6 7 2 4 8	1 3 5 6 2 4 7 8	1 3 5 2 4 6 7 8	1 2 3 4 5 6 7 8
5 3 1 6 8 7 2 4	3 1 5 6 7 2 4 8	1 3 5 6 2 4 7 8	1 3 2 5 4 6 7 8	1 2 3 4 5 6 7 8
5 3 1 6 8 7 2 4	3 1 5 6 7 2 4 8	1 3 5 2 6 4 7 8	1 3 2 4 5 6 7 8	
5 3 1 6 7 8 2 4	3 1 5 6 2 7 4 8	1 3 5 2 4 6 7 8		
5 3 1 6 7 2 8 4	3 1 5 6 2 4 7 8			
5 3 1 6 7 2 4 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** 6 5 3 1 8 7 2 4
  - 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 (previous) 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 new 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 new 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 new 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 new 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 new 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 new 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 new 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 new 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

Divide: Pick a pivot x in array A

- Partition the array into sub-arrays

Conquer: Recursively sort sub-arrays L & G

6 5 3 1 8 7 2 4

<x	x	>x
L	pivot	G

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)

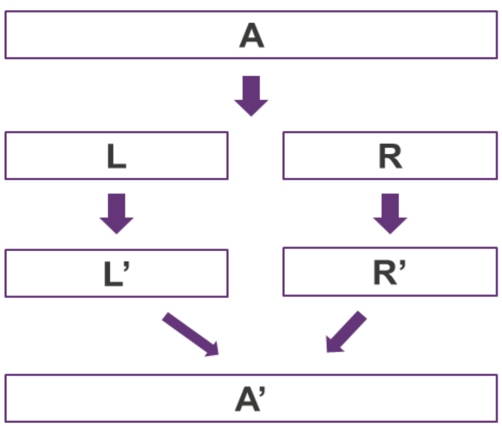
- External sorting assumes that the data does not fit into main memory
- Most common external sorting is sort-merge
- Break the file up into unsorted subfiles, sort the subfiles, and then merge the subfiles into larger and larger sorted subfiles until the entire file is sorted

```
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



sorted

merging

Divide & Conquer strategy

4.2. Serial External Sorting (cont'd)

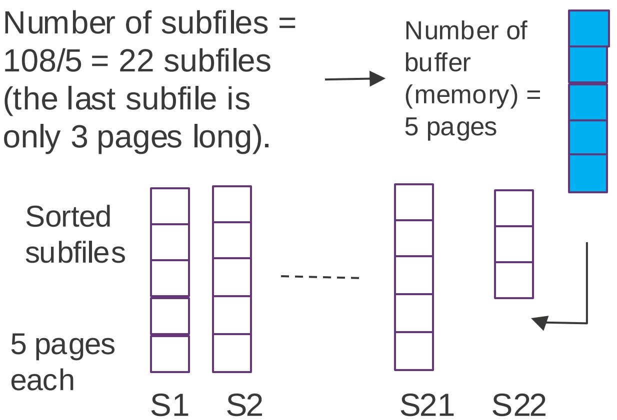
- **Example**
  - File size to be sorted = 108 pages, number of buffer (or memory size) = 5 pages
  - Number of subfiles = 108/5 = 22 subfiles (the last subfile is only 3 pages long).
  - **Pass 0** (sorting phase): For each subfile, **read from disk**, **sort in main-memory**, and **write to disk** (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

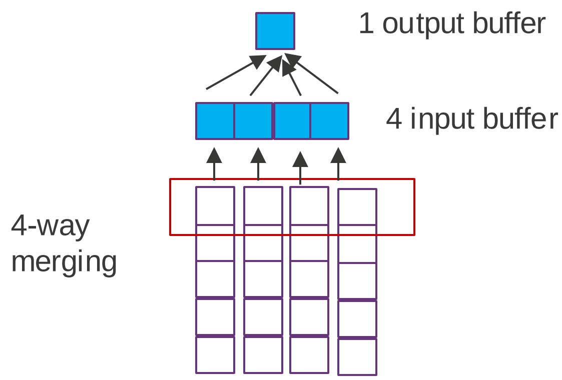
  - Summary: 108 pages and 5 buffer pages require 4 passes

Pass 0 (sorting phase):

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



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)

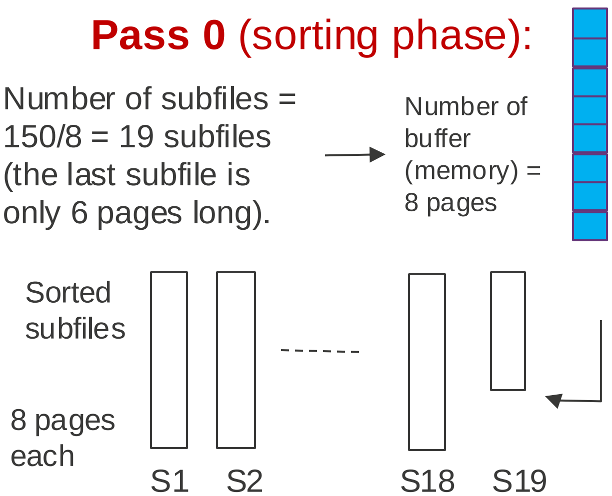
4.2. Serial External Sorting (cont'd)

- **Exercise 3**
  - 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?
- Solution:
- File size to be sorted = 150 pages, number of buffer (or memory size) = 8 pages
- Number of subfiles = 150/8 = 19 subfiles (Last subfile has only 6 pages)
- **Pass 0 (sorting phase):** For each subfile, read from disk, sort in main-memory, and write to disk
- Merging phase: We use **7 buffers for input** and **1 buffer for output**
- **Pass 1**: Read 7 sorted subfiles and perform 7-way merging. Repeat the 7-way merging until all subfiles are processed. Result = 3 subfiles
- **Pass 2:** Merge the 3 subfiles
- Summary: 150 pages and 8 buffer pages require 3 passes

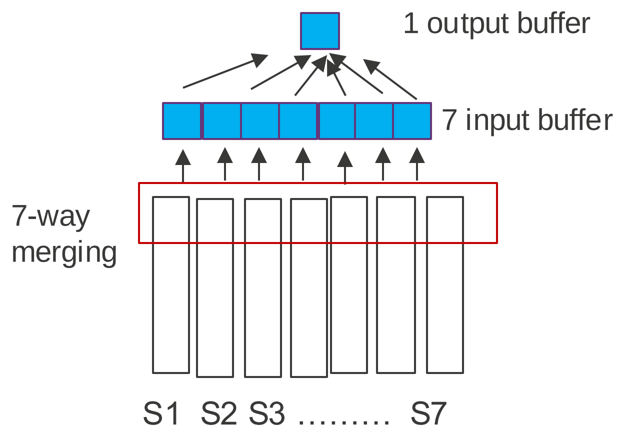
File size to be sorted = 150 pages, number of buffer (or memory size) = 8 pages

Pass 0 (sorting phase):

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



Pass 1 (Merging phase):



S1-S7, S8-S14 $\xrightarrow{\text{7-way merging}}$ SS1, SS2 — 7 X 8 pages = 56 pages each

S15-S18 & S19 $\xrightarrow{\text{5-way merging}}$ SS3 — 4 X 8 + 6 pages = 38 pages

Pass 2 (Merging phase):

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

4.3. Parallel External Sort

Parallel Sort:

Step 1: **Data partitioning**: Divide unsorted array into partitions/sub-arrays

Step 2: **Local sort**: Each sub-array sorted by a processor in parallel, then **combine/merge** results

5 different Algorithms

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

Without data redistribution before merging

- Parallel Redistribution Binary-Merge Sort
- Parallel Redistribution Merge-All Sort
- Parallel Partitioned Sort (without local sort)

With data redistribution before merging

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
    - (Transfer of data from many P's to one P)

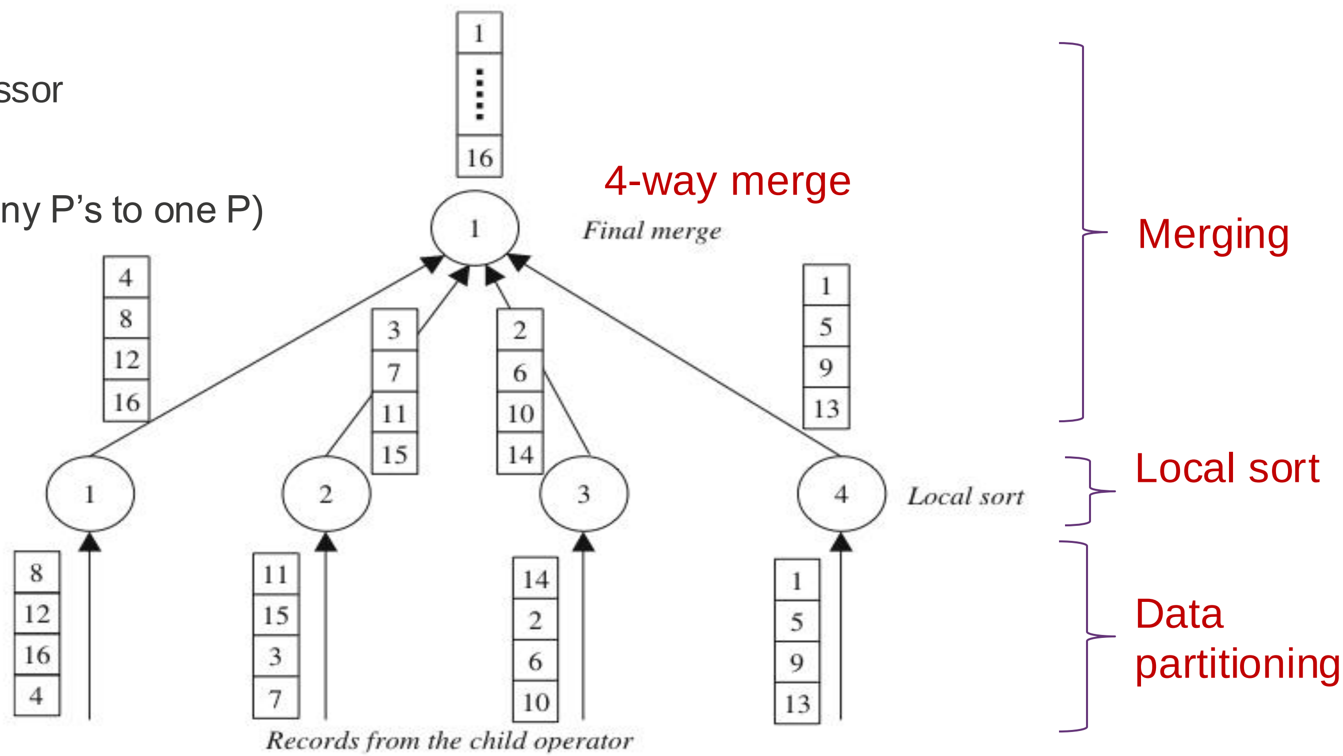


Figure 4.3 Parallel merge-all sort

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

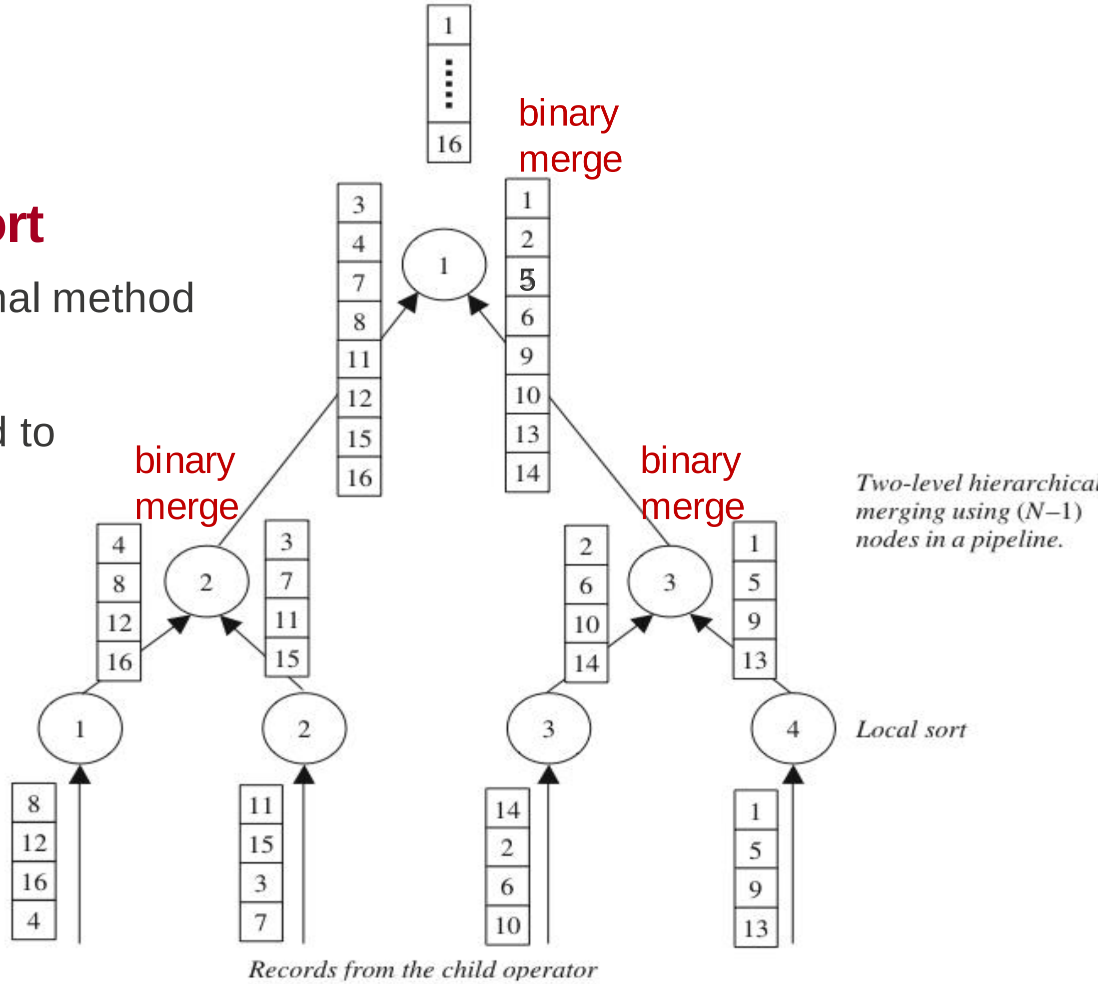
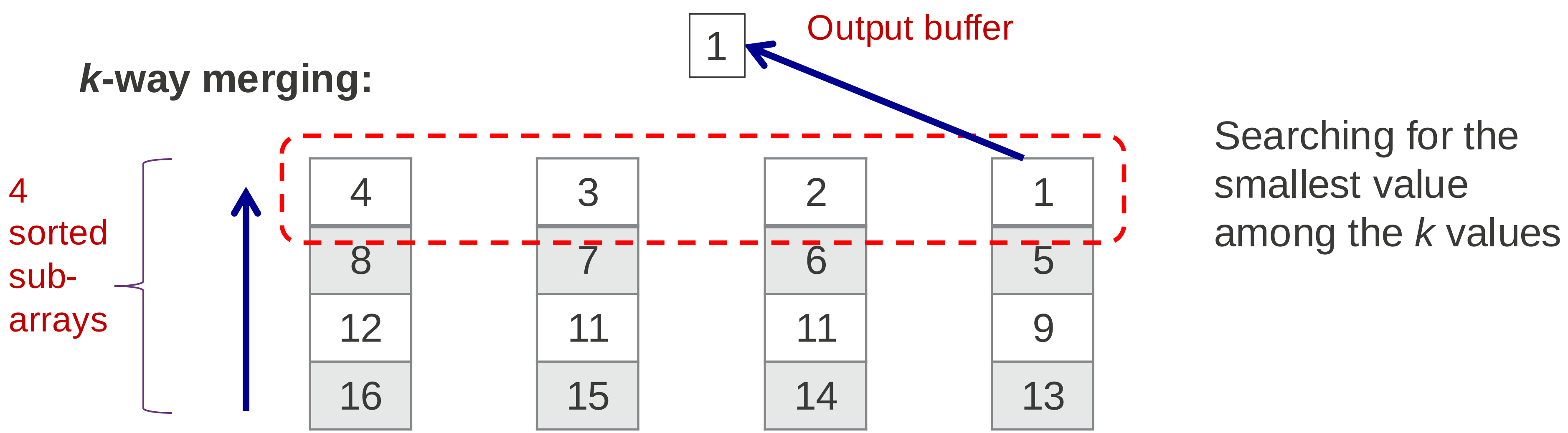


Figure 4.4 Parallel binary-merge sort

4.3. Parallel External Sort (cont'd)

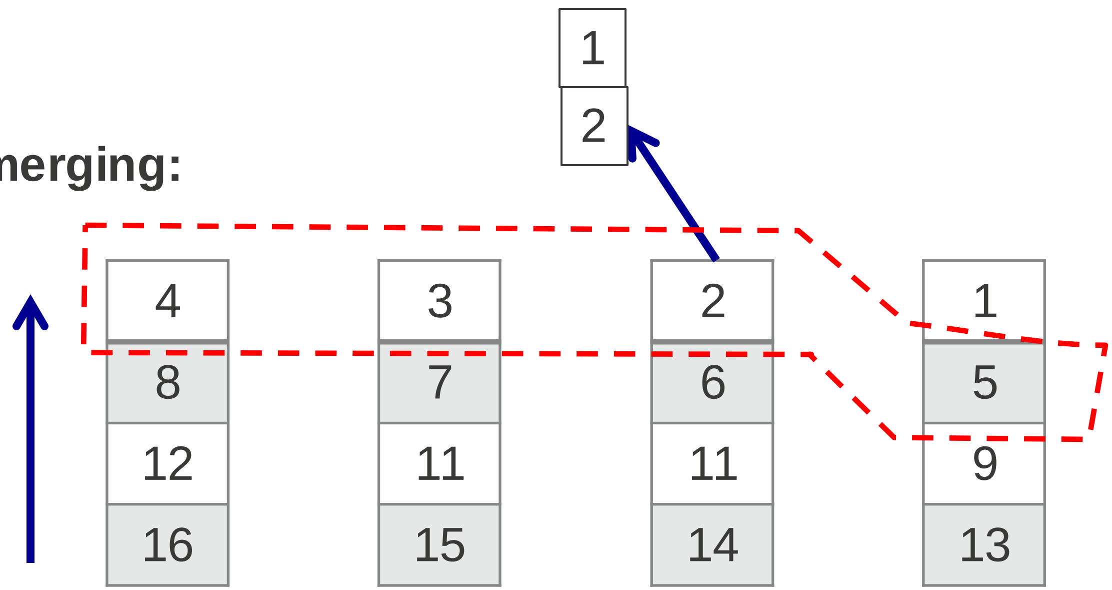
- **Merging**
  - Binary merging vs. *k*-way merging
  - In *k*-way merging, the **searching for the smallest value among *k* partitions** is done at the same time
  - In binary merging, it is pairwise, but can be time consuming if the list is long
  - System requirements: ***k*-way merging requires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

***k*-way merging:**



4.3. Parallel External Sort (cont'd)

***k*-way merging:**



Searching for the smallest value among the *k* values

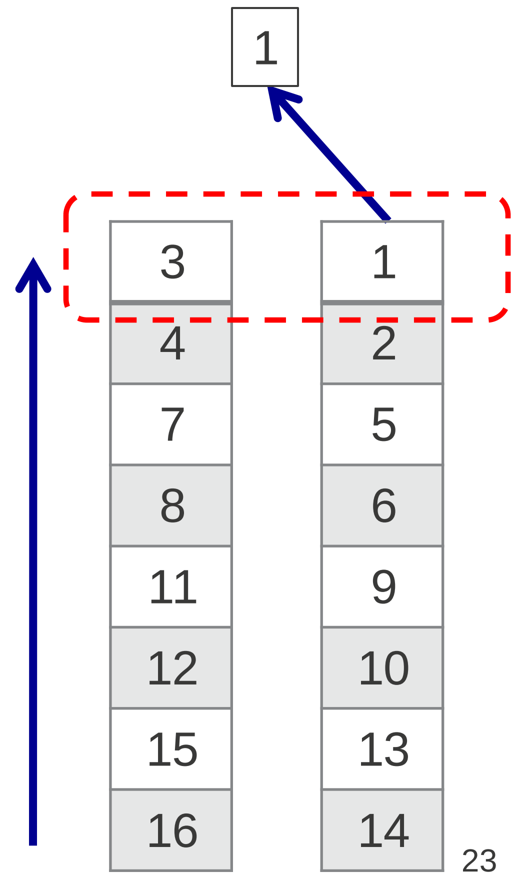
and so on…

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



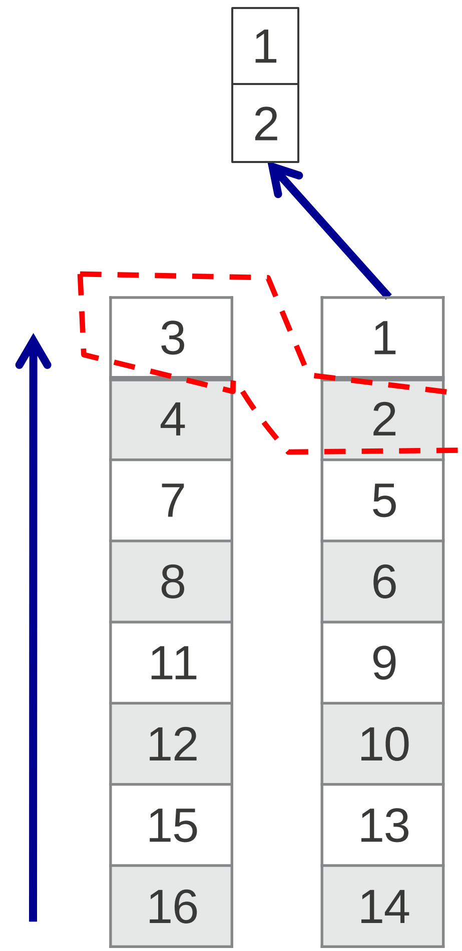
4.3. Parallel External Sort (cont'd)

- **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
  - Merging shorter arrays
  - All processors are involved in merging at each level
- Problem: height of the tree, skewness

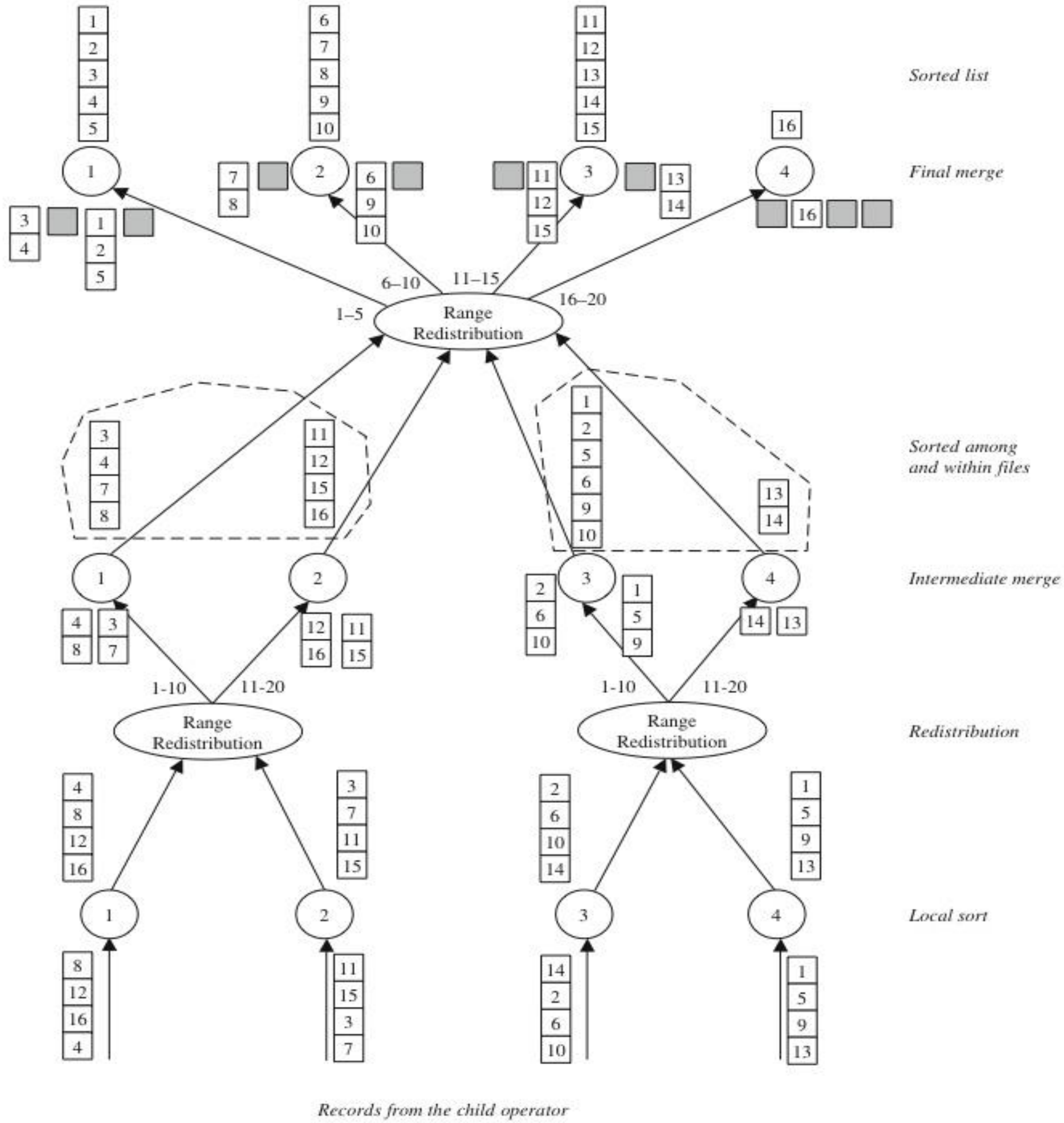


Figure 4.6 Parallel redistribution binary-merge sort

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
  - All processors are involved in merging at each level
  - Merging shorter arrays
- Skew problem in the merging

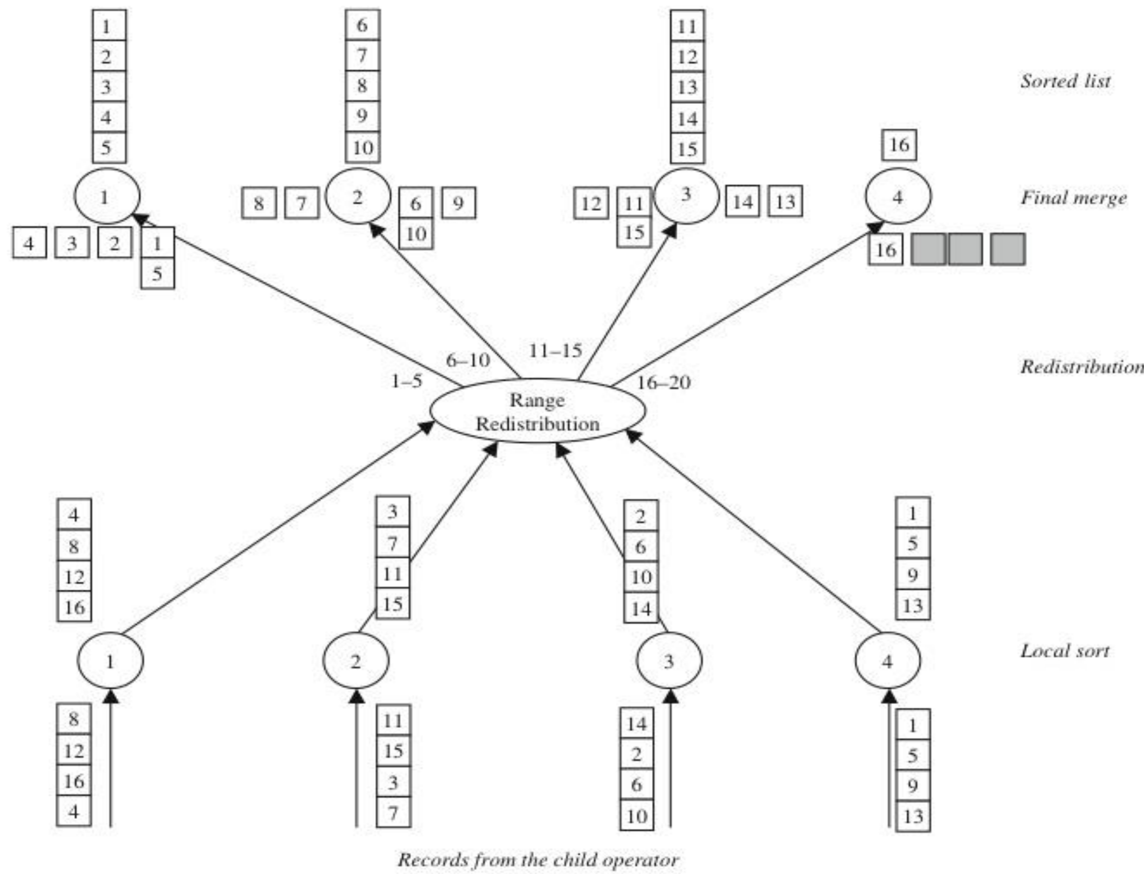


Figure 4.7 Parallel redistribution merge-all sort

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

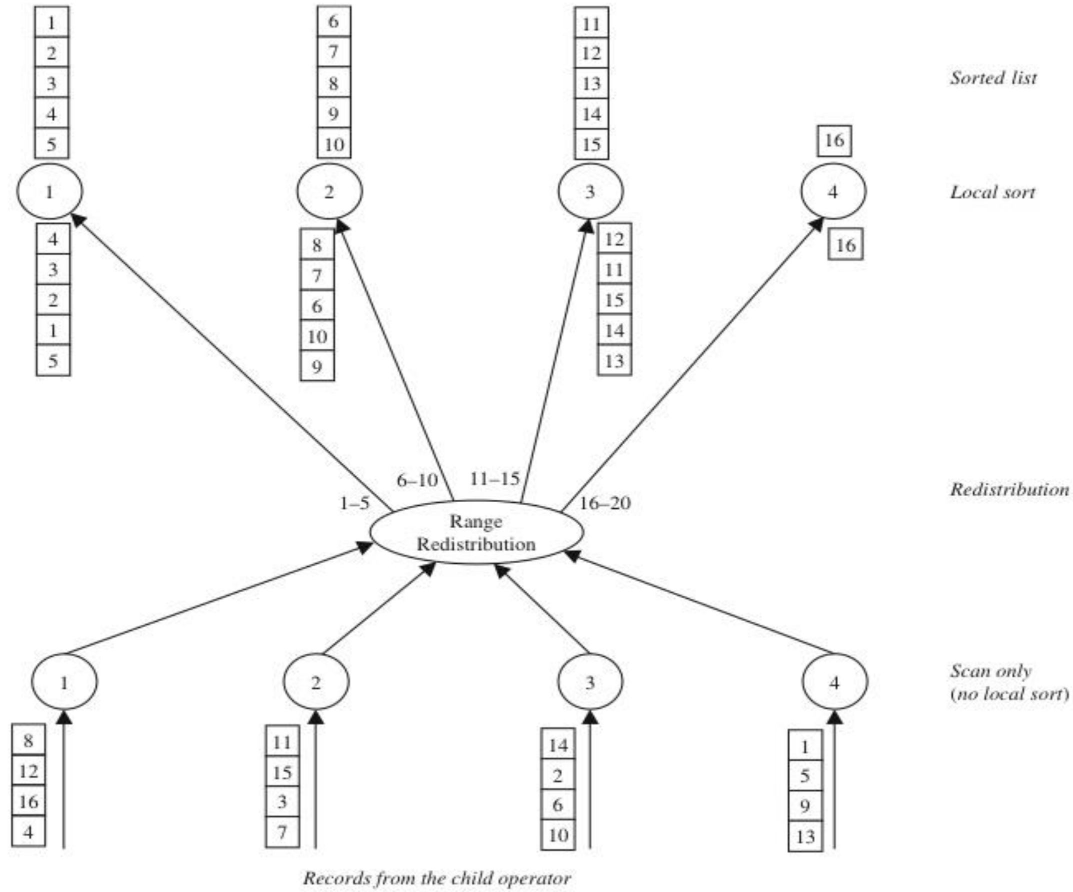


Figure 4.8 Parallel partitioned sort