Sorting continued...

The Good

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]**

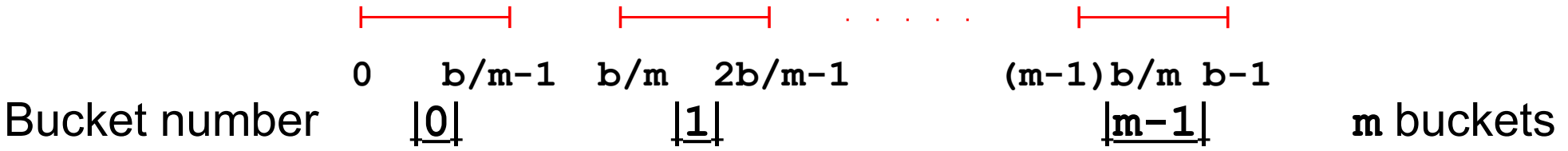
Example (**n=8, b=24**): **a = {9, 15, 3, 0, 6, 21, 18, 12}**

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



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)

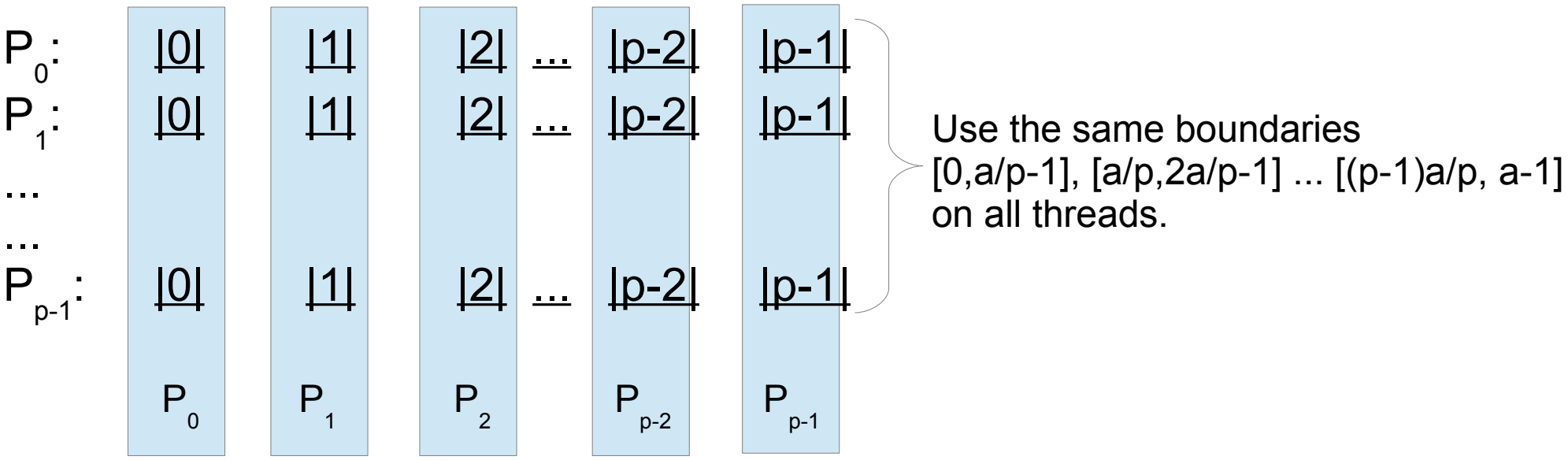
Step 2: Concatenate:
O(n) (assuming buckets are arrays)
O(m) if buckets are lists

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?

Sample Sort

Bucket sort is fast but requires knowledge about distribution

Sample sort tries to mimic the behavior of bucket sort.

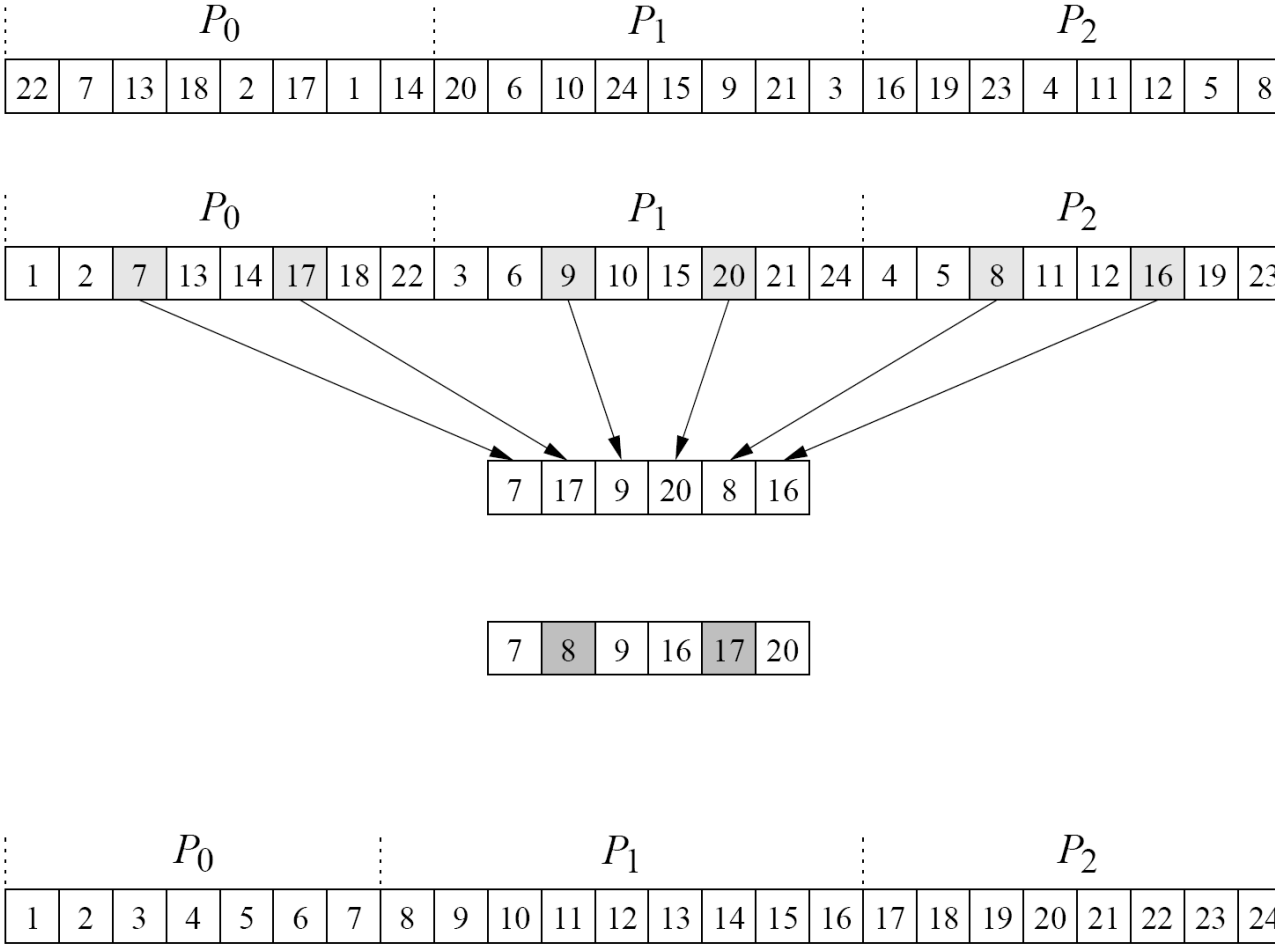
Each processor is responsible for n/p elements:

Sort n/p elements in time O(n/p log(n/p))
Pick p-1 evenly spaced elements ("dividers") from each sorted sequence

Put all p(p-1) dividers in one array
Sort divider array sequentially
Pick every p'th divider as bucket delimiter // No bucket will be > 2n/p
Run parallel bucket sort // Use binary search to locate bucket

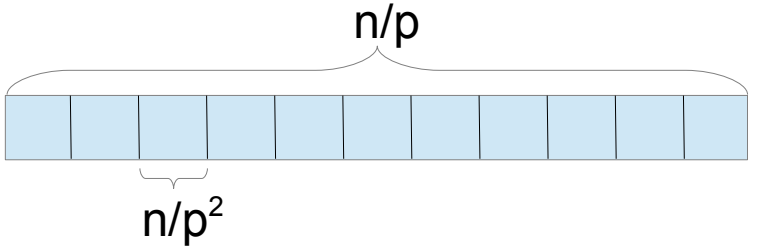
$T_p = O(n/p \log(n/p)) + O(p^2 \log(p)) + O(n/p \log(p))$

Running Sample sort



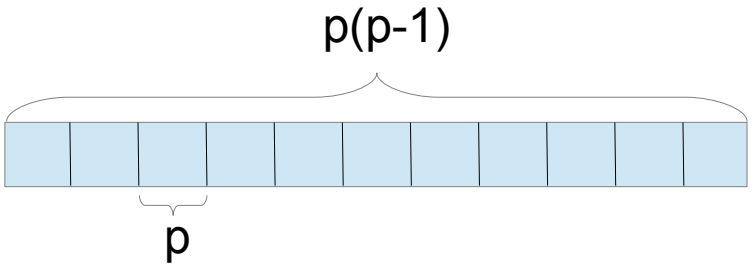
Proof that no bucket will contain more than 2n/p elements

Following the local sorting each thread picks out p-1 evenly spaced dividers from its sorted list creating p equal length intervals.



Each local interval will then contain at most n/p^2 elements.

Following the sequential sorting of all the p(p-1) dividers, p - 1 evenly spaced dividers are chosen from the sorted list. These dividers define the final buckets.

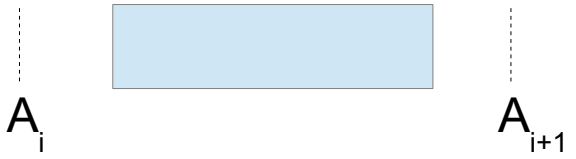


Each bucket will now contain at most p dividers, each one being a divider from the original p sequences of length n/p.

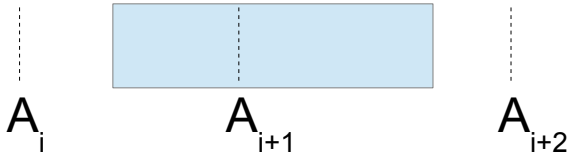
Proof, continued

Consider one such bucket. We divide the original p lists into three groups depending on how many of its dividers are in this bucket. Let A_i denote the dividers from list A.

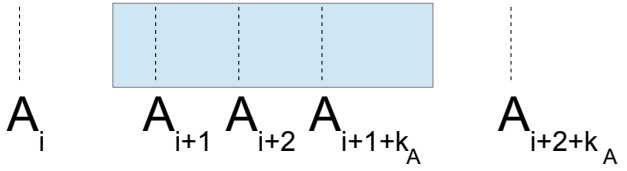
a) 0 dividers in the bucket



b) 1 divider in the bucket



c) $1+k_A$ dividers in the bucket where $k_A > 0$.



Observations:

A can at most contribute n/p^2 elements to the bucket.

A can at most contribute $2n/p^2$ elements to the bucket.

A can at most contribute $2n/p^2 + k_A n/p^2$ elements to the bucket.

Let q be the number of sequences in a), r the number in b), and s the number in c). Then $q + r + s = p$ since every sequence must be in either a), b), or c).

Also, $r + s + \Sigma k_A = p$, since there are p dividers in the bucket.

This shows that $\Sigma k_A = q$.

Proof, completed

The total number of elements that will be placed in the bucket following the bucket sort is then no more than the following expression:

a) b) c)

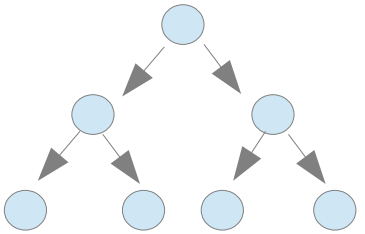
$q^* n/p^2 + r * 2n/p^2 + \Sigma(2n/p^2 + k_A n/p^2) =$

$2n/p^2 (q + r + s) =$ Since $\Sigma 2n/p^2 = 2n/p^2 s$ and $\Sigma k_A = q$

$2n/p$ Since $q + r + s = p$

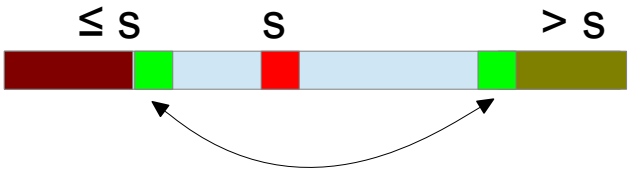
Parallelizing Quicksort

Using p processors on each level gives same asymptotic performance as parallel Mergesort → Not scalable!



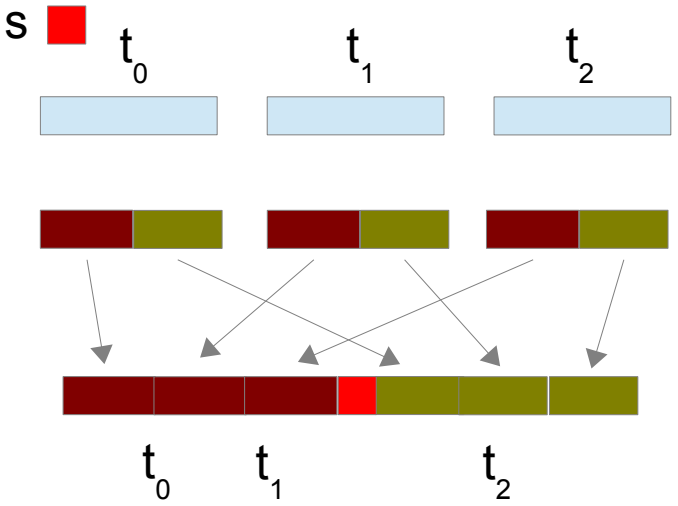
The problem is in the partitioning:

Only one processor involved in partitioning each sequence:



Need a parallel partitioning algorithm:

- Select a common pivot element s
- Each processor performs local partitioning on n/p data elements
- Combine partitioned results to global solution

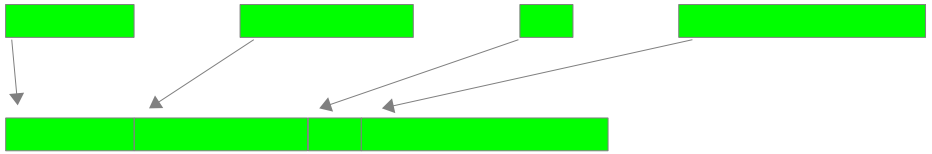


Challenges:

- How can we efficiently combine the partitioned results?
- How do we allocate processes to each remaining sequence?

Prefix Sum

Problem: Packing lists of different length into a common array:



Where in the array should each list start?

General case:
Have numbers $x_0, x_1, x_2, \ldots, x_k$
Wish to compute $s(j) = \sum_{i=0}^{j-1} x_i$ for each j where $0 \le j \le k$.

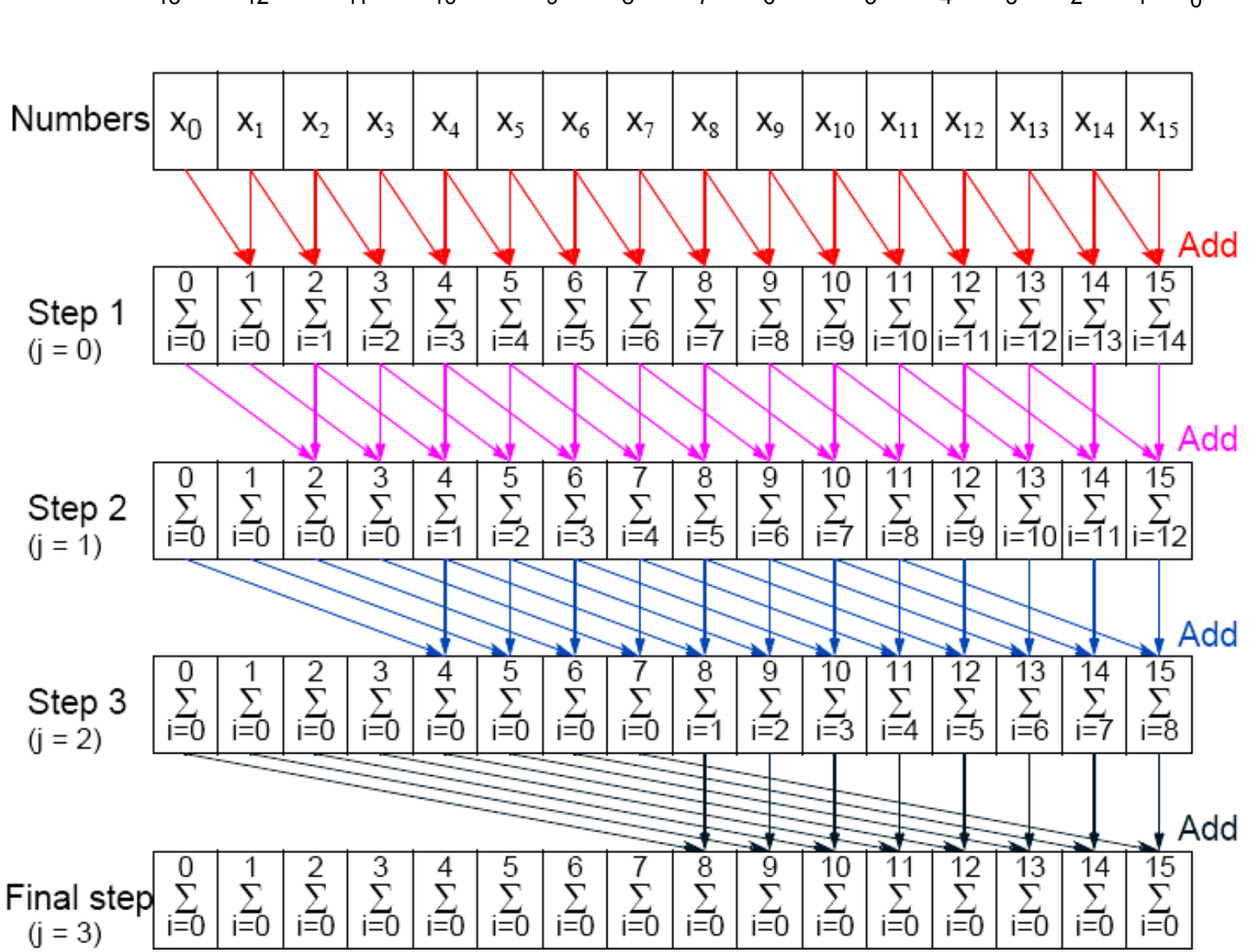
Sequentially:

```
s(0) = 0
for i = 1 to k
    s(i) = s(i-1) + x_{i-1}
```

- Each process p_i computes local value x_i
- One process computes entire prefix sum
- Each process stores values from position s_i

Parallel Prefix Sum

$s(13) = x_{13} + x_{12} + (x_{11} + x_{10}) + (x_9 + x_8 + x_7 + x_6) + (x_5 + x_4 + x_3 + x_2 + x_1 + x_0)$



Implementation of Parallel Prefix Sum

Sequential code:

```
For all i: s(i) = x_i
for (j = 0; j < log(n); j++)        // Loop over steps
   for (i = n-1; i >= 2^j; i--)     // Compute one step
      s(i) = s(i) + s(i - 2^j);
```

How would you do this using:

- Shared memory?
- Distributed memory?

Radix sort

Only works for integers

Sorts by considering the digits of the numbers

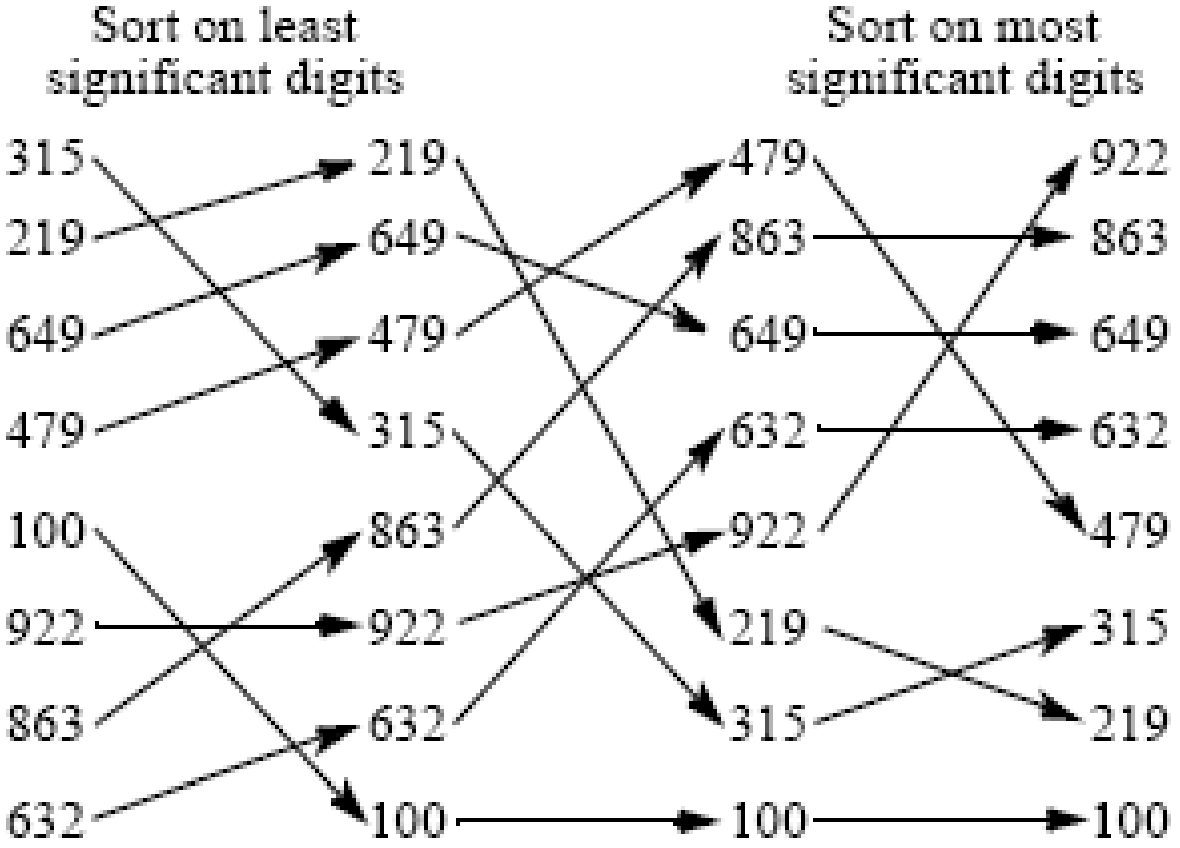
For each digit d_i starting from the least significant one:

Sort the numbers based on d_i using a stable sorting algorithm

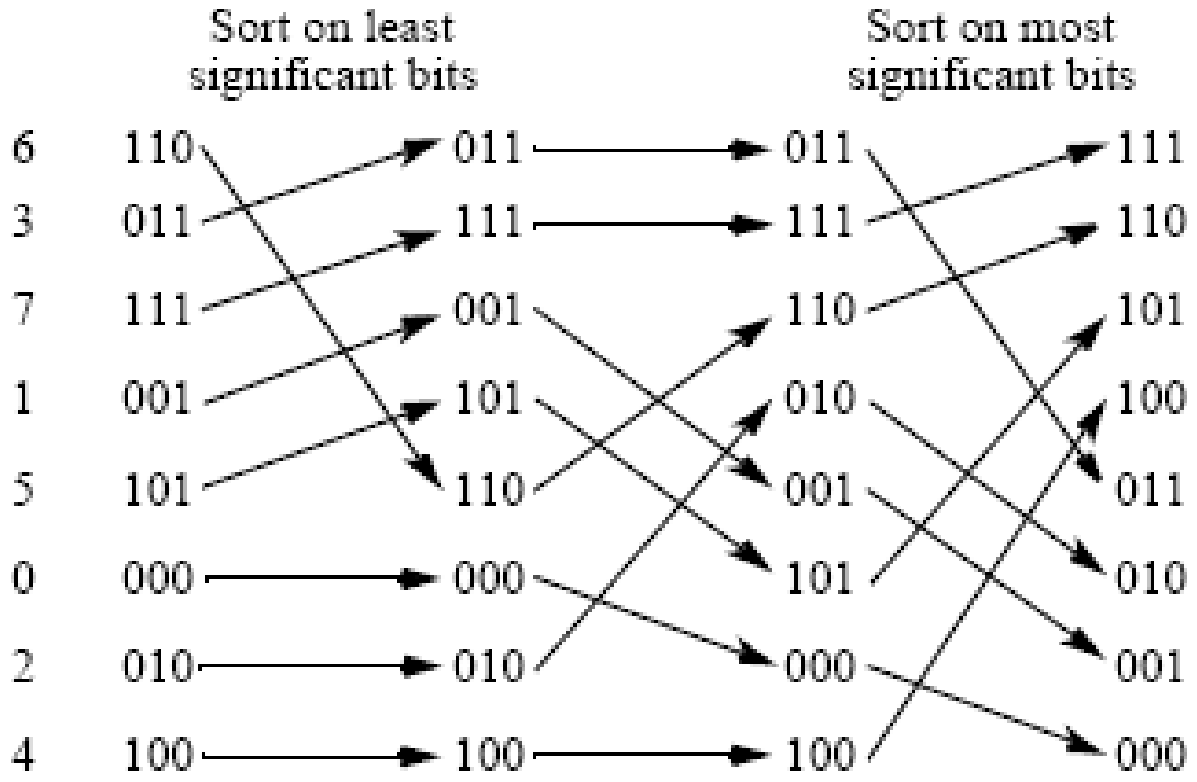
Invariant: After i iterations, the numbers are sorted on the i last digits

Stable sort: if *a*=*b* and *a* occurs before *b* in the input, then *a* should be placed before *b* in the sorted list.

Radix sort using decimal digits

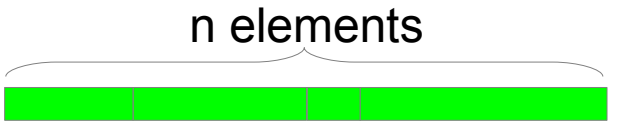


Radix sort using binary digits



Implementing sequential Radix sort

Using an array of length n to hold the b buckets:



One step of the algorithm:

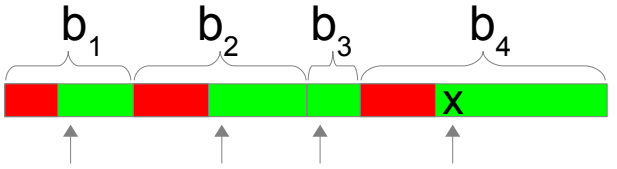
Count number of elements that will go into each bucket: 3 5 1 8

Perform prefix sum on number of elements in each bucket: 0 3 8 9

This gives starting point of each bucket

<u>For each element x</u>

Place x in first empty position of the appropriate bucket



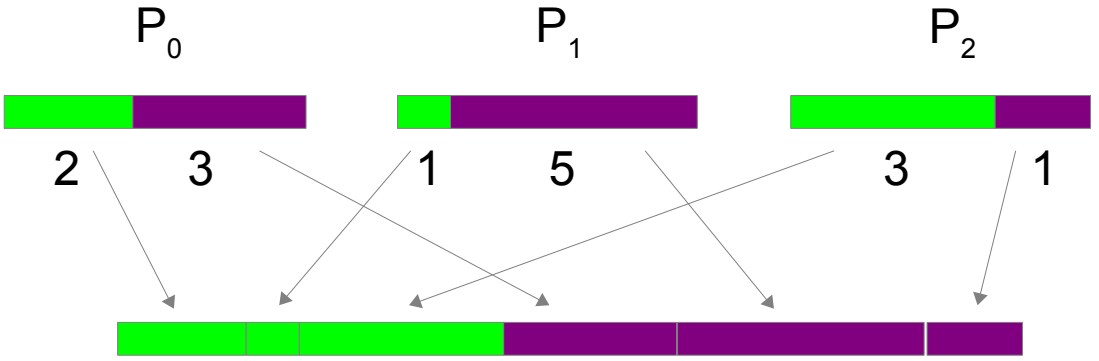
Parallel Radix sort

Procedure for each digit:

1. Divide data among processors in chunks of n/p elements (static partitioning)
2. Using current digit:

   Each processor counts number of elements in each of b local bins

   (b could be 2, 10, 256 etc)
3. Perform (parallel?) prefix sum on each bucket size and calculate starting points in global list
4. Move elements into bins



b_0 b_1

(2 3 6) (3 8 9)