Sorting

”The Good, the Bad and the Ugly”

Today: The Bad!

Sorting Algorithms

A number of $\Theta(n^2)$ algorithms:
• Selection sort
• Buble sort
• Insertion sort

Several O(n log n) algorithms:
• Quicksort
• Mergesort
• Heapsort

Other sorting algorithms
• Counting sort
• Radixsort
• Bucketsort

Selection sort

Given array A[0:n-1] of unsorted numbers
Order elements of A in increasing order

<u>General idea:</u>
Find the smallest element, swap with the first element
Repeat n-2 times

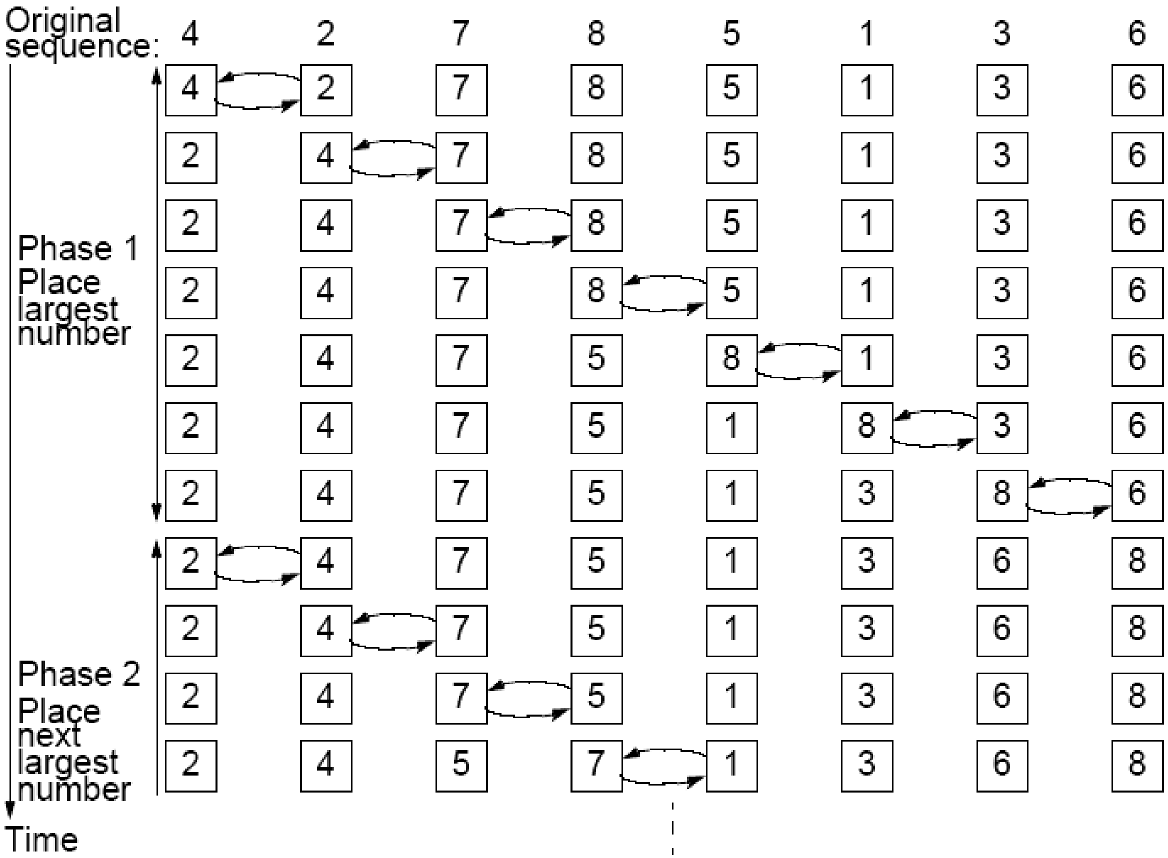
```
For i = 0 to n-2
    smallest = A[i]
    position = i

    For j=i+1 to n
        If A[j] < smallest
            smallest = A[i]
            postition = j

    Swap A[i] and A[position]
```

Bubble sort

For i = n-1 to 1 // Bubble i+1'st largest element to index i
  For j = 1 to i
    compare-swap a_j and a_{j+1}



Time:

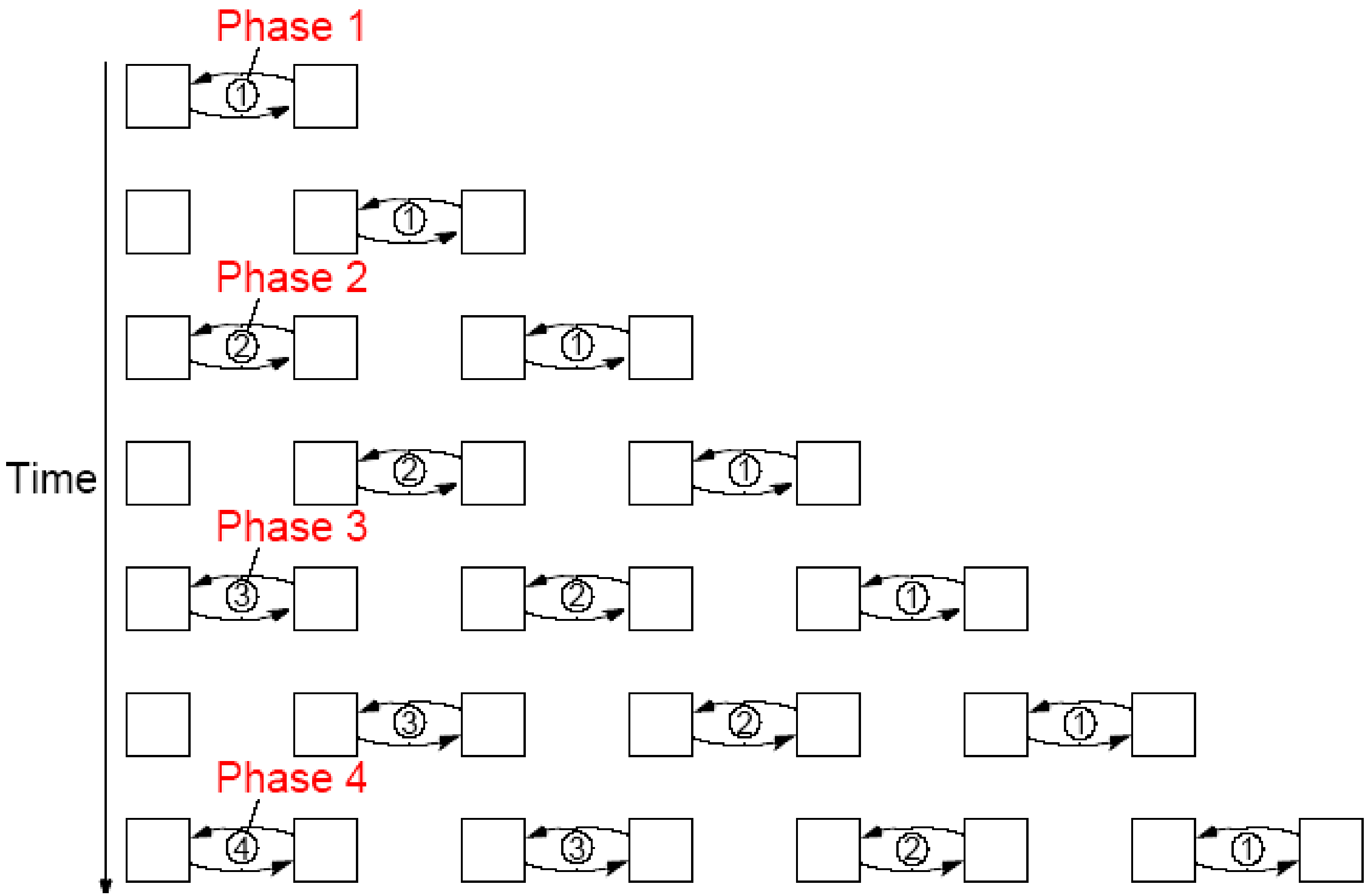
$\sum_{i=1}^{n-1} i = n(n-1)/2$

$= \Theta(n^2)$

Note:
With one thread per element the second stage could start earlier.

Parallel Bubble Sort

Phase i starts as soon as possible as long as it does not overtake iteration i+1.



With p = n the algorithm takes time Θ(n).
Perfect speedup compared to sequential bubblesort,
but only log(n) compared to mergesort.

Odd-Even Transposition Sort

"Use all the threads in each round."

For step = 1, n

Even step: Even p_i talks to p_{i+1}, Odd p_i talks to p_{i-1}

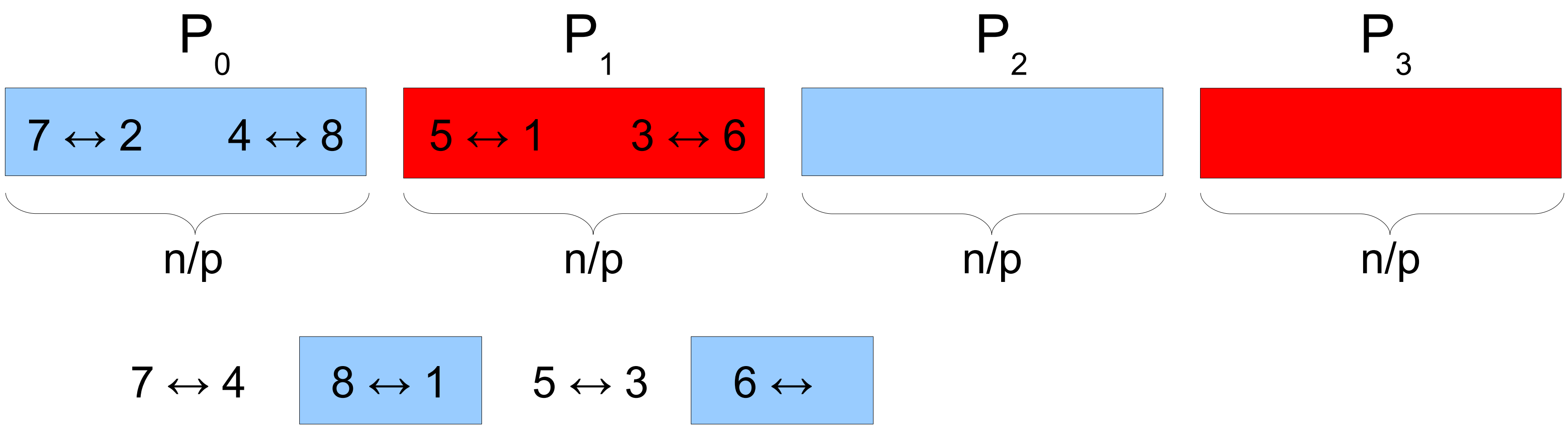
Odd step: Even p_i talks to p_{i-1}, Odd p_i talks to p_{i+1}

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

Time

Practical Implementation

Since p << n each thread must handle n/p elements in each iteration.

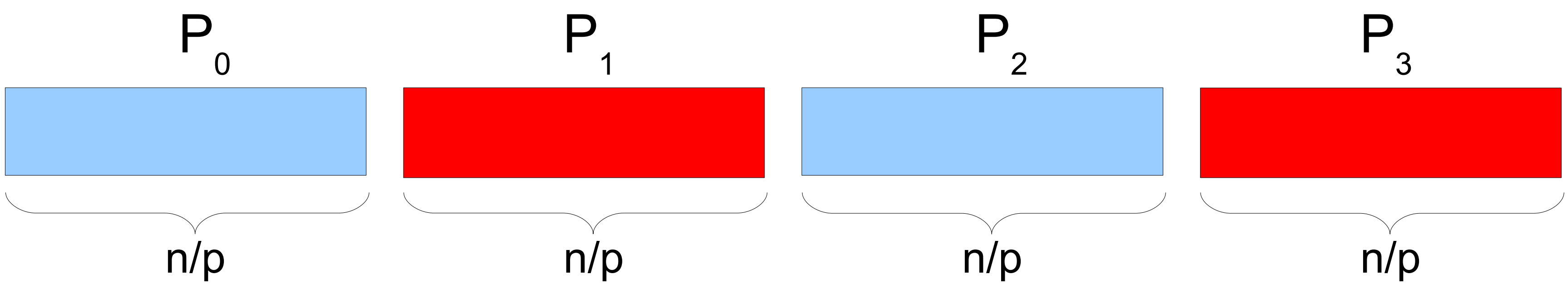


n iterations, O(n/p) work in each iteration

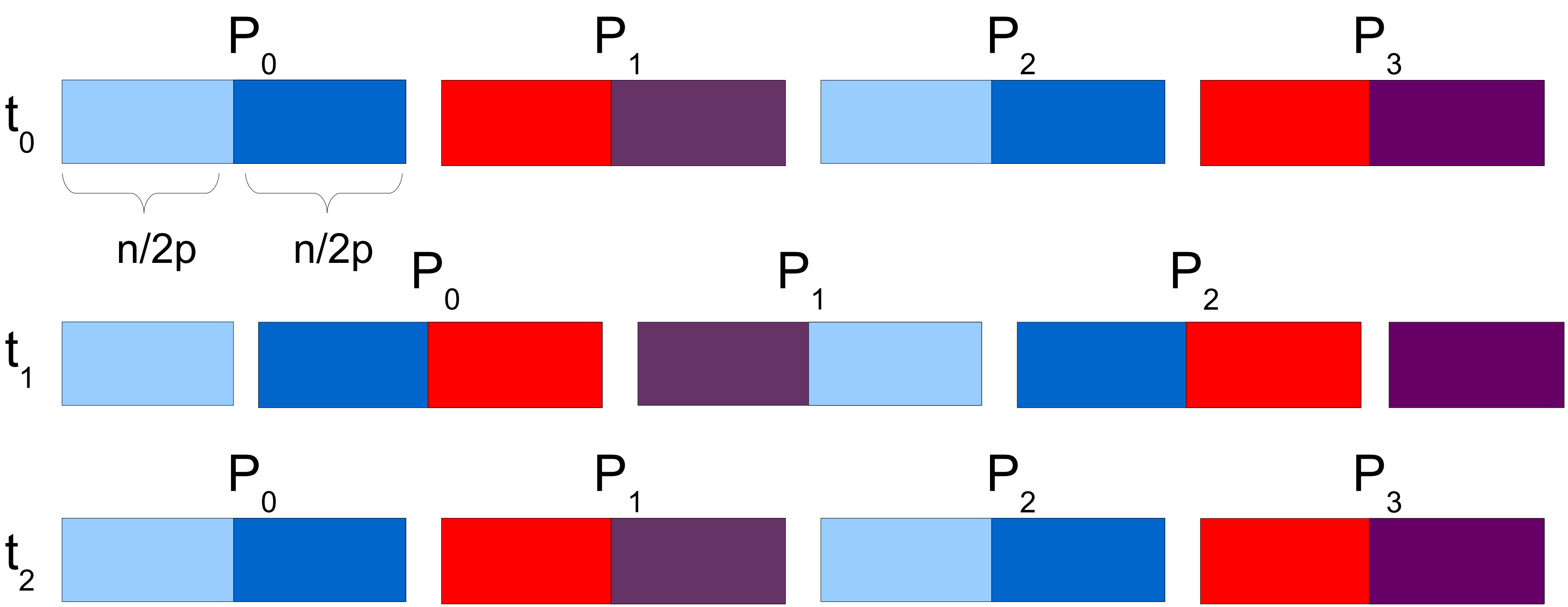
$\rightarrow T_p = O(n^2/p)$

Practical Implementation

Since p << n each thread must handle n/p elements in each iteration.



Could move anywhere from 1 to n/p elements between processes



Practical Implementation

Better strategy: Pre-sort lists of n/(2p) elements each

Perform 2p steps of block merge, each taking time Θ(n/p).

$$T_p = \underbrace{O(n/p \log(n/p))}_{\text{Presort}} + \underbrace{\Theta(n)}_{2p*n/p}$$

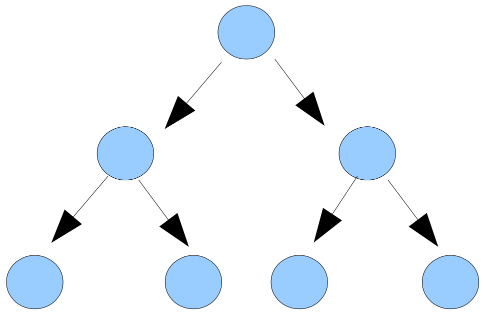
Note:

- Can stop when there is no data movement
- Could perform initial long range movements
- Might make subsequent algorithm faster

Divide and Conquer

From sequential algorithms:

Quicksort
Work is in partitioning



Mergesort
Work is in putting things together

Tasks on each level are typically independent

In parallel:
Divide until parts are of size n/p, and then use a sequential algorithm
Can also divide in more than two parts on each level.

Parallel Mergesort

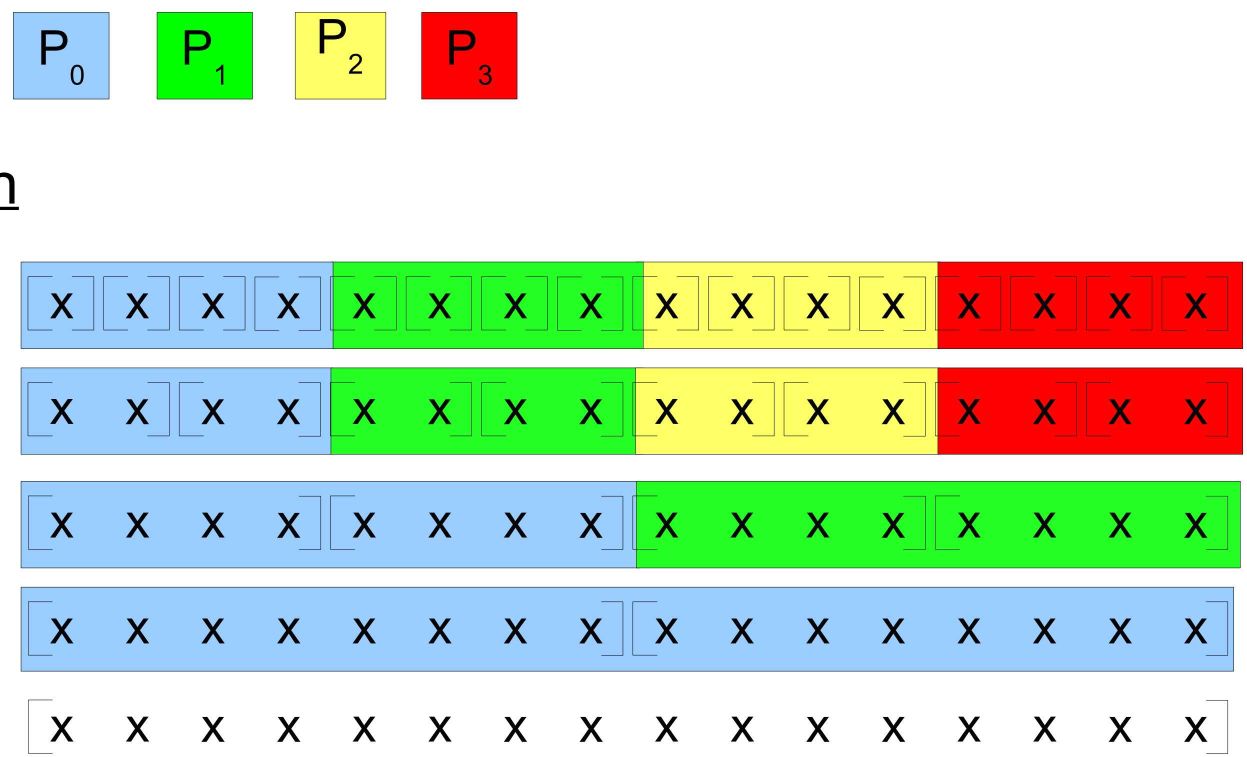
Sequential:

```
for(i=1; i<n; i*=2)
  merge two and two
  adjacent lists of length i
```

- log n levels
- merge takes O(n) time on each level

$\rightarrow t_{seq} = O(n \log(n))$

Parallel:



Level	Length
0	1
1	2
2	4
3	8
Sorted	16

Length = n/p

Parallel Running Time

1. stage. Every processor sorts n/p elements, $T1_p = O(n/p \log(n/p))$

2. stage. Length of lists doubles for every level,

- starts with length n/p, ends with length n
- Length = n/p, 2n/p, 4n/p, ... , n
- Total of log p steps

Thread 0 dominates running time of second stage

$T2_p = 2^1 n/p + 2^2 n/p + 2^3 n/p \ldots + 2^{\log(p)} n/p$ (note $2^{\log(p)} = p$)

$= n/p * (2^1 + 2^2 + 2^3 \ldots + 2^{\log(p)}) = n/p * (2p-1) = 2n - n/p$

$T_p = T1_p + T2_p = 2n + n/p (\log(n/p) - 1)$

$S = T_{seq}/T_p \approx n \log(n) / (2n + n/p \log(n/p)) < ½ \log(n)$

Bucket Sort

Input: **n** integers **a[0..n-1]** (evenly distributed) on **[0,b-1]**
Output: **a[0] ≤ a[1] ≤ ... ≤ a[n-1]**

Example (**n=8, b=24**): **a = {9, 15, 3, 0, 6, 21, 18, 12}**

Algorithm:
Divide interval **[0,b-1]** into **m** buckets of equal width:

Example (**m=4**): Bucket width = **b/m = 24/4=6**
[0,6), [6,12), [12,18), [18,24)

Step 0: Put **a[0..n-1]** in buckets:
{}, {}, {}, {}

Bucket Sort

Input: **n** integers **a[0..n-1]** (evenly distributed) on **[0,b-1]**
Output: **a[0] < a[1] < ... < a[n-1]**

Example (**n=8, b=24**): **a = {9, 15, 3, 0, 6, 21, 18, 12}**

Algorithm:
Divide interval **[0,b-1]** into **m** buckets of equal width:

Example (**m=4**): Bucket width = **b/m = 24/4=6**
[0,6), [6,12), [12,18), [18,24)

Step 0: Put **a[0..n-1]** in buckets:
{}, {9}, {}, {}

Bucket Sort

Input: **n** integers **a[0..n-1]** (evenly distributed) on **[0,b-1]**
Output: **a[0] < a[1] < ... < a[n-1]**

Example (**n=8, b=24**): **a = {9, 15, 3, 0, 6, 21, 18, 12}**

Algorithm:
Divide interval **[0,b-1]** into **m** buckets of equal width:

Example (**m=4**): Bucket width = **b/m = 24/4=6**
[0,6), [6,12), [12,18), [18,24)

Step 0: Put **a[0..n-1]** in buckets:
{}, {9}, {15}, {}

Bucket Sort

Input: **n** integers **a[0..n-1]** (evenly distributed) on **[0,b-1]**
Output: **a[0] < a[1] < ... < a[n-1]**

Example (**n=8, b=24**): **a = {9, 15, 3, 0, 6, 21, 18, 12}**

Algorithm:
Divide interval **[0,b-1]** into **m** buckets of equal width:

Example (**m=4**): Bucket width = **b/m = 24/4=6**
[0,6), [6,12), [12,18), [18,24)

Step 0: Put **a[0..n-1]** in buckets:
{3}, {9}, {15}, {}

Bucket Sort

Input: **n** integers **a[0..n-1]** (evenly distributed) on **[0,b-1]**
Output: **a[0] < a[1] < ... < a[n-1]**

Example (**n=8, b=24**): **a = {9, 15, 3, 0, 6, 21, 18, 12}**

Algorithm:
Divide interval **[0,b-1]** into **m** buckets of equal width:

Example (**m=4**): Bucket width = **b/m = 24/4=6**
[0,6), [6,12), [12,18), [18,24)

Step 0: Put **a[0..n-1]** in buckets:
{3, 0}, {9}, {15}, {}

Bucket Sort

Input: **n** integers **a[0..n-1]** (evenly distributed) on **[0,b-1]**
Output: **a[0] < a[1] < ... < a[n-1]**

Example (**n=8, b=24**): **a = {9, 15, 3, 0, 6, 21, 18, 12}**

Algorithm:
Divide interval **[0,b-1]** into **m** buckets of equal width:

Example (**m=4**): Bucket width = **b/m = 24/4=6**
[0,6), [6,12), [12,18), [18,24)

Step 0: Put **a[0..n-1]** in buckets:
{3, 0}, {9, 6}, {15}, {}

Bucket Sort

Input: **n** integers **a[0..n-1]** (evenly distributed) on **[0,b-1]**
Output: **a[0] < a[1] < ... < a[n-1]**

Example (**n=8, b=24**): **a = {9, 15, 3, 0, 6, 21, 18, 12}**

Algorithm:
Divide interval **[0,b-1]** into **m** buckets of equal width:

Example (**m=4**): Bucket width = **b/m = 24/4=6**
[0,6), [6,12), [12,18), [18,24)

Step 0: Put **a[0..n-1]** in buckets:
{3, 0}, {9, 6}, {15}, {21}

Bucket Sort

Input: **n** integers **a[0..n-1]** (evenly distributed) on **[0,b-1]**
Output: **a[0] < a[1] < ... < a[n-1]**

Example (**n=8, b=24**): **a = {9, 15, 3, 0, 6, 21, 18, 12}**

Algorithm:
Divide interval **[0,b-1]** into **m** buckets of equal width:

Example (**m=4**): Bucket width = **b/m = 24/4=6**
[0,6), [6,12), [12,18), [18,24)

Step 0: Put **a[0..n-1]** in buckets:
{3, 0}, {9, 6}, {15}, {21, 18}

Bucket Sort

Input: **n** integers **a[0..n-1]** (evenly distributed) on **[0,b-1]**
Output: **a[0] < a[1] < ... < a[n-1]**

Example (**n=8, b=24**): **a = {9, 15, 3, 0, 6, 21, 18, 12}**

Algorithm:
Divide interval **[0,b-1]** into **m** buckets of equal width:

Example (**m=4**): Bucket width = **b/m = 24/4=6**
[0,6), [6,12), [12,18), [18,24)

Step 0: Put **a[0..n-1]** in buckets:
{3, 0}, {9, 6}, {15, 12}, {21, 18}

Bucket Sort

Input: **n** integers **a[0..n-1]** (evenly distributed) on **[0,b-1]**
Output: **a[0] < a[1] < ... < a[n-1]**

Example (**n=8, b=24**): **a = {9, 15, 3, 0, 6, 21, 18, 12}**

Algorithm:
Divide interval **[0,b-1]** into **m** buckets of equal width:

Example (**m=4**): Bucket width = **b/m = 24/4=6**
[0,6), [6,12), [12,18), [18,24)

Step 0: Put **a[0..n-1]** in buckets:
{3, 0}, {9, 6}, {15, 12}, {21, 18}
Step 1: Sort buckets:
{0, 3}, {6, 9}, {12, 15}, {18, 21}

Bucket Sort

Input: **n** integers **a[0..n-1]** (evenly distributed) on **[0,b-1]**
Output: **a[0] < a[1] < ... < a[n-1]**

Example (**n=8, b=24**): **a = {9, 15, 3, 0, 6, 21, 18, 12}**

Algorithm:
Divide interval **[0,b-1]** into **m** buckets of equal width:

Example (**m=4**): Bucket width = **b/m = 24/4=6**
[0,6), [6,12), [12,18), [18,24)

Step 0: Put **a[0..n-1]** in buckets:
{3, 0}, {9, 6}, {15, 12}, {21, 18}
Step 1: Sort buckets:
{0, 3}, {6, 9}, {12, 15}, {18, 21}
Step 2: Concatenate buckets (into **a**):
{0, 3, 6, 9, 12, 15, 18, 21}

Bucket Sort

`0    b/m-1  b/m   2b/m-1  . . . . .   (m-1)b/m  b-1`

Bucket number |0| |1| |m-1| **m** buckets

Step 0: **a[0],...,a[n-1]** are put in their corresponding bucket:

  w = b/m = bucket width

  for **i=0,...,n-1**

    Find bucket index: **j = a[i]/w**

    Put **a[i]** in bucket **j**

Step 1: Sorting:

  for **j=0,..., m-1**

    Sort bucket **j**

Step 2: Concatenate buckets:

  for **j=0,..., m-1**

    Move content of bucket **j** to **a**

Bucket Sort

Expected running time:

Step 0: Put in buckets:
    O(n)

Step 1: Sort buckets (using MergeSort or HeapSort):
    O(m n/m log(n/m)) (assuming a[0..n-1] are distributed evenly)

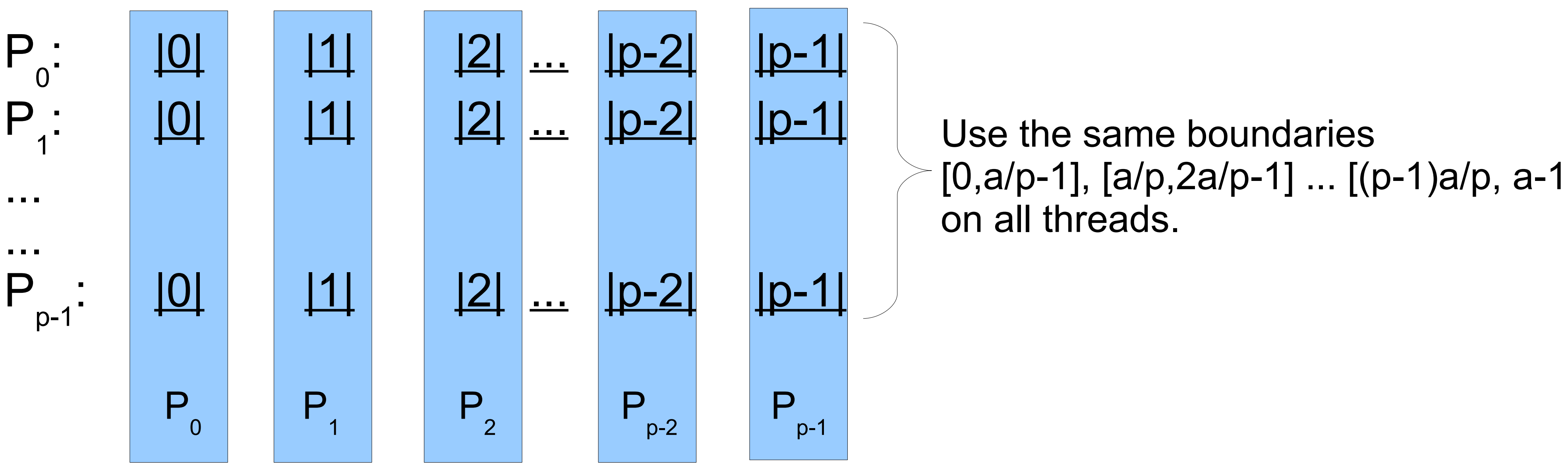
Step 2: Concatenate:
    O(n) (assuming buckets are arrays)
    O(m) if buckets are lists

Total time:
    O(n log(n/m))
    linear if n/m=k is a constant: Choose m=n/k.

Parallel Bucket Sort

Algorithm:

- Divide data evenly between the threads,
- Put data into one of p local buckets.



- Thread P_i sorts the data in all the i'th buckets

Issues:

Which local sorting algorithm to use? How much space is needed for buckets?
Where should each thread store its final result?

Generating random numbers

Pseudorandom numbers in parallel programs:

- Centralized: A dedicated thread generates all pseudorandom numbers
  - May involve many requests to the generating thread
  - Not very suitable for Monte Carlo simulation
- Distributed: Each thread generates the numbers as they are needed
  - Sequences must be distinct
  - Seeds must be distinct
  - Can use a hash function of (time and) process rank

srandom(),random(): Sequential methods for seeding and generating random numbers, all numbers are generated from the same sequence.

srandom_r(), random_r(): Thread safe, generates different sequences on different threads. Slightly more complicated to use.