Multicore Computing
Lecture11 - Parallel Sorting

SUNG KYUN KWAN UNIVERSITY

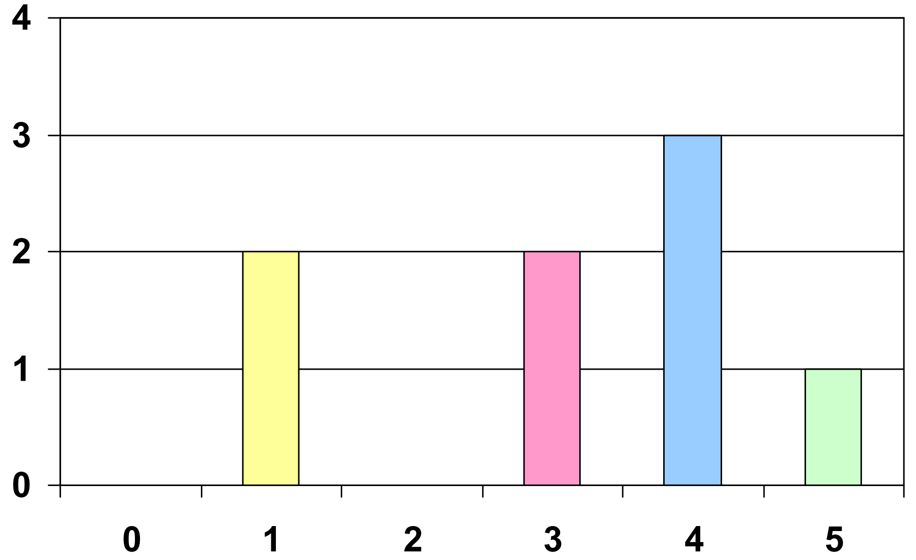
남 범 석
bnam@skku.edu

Sorting Algorithms

- Process of arranging unordered items into order
- Internal versus external sorting
  - External sorting uses auxiliary storage
- Comparison-based
  - Compare pairs of elements and exchange
  - e.g.) quick sort, merge sort, etc
  - $O(n \log n)$
- Non-comparison-based
  - Use known properties of elements
  - e.g.) counting sort, radix sort, etc
  - $O(n)$

Counting Sort

- Idea: build a histogram of the keys and compute position in answer array for each element
  - A = [3, 5, 4, 1, 3, 4, 1, 4]



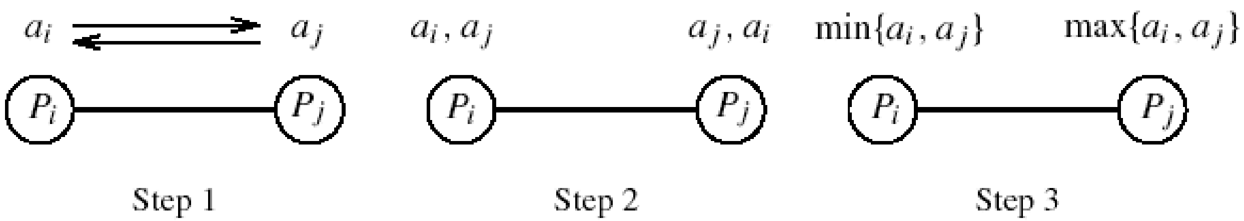
- Make temp array B, and write values into position
  - B = [1, 1, 3, 3, 4, 4, 4, 5]
  - Cost = O(#keys + size of histogram)
    - What if histogram too large (eg all 32-bit ints? All words?)

- $O(n \log n)$ optimal sequential sorting algorithm

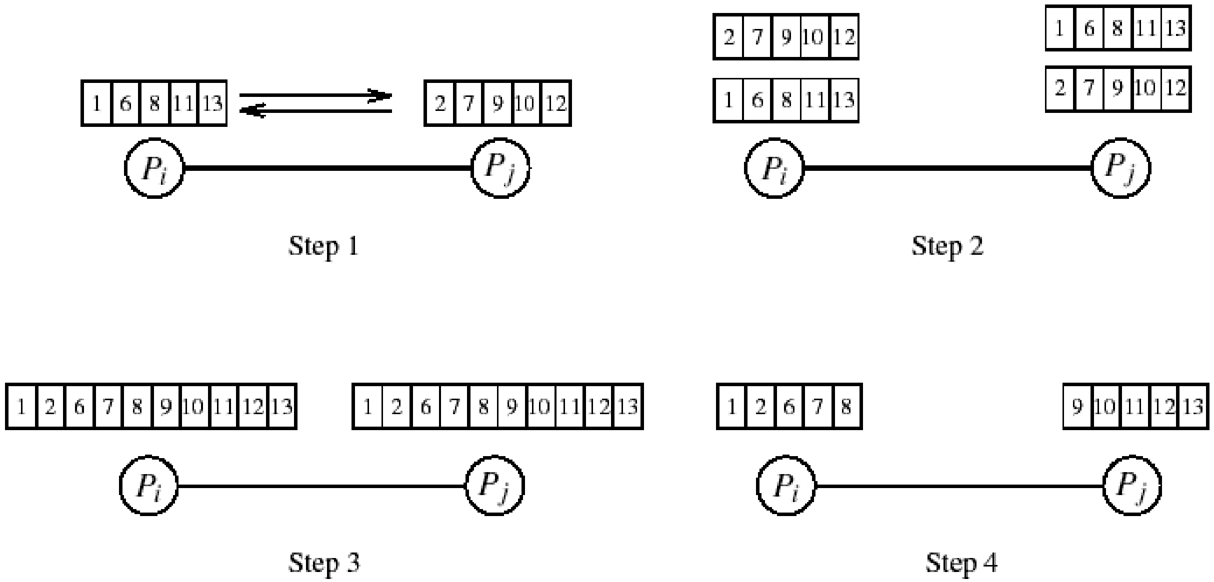
- Comparison-based parallel sorting using n processors:
  - $O(n \log n) / \mathrm{n} = O(\log n)$

Single vs. Multi Element Comparision

- One element per processor
  - Two numbers, say *A* and *B*, are compared between P_0 and P_1.



- Multiple elements per processor



Odd-Even Transposition Sort

Time Step	P_0	P_1	P_2	P_3	P_4	P_5	P_6	P_7
0	4	2	7	8	5	1	3	6
1	2	4	7	8	1	5	3	6
2	2	4	7	1	8	3	5	6
3	2	4	1	7	3	8	5	6
4	2	1	4	3	7	5	8	6
5	1	2	3	4	5	7	6	8
6	1	2	3	4	5	6	7	8
7	1	2	3	4	5	6	7	8

Parallel time complexity: $\boldsymbol{T_{par} = O(n)}$ (for P=n)

Odd-Even Transposition Sort – Example (N>>P)

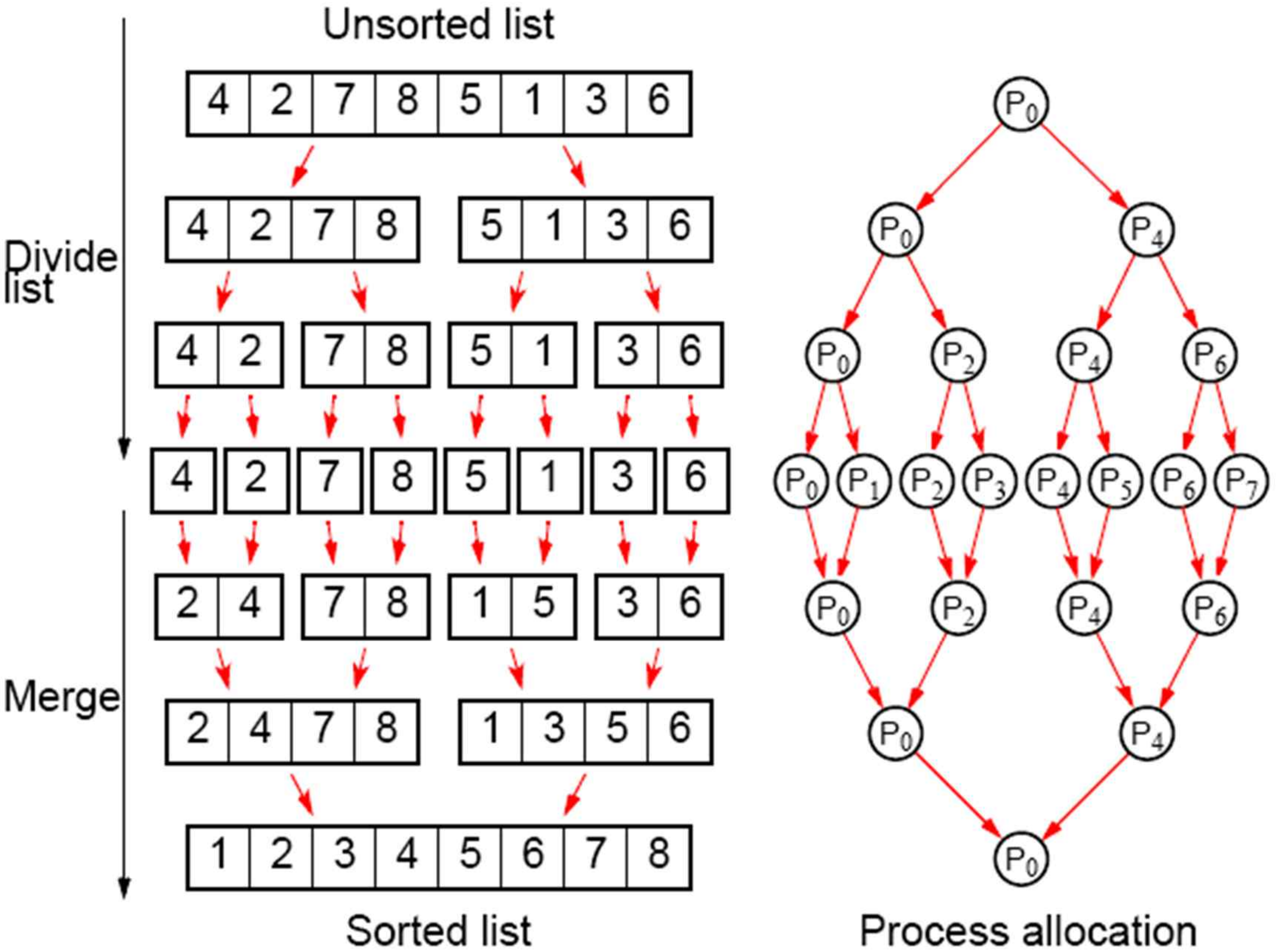
- Each processor gets n/p numbers. First, sort n/p locally, then run odd-even transposition algorithm each time doing a merge-split for 2n/p numbers.

	P_0	P_1	P_2	P_3
	13 7 12	8 5 4	6 1 3	9 2 10
Local sort	7 12 13	4 5 8	1 3 6	2 9 10
O-E	4 5 7	8 12 13	1 2 3	6 9 10
E-O	4 5 7	1 2 3	8 12 13	6 9 10
O-E	1 2 3	4 5 7	6 8 9	10 12 13
E-O				
SORTED:	1 2 3	4 5 6	7 8 9	10 12 13

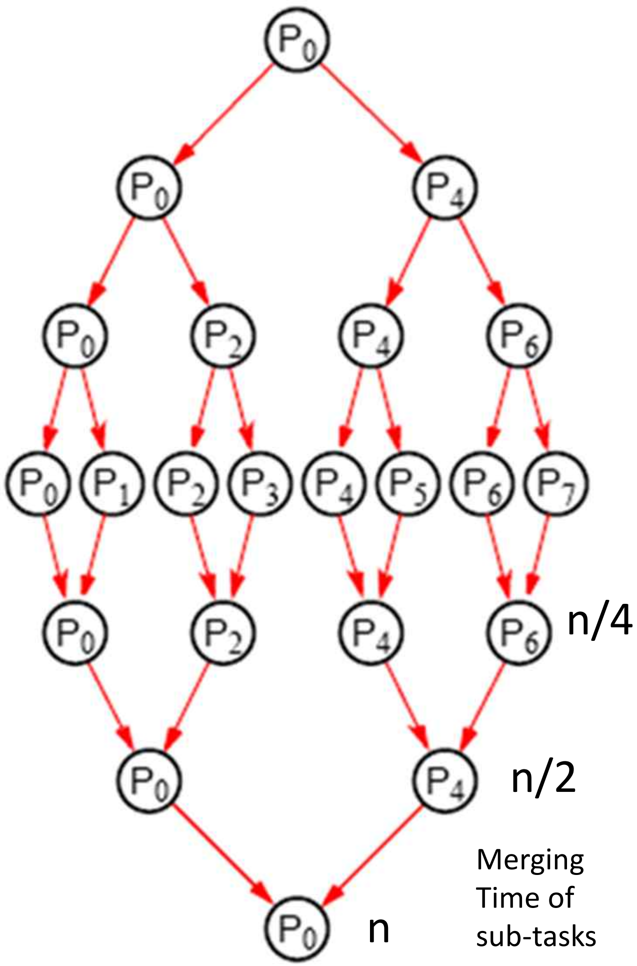
Time complexity: T_{par} = (Local Sort) + (p merge-splits) +(p exchanges)

$$T_{par} = (n/p)\log(n/p) + p*(n/p) + p*(n/p) = (n/p)\log(n/p) + 2n$$

Parallelizing Merge-Sort



Parallelizing Merge-Sort



Process allocation

Sequential :

$$T_{seq} = 1 * n + 2 * \frac{n}{2} + 2^2 * \frac{n}{2^2} + \ldots + 2^{\log n} * \frac{n}{2^{\log n}}$$

$$T_{seq} = O(n \log n)$$

Parallel :

$$T_{par} = 2\left(\frac{n}{2^0} + \frac{n}{2^1} + \frac{n}{2^2} + \ldots + \frac{n}{2^k} \ldots + 2 + 1\right)$$

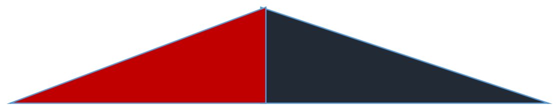
$$= 2n\left(2^0 + 2^{-1} + 2^{-2} + \ldots + 2^{-\log n}\right)$$

$$T_{par} = O(4n)$$

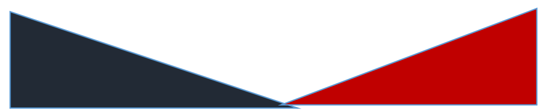
Bitonic Sort

- Create a *bitonic sequence* then sort the sequence
- *Bitonic sequence*
  - sequence of elements $<a_0, a_1, \ldots, a_{n-1}>$ *is bitonic if*
  - $\exists i, 0 \leq i \leq n-1$ $s.t$ $<a_0, a_1, \ldots, a_i>$ is monotonically increasing
  - *And* $<a_i, a_{i+1}, \ldots, a_{n-1}>$ is monotonically decreasing

  Or

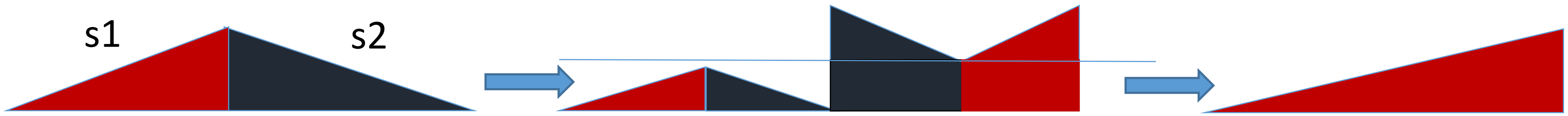


  - $\exists i, 0 \leq i \leq n-1$ $s.t$ $<a_0, a_1, \ldots, a_i>$ is monotonically decreasing
  - *And* $<a_i, a_{i+1}, \ldots, a_{n-1}>$ is monotonically increasing



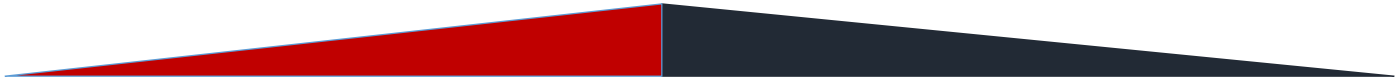
- For example,
  - <1,4,6,8,3,2> and <9,8,3,2,4,6> are both bitonic

Bitonic Merge: Sorting Bitonic Sequence



- Compare the first element of s1 with the first element of s2, then the second element of s1 with second element of s2 and so on.
- We exchange elements if an element of s2 is smaller.
- After above compare and exchange steps, we get two bitonic sequences of length n/2 such that all elements in first bitonic sequence are smaller than all elements of second bitonic sequence.
- We repeat the same process within two bitonic sequences and we get four bitonic sequences of length n/4 such that all elements of leftmost bitonic sequence are smaller and all elements of rightmost.
- If all these bitonic sequence are sorted and every bitonic sequence has one element, we get the sorted array.

Bitonic Merge - Example



- 3 5 8 9 10 12 14 20 95 90 60 40 35 23 18 0

- 3 5 8 9 10 12 14 0 | 95 90 60 40 35 23 18 20

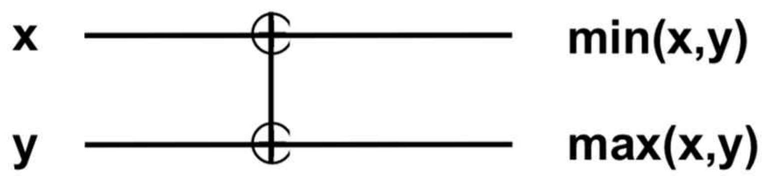
Smaller than 18 No smaller than 18

Smaller than 9 No smaller than 9 Smaller than 40 No smaller than 40

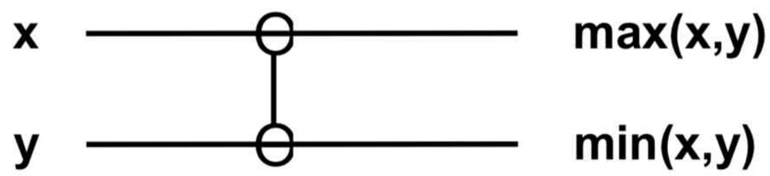
- 3 5 8 0 | 10 12 14 9 | 35 23 18 20 | 95 90 60 40
- 3 0 | 8 5 | 10 9 | 14 12 | 18 20 | 35 23 | 60 40 | 95 90
- 0 3 | 5 8 | 9 10 | 12 14 | 18 20 | 23 35 | 40 60 | 90 95

Sorting Network

- Network of comparators designed for sorting
- Comparator : two inputs x and y; two outputs x' and y'
  - Two types
  - increasing (denoted ⊕): x' = min(x,y) and y' = max(x,y)



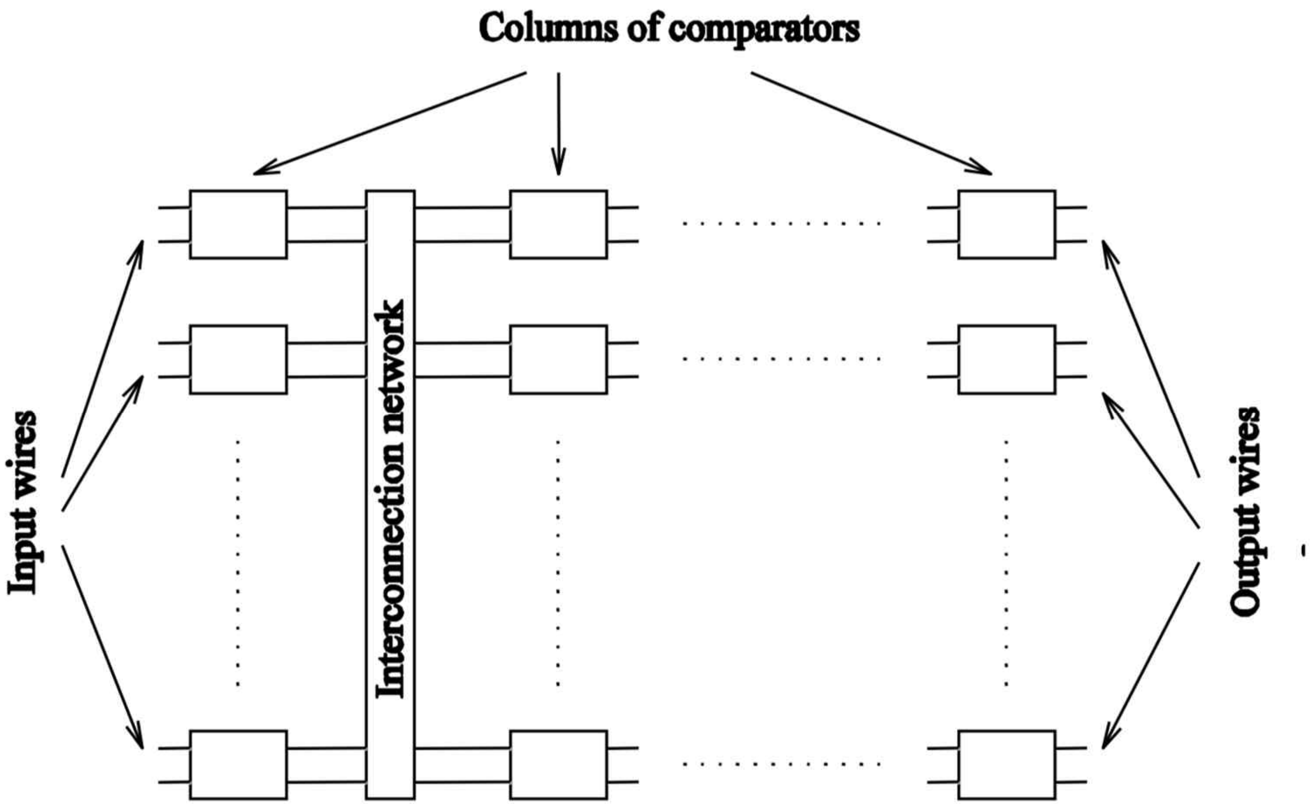
  - decreasing (denoted Ɵ) : x' = max(x,y) and y' = min(x,y)



- Sorting network speed is proportional to its depth

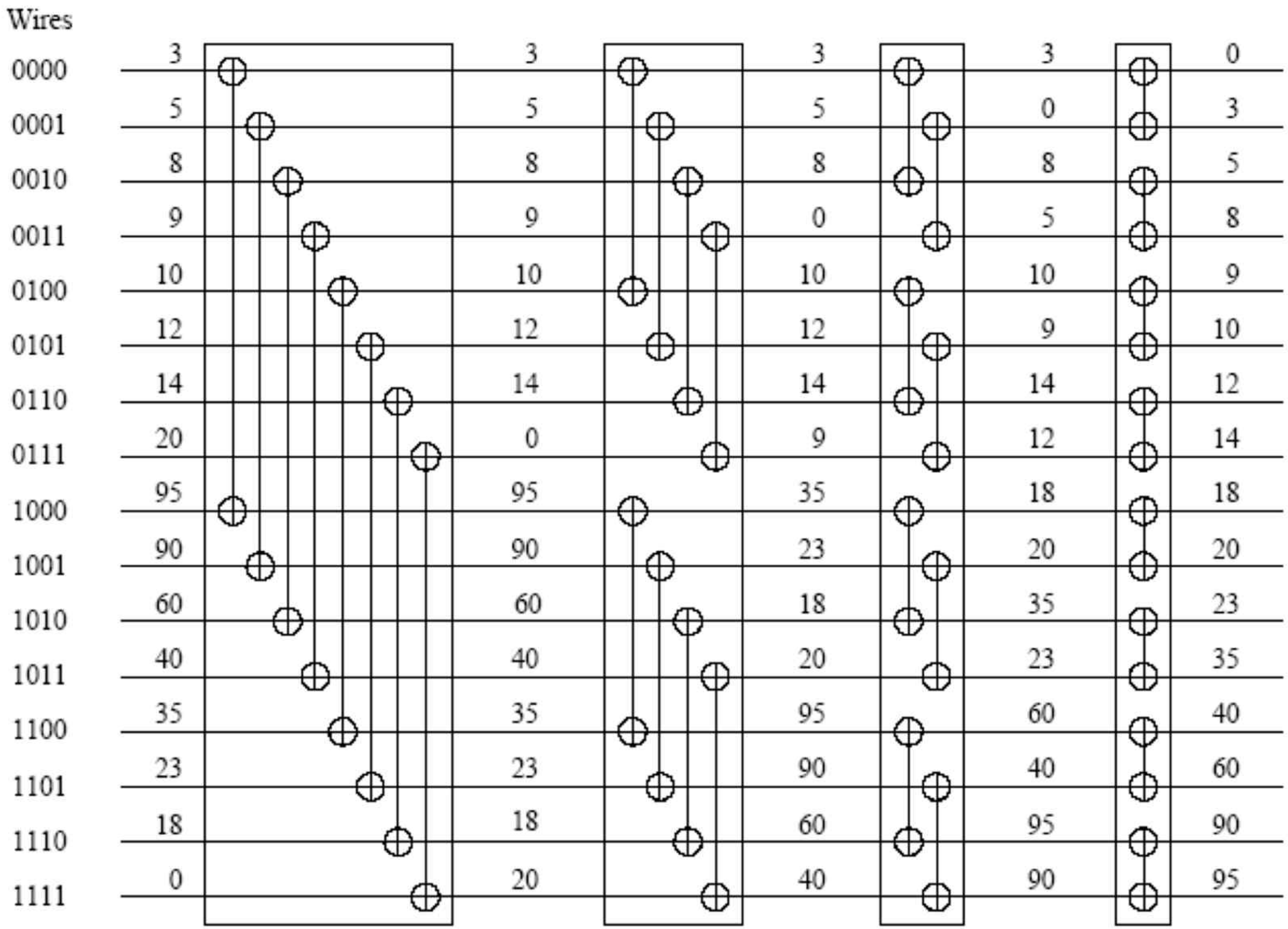
Sorting Network

- Network structure: a series of columns
- Each column consists of a vector of parallel comparators



- Input: Bitonic Sequence

⊕ **BM[16]**

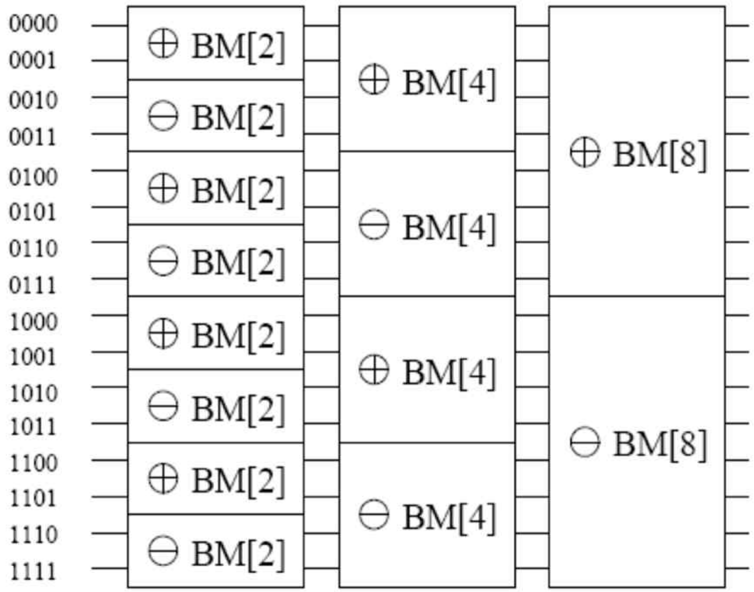


Bitonic Sort

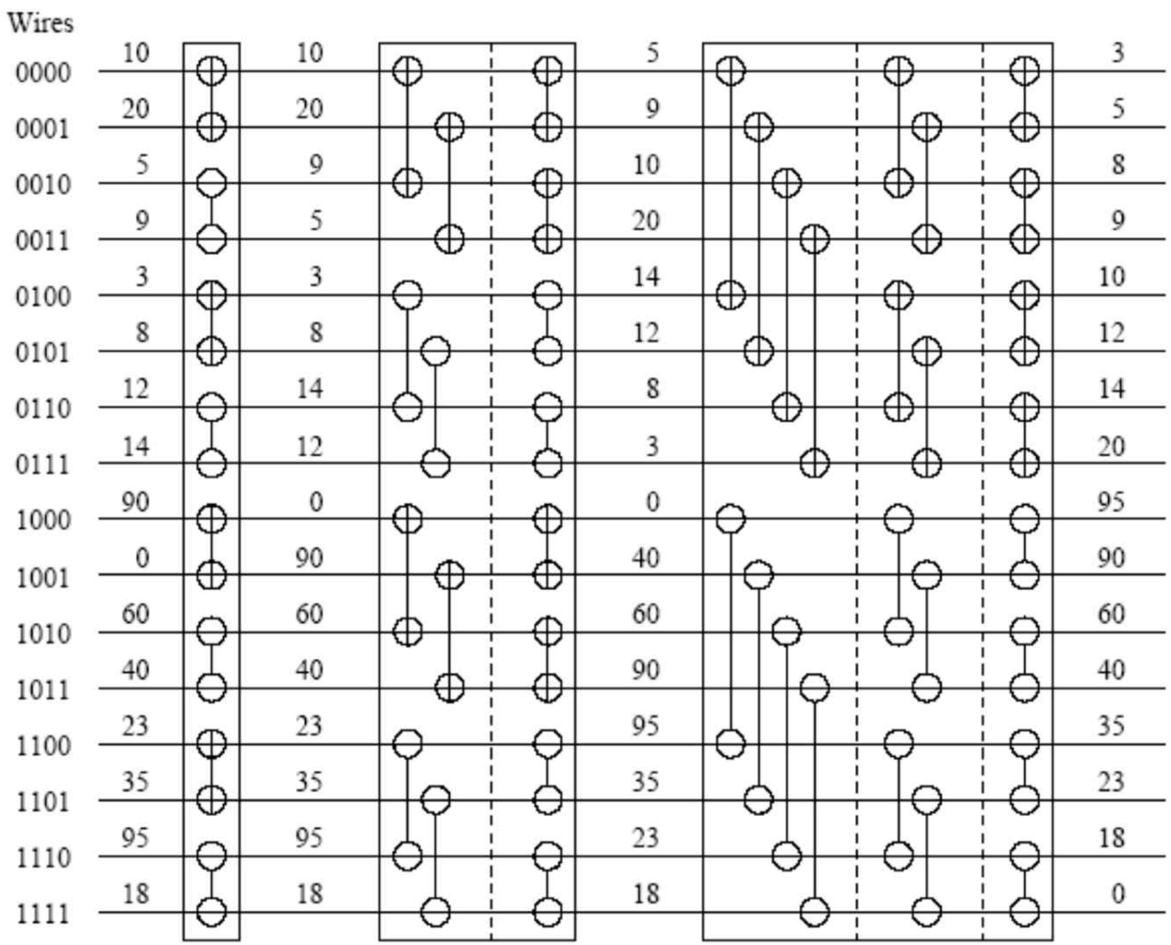
- Q: How to sort an unsorted sequence ?
- Two steps
  - Build a bitonic sequence
  - Sort it using a bitonic merging network

Building a Bitonic Sequence

- Build a single bitonic sequence from the given sequence
  - any sequence of length 2 is a bitonic sequence.
  - build bitonic sequence of length 4
    - sort first two elements using ⊕BM[2]
    - sort next two using ⊖BM[2]
- Repeatedly merge to generate larger bitonic sequences
  - ⊕BM[k] & ⊖BM[k]: bitonic merging networks of size k

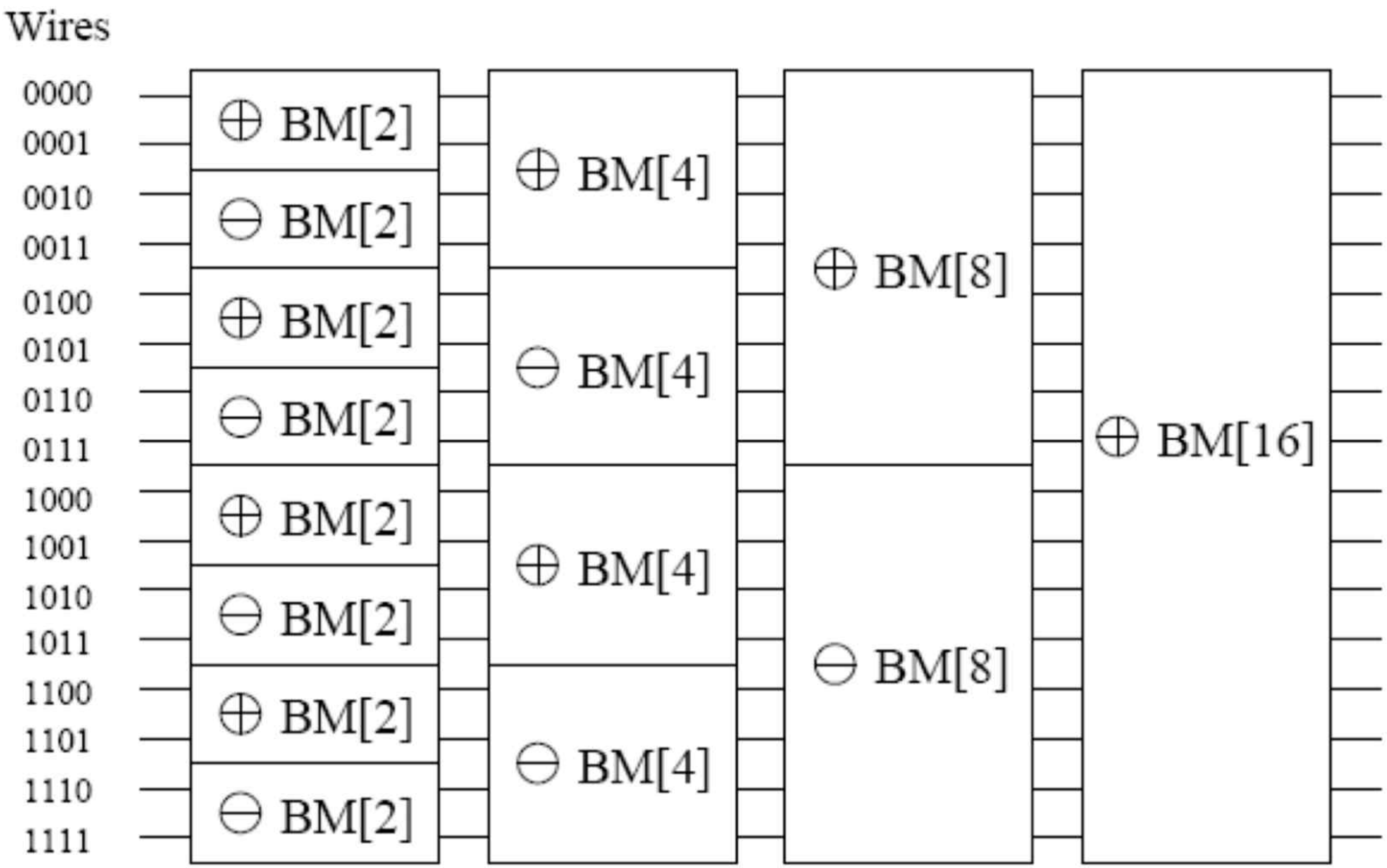


Building a Bitonic Sequence



- Input: sequence of 16 unordered numbers
- Output: a bitonic sequence of 16 numbers

Bitonic Sort, n = 16



- First 3 stages create bitonic sequence input to stage 4
- Last stage (⊕BM[16]) yields sorted sequence

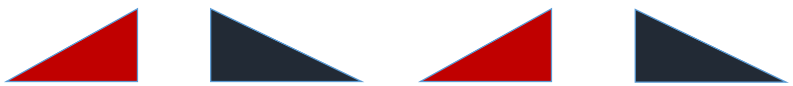
Bitonic Sort - Example

- Convert the following sequence to bitonic sequence:
- 3, 7, 4, 8, 6, 2, 1, 5

- [Step 1] Consider each 2-consecutive elements as bitonic sequence and apply bitonic sort on each 2- pair elements.

$$\begin{matrix} a \\ b \end{matrix} \Rightarrow \begin{matrix} \min(a,b) \\ \max(a,b) \end{matrix} \qquad \begin{matrix} a \\ b \end{matrix} \Rightarrow \begin{matrix} \max(a,b) \\ \min(a,b) \end{matrix}$$

3 7 4 8 6 2 1 5



→ 3 7 8 4 2 6 5 1

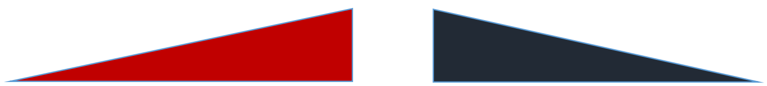
Bitonic Sort - Example

- Convert the following sequence to bitonic sequence:
- 3, 7, 4, 8, 6, 2, 1, 5

- [Step 2] In next step, take two 4 element bitonic sequences and so on.

→ 3 7 8 4 2 6 5 1

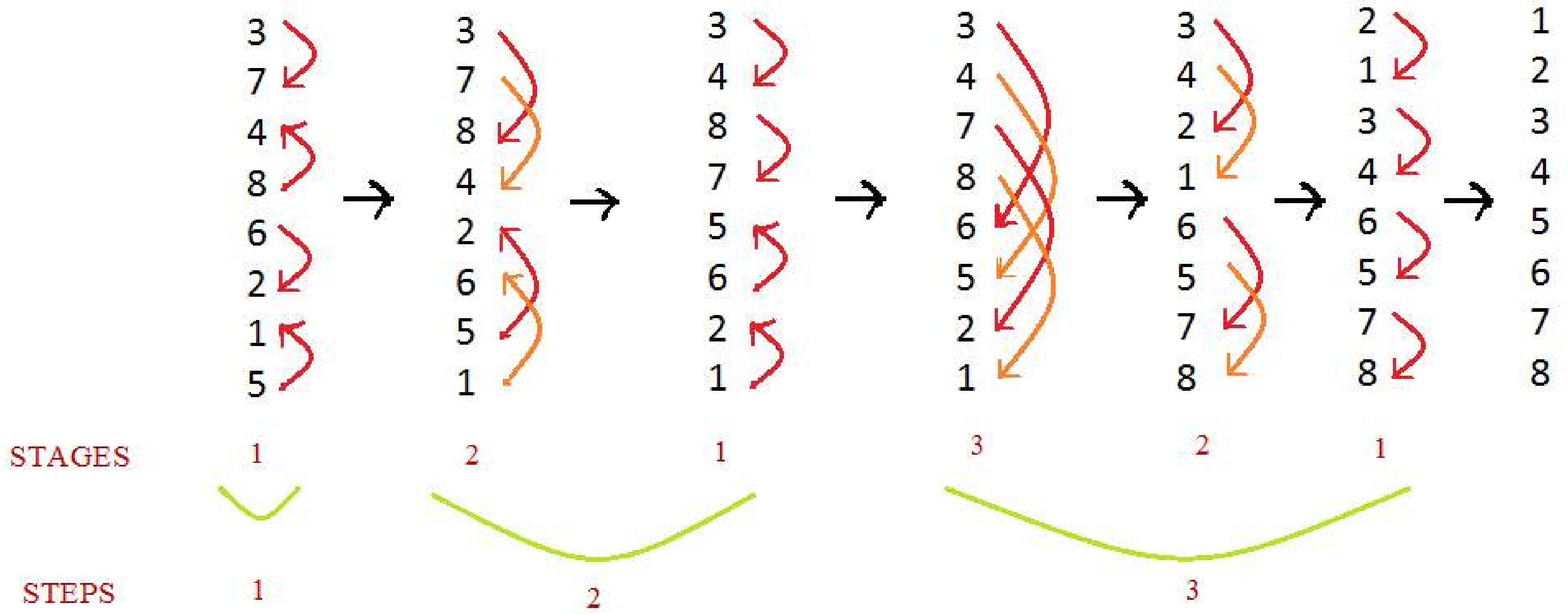
→ 3 4 8 7 5 6 2 1



→ 3 4 7 8 6 5 2 1

- Convert the following sequence to bitonic sequence:
- 3, 7, 4, 8, 6, 2, 1, 5

- [Step 3] In next step, take one 8 element bitonic sequence and do the bitonic merge.



Bitonic Sort – Sequential C code

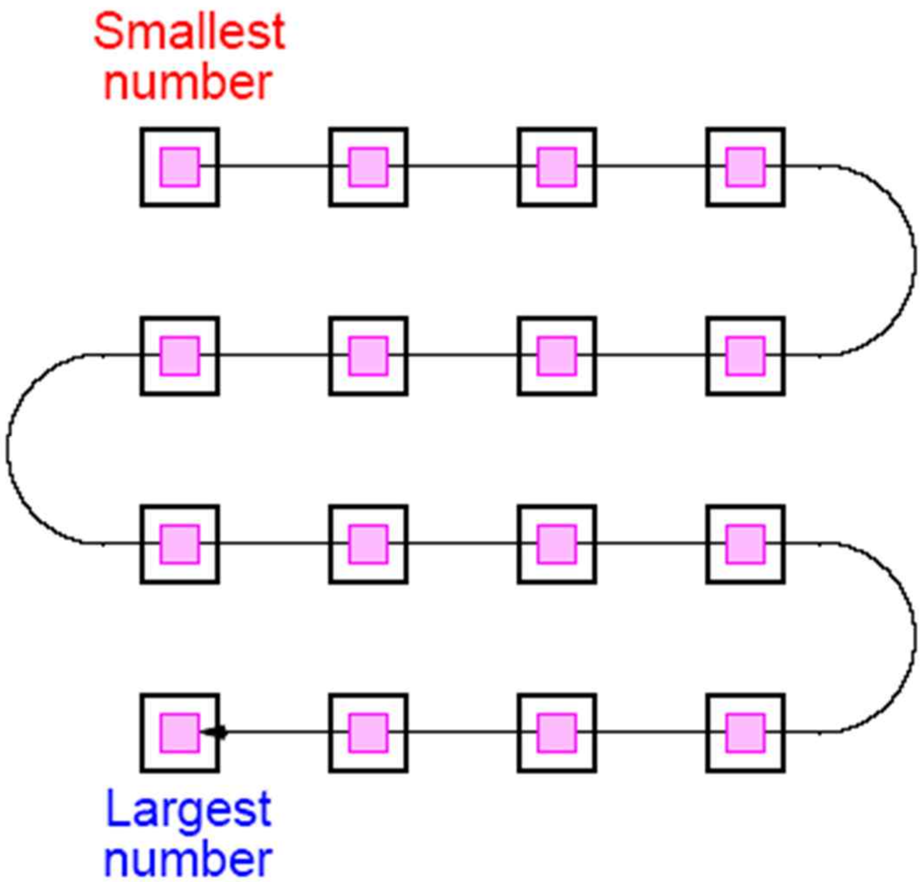
```
void bitonicSort(int a[],int low, int cnt, int dir)
{
   if (cnt>1)
   {
      int k = cnt/2;

      // sort in ascending order since dir here is 1
      bitonicSort(a, low, k, 1);

      // sort in descending order since dir here is 0
      bitonicSort(a, low+k, k, 0);

      // Will merge whole sequence in ascending order
      // since dir=1.
      bitonicMerge(a,low, cnt, dir);
   }
}
```

```
void bitonicMerge(int a[], int low, int cnt, int dir)
{
   if (cnt>1)
   {
      int k = cnt/2;
      for (int i=low; i<low+k; i++)
         compAndSwap(a, i, i+k, dir);
      bitonicMerge(a, low, k, dir);
      bitonicMerge(a, low+k, k, dir);
   }
}
```

Bitonic Sort

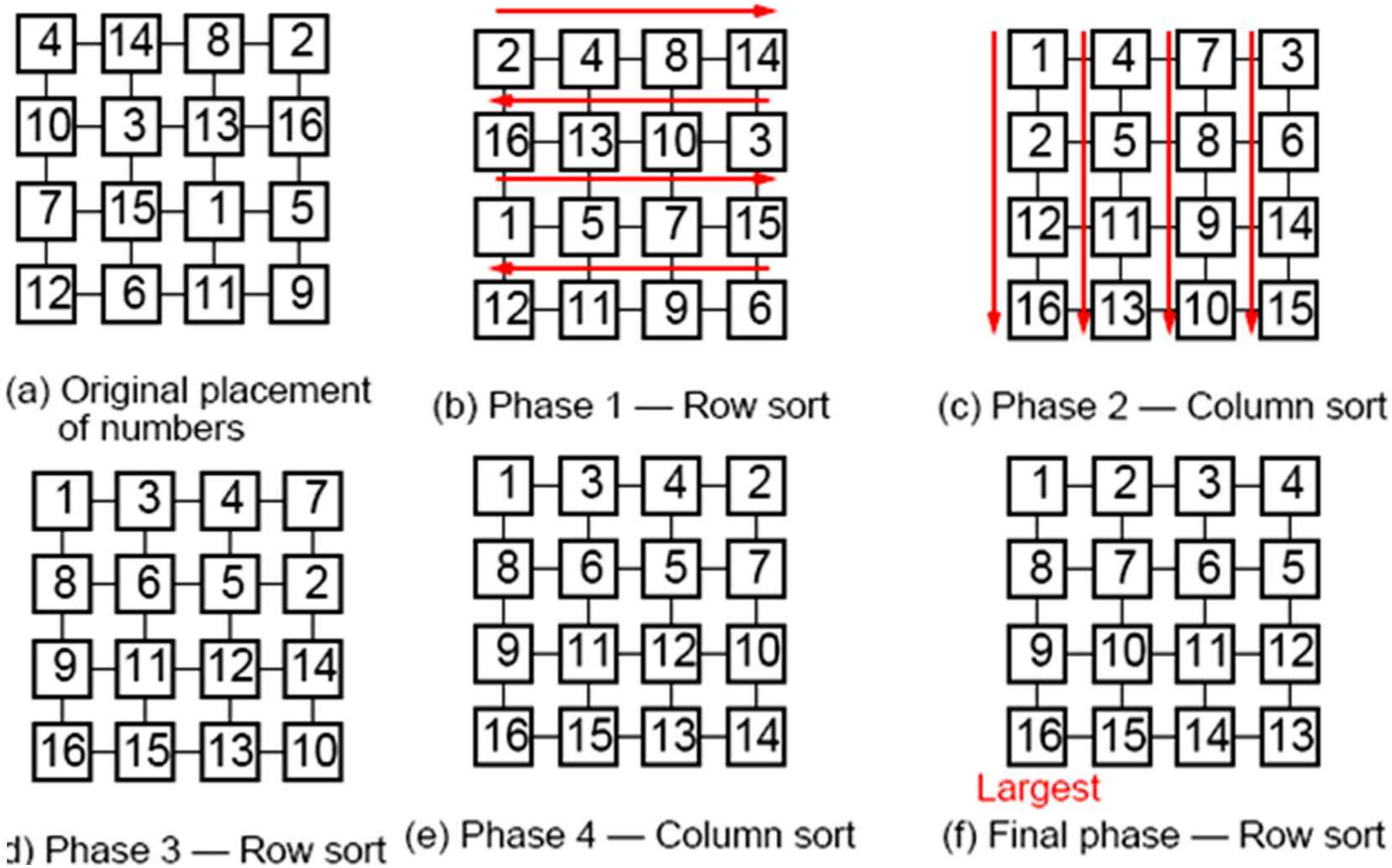
- A lot more comparisons than other sorting algorithms
- But
  - Compares elements in predefined sequence
  - Sequence of comparison doesn't depend on data
- Therefore, more suitable for parallel computing

- The layout of a sorted sequence on a mesh could be row by row or snakelike:



Shear Sort

- Alternate row and column sorting until list is fully sorted.
- Alternate row directions to get snake-like sorting:



Shear Sort - Complexity

- One a nxn mesh, it takes 2 log(n) phases to sort n^2 numbers.

$$T_{par}^{shearsort} = O(n \log n) \quad \textbf{on a n x n mesh}$$

- Sequential sorting takes T = O(n^2 log n). Therefore,

$$Speedup_{shearsort} = \frac{T_{seq}}{T_{par}} = O(n) \quad (\textbf{for } \mathbf{P = n^2})$$

$$\textbf{However, } efficiency = \frac{1}{n}$$

- Assume numbers are in positional digit representation.
  - The digits represent values
  - The position of each digit indicates their relative weighting
  - E.g., binary or decimal numbers

- Radix sort starts at the least significant digit and sorts the numbers according to their digits.
- The sequence is then sorted according to the next least significant digit and so on until the most significant digit, after which the sequence is sorted.

Radix Sort: Separate Key Into Parts

- Divide keys into parts, e.g., by digits (radix)
- Using counting sort on these each radix:
  - Start with least-significant

sat	run	sat	pin
saw	pin	saw	run
tip	tip	pin	sat
run	sat	tip	saw
pin	saw	run	tip

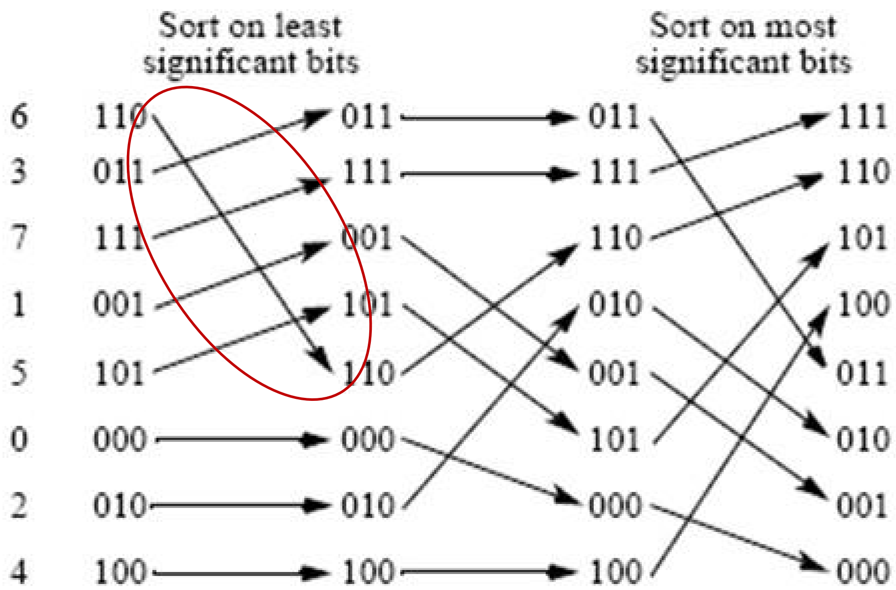
sort on 3rd character

sort on 2nd character

sort on 1st character

- Cost = O(#keys * #characters)

- Example: Radix Sort using binary digits



- Ok, it's an easy algorithm.
- But how do we know where to put each element, i.e., new location, if it has to move?
  - → Build histograms

- Overview:
- 1. Count occurrences of each digit/process pair
  - Histogram[p][d]: Occurrences of digit d on process p
  - Local computation
- 2. Find global block offset for each digit/process pair
  - offset[p][d]: Block offset for digit d on proc p
  - Prefix sum of Histogram[][] in column-major order
- 3. Find destination index for each input element
  - index[p][k]: Destination for element k on p
- 4. Apply the input-to-output permutation
  - output[index[k]]=input[k]
  - Permutation / all-to-all

Parallel Radix Sort

- Build a histogram for each bit

```
Element #    0    1    2    3    4
Value:       7   14    4    1    6
Binary:   0111 1110 0100 0001 0110
bit 0:       1    0    0    1    0
```

- →

	Zero bits	One bit
Histogram:	3	2

- prefix sum on these histogram values
- →

	Zero bits	One bit
Histogram:	3	2
Prefix Sum:	0	3

- Determine the relative offset using prefix sum

```
Element #    0    1    2    3    4
Value:       7   14    4    1    6
Binary:   0111 1110 0100 0001 0110
bit 0:       1    0    0    1    0
offset:      0    0    1    1    2
```

- Determine the location to move

```
Element #     0    1    2    3    4
Value:        7   14    4    1    6
Binary:    0111 1110 0100 0001 0110
bit 0:        1    0    0    1    0
offset:       0    0    1    1    2
index:        3    0    1    4    2
```

```
Element #     0    1    2    3    4
Value:       14    4    6    7    1
Binary:    1110 0100 0110 0111 0001
```

- Repeat for the next bit

- Q: Complexity?