Theoretically-Efficient and Practical Parallel In-Place Radix Sorting

Authors: Omar Obeya, Endrias Kahssay, Edward Fan, Julian Shun

Why Radix Sort?

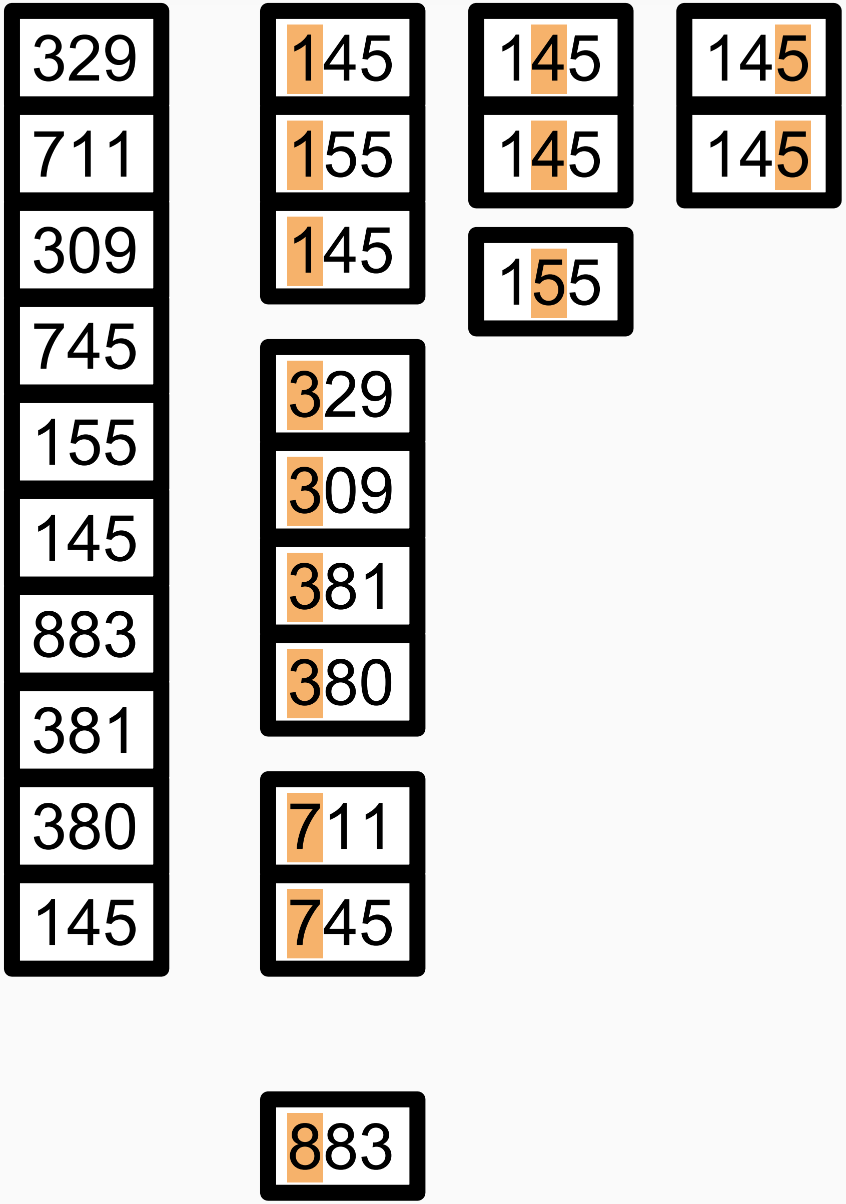
Takes O(n) work for fixed length integers.

Comparison-based sorts take $\Omega(n \log(n))$ work.

(Most Significant Digit First) Radix Sort

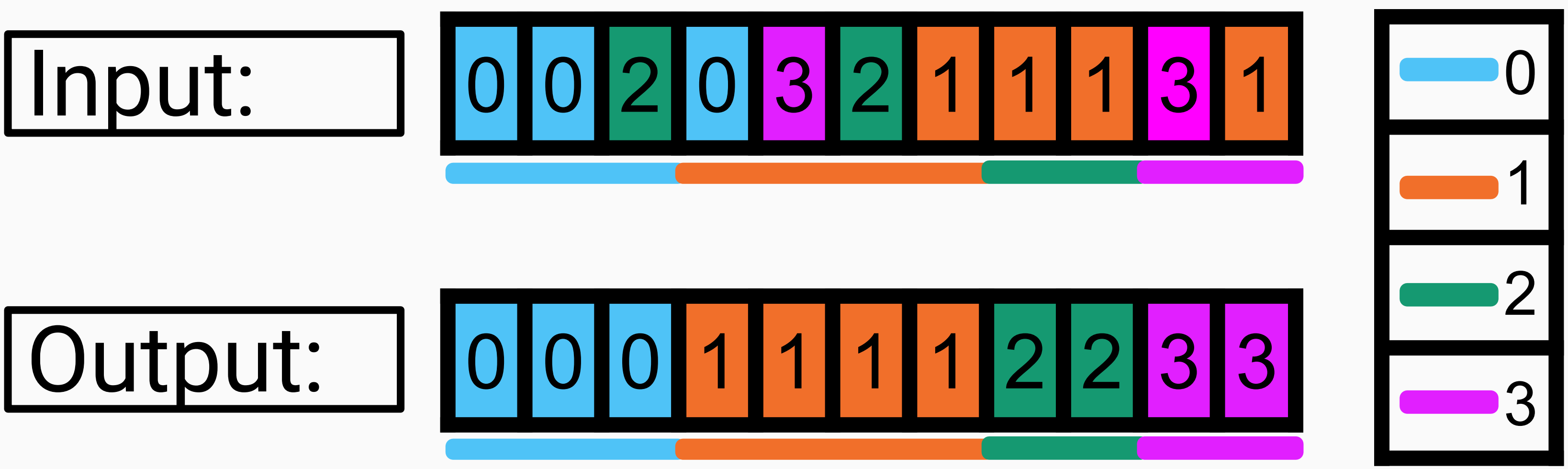
Radix Sort

- Sort elements according to one digit at a time.
- Most significant digit to least significant digit.
- Recurse on elements with equal digits.



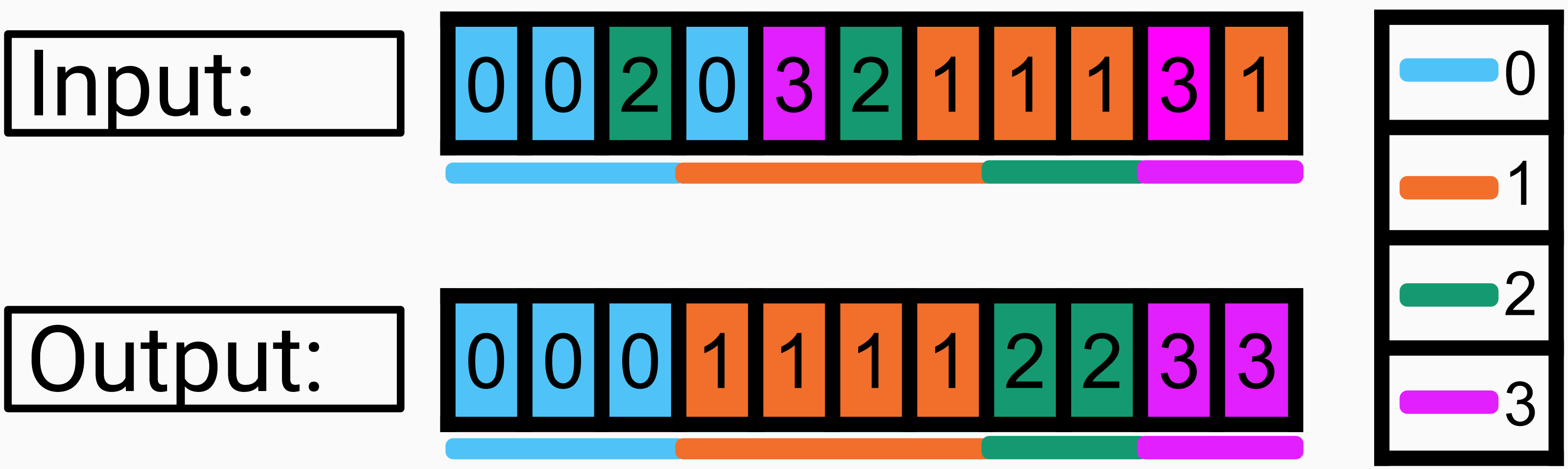
Terminology: Country

Country: sub-array that will include elements belonging to the same bucket after sorting.



Radix Sort: Subproblem

Sort elements according to digits such that each element is in the correct country.

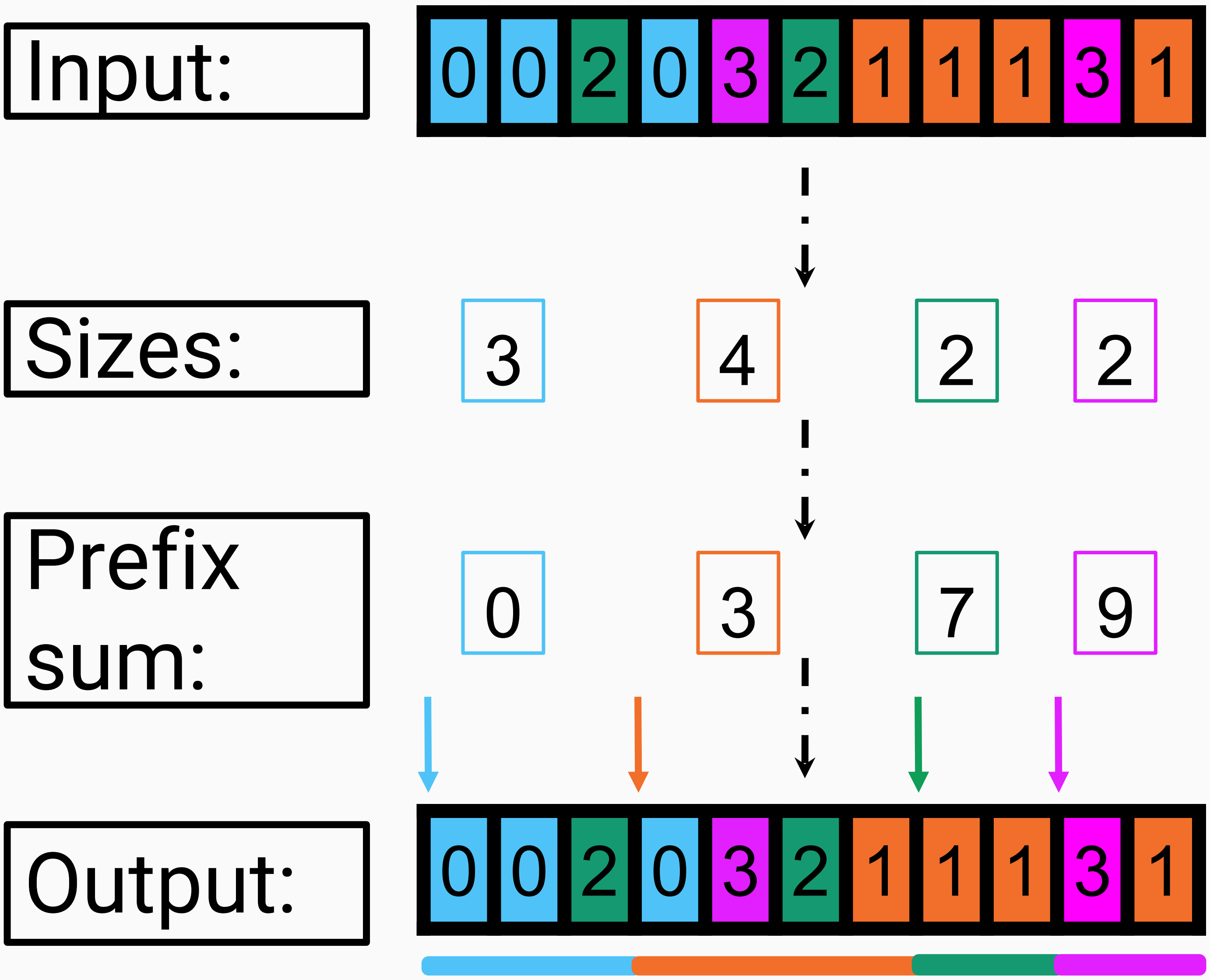


Serial In-place Radix Sort

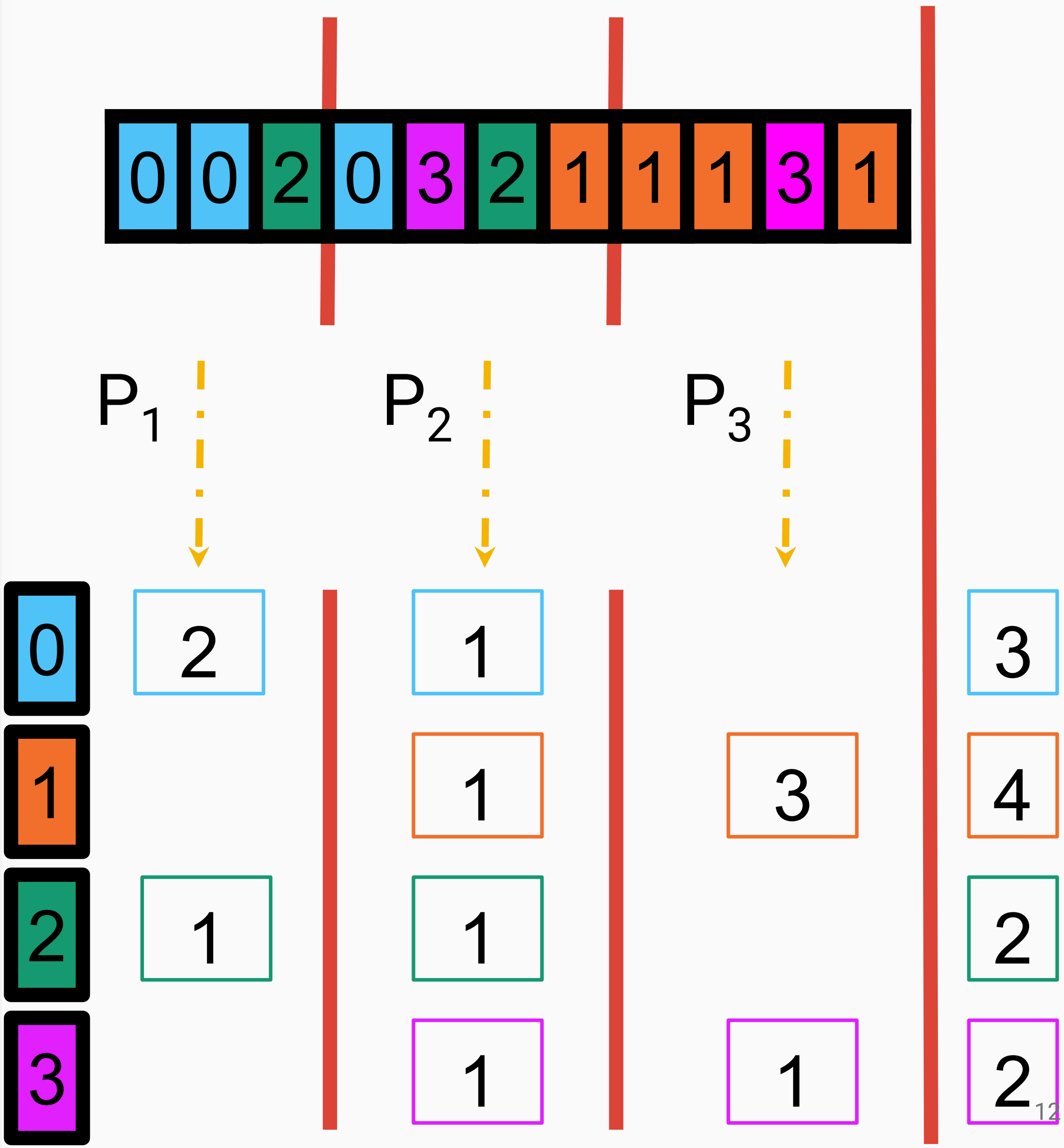
1. Find start location of each country (Histogram Building).

2. Move items to the correct country in-place.

Histogram Building



Parallel Histogram Building



Serial In-place Radix Sort

Initialize pointer to beginning of each country

For each country:

While (pointer not at end of country) {

While(item pointed to is not in correct country) {

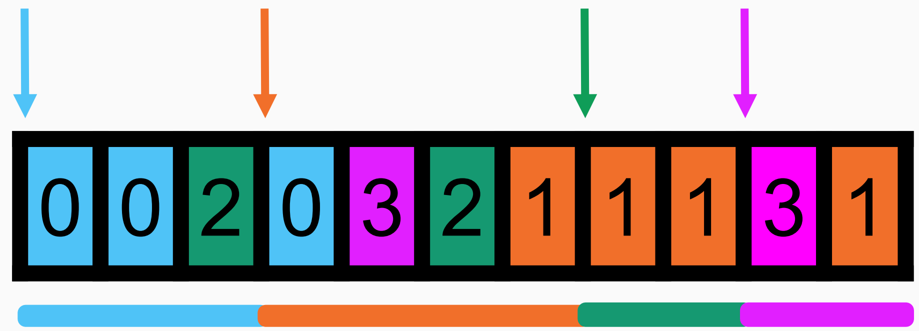
Swap item to location pointed to in target country

Increment target country pointer

}

Increment current country pointer

}



Serial In-place Radix Sort

Initialize pointer to beginning of each country

For each country:

While (pointer not at end of country) {

While(item pointed to is not in correct country) {

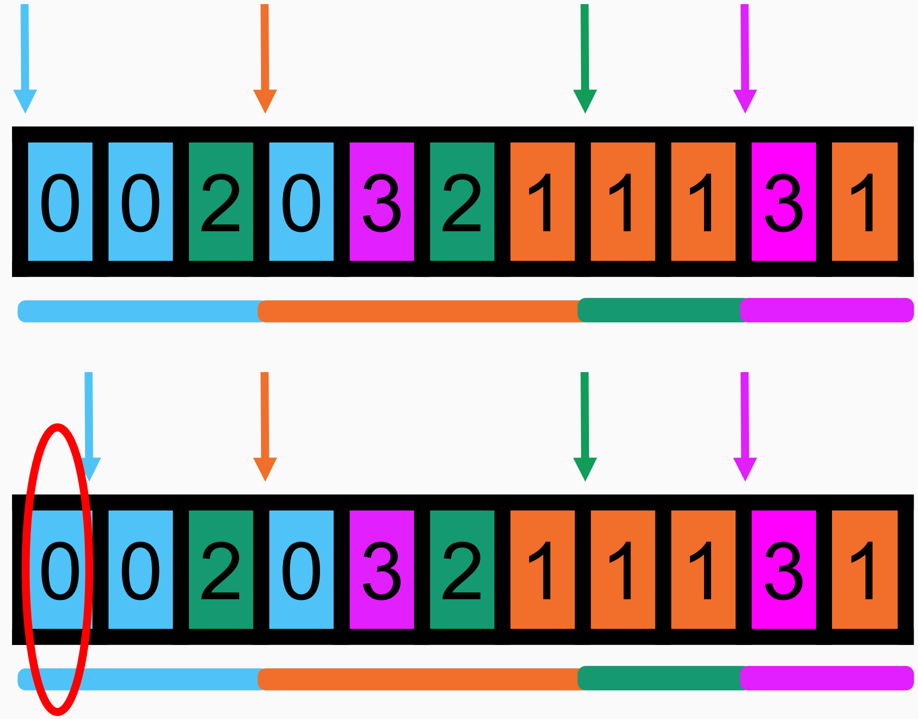
Swap item to location pointed to in target country

Increment target country pointer

}

Increment current country pointer

}



Serial In-place Radix Sort

Initialize pointer to beginning of each country

For each country:

While (pointer not at end of country) {

While(item pointed to is not in correct country) {

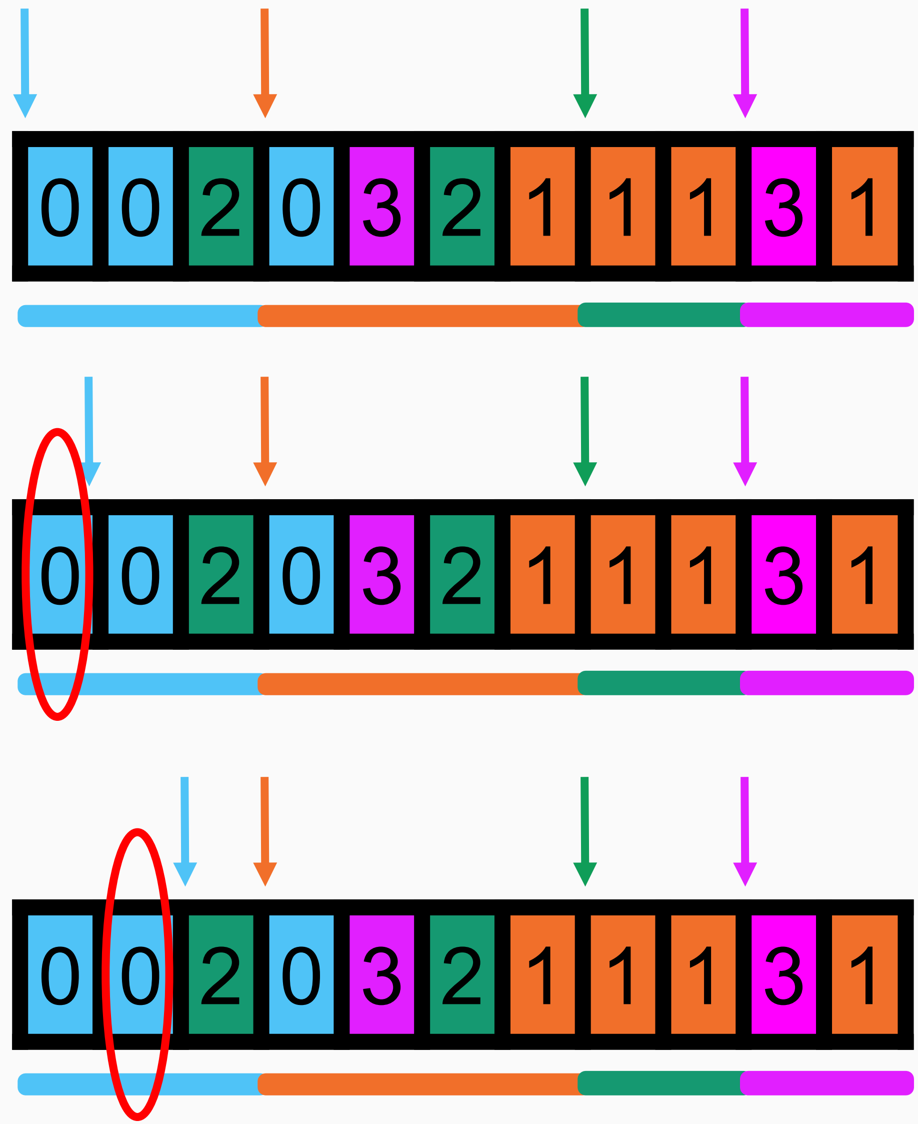
Swap item to location pointed to in target country

Increment target country pointer

}

Increment current country pointer

}



Serial In-place Radix Sort

Initialize pointer to beginning of each country

For each country:

While (pointer not at end of country) {

While(item pointed to is not in correct country) {

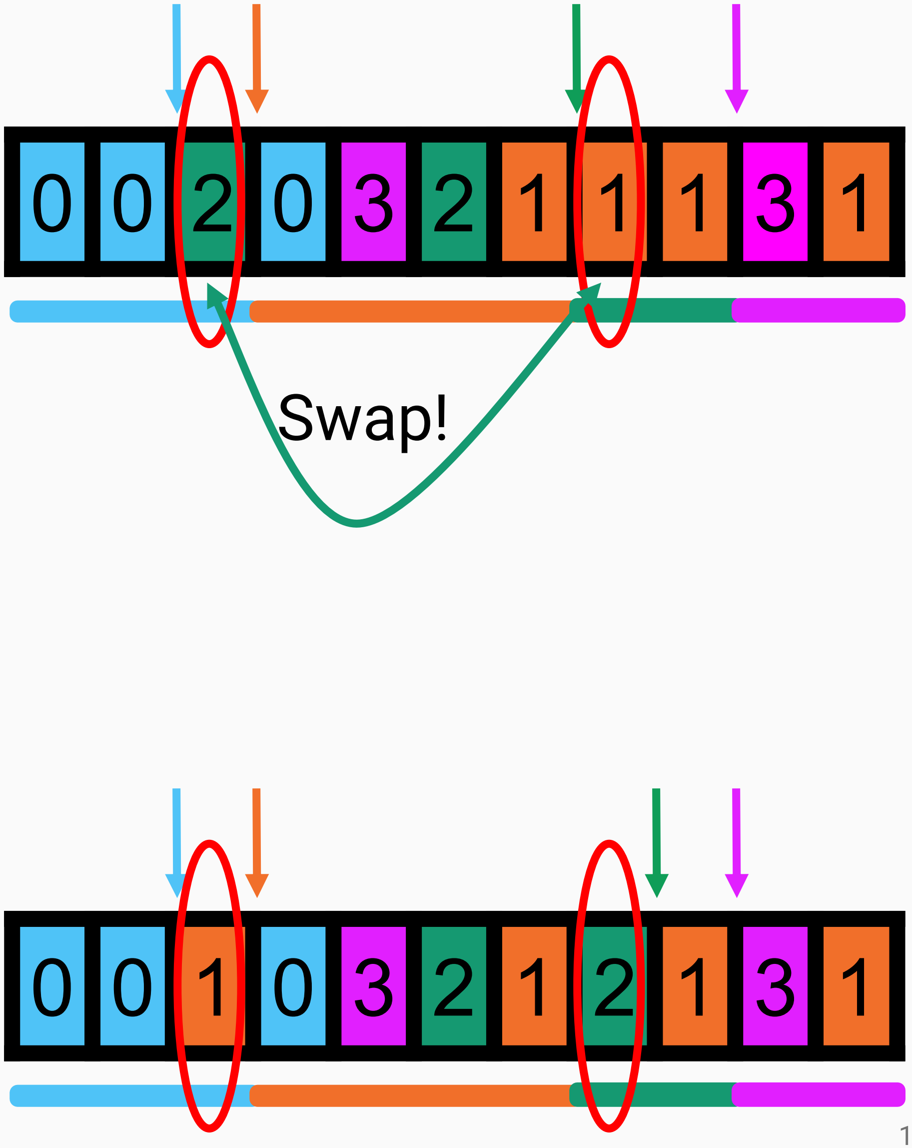
Swap item to location pointed to in target country

Increment target country pointer

}

Increment current country pointer

}



Serial In-place Radix Sort

Initialize pointer to beginning of each country

For each country:

While (pointer not at end of country) {

While(item pointed to is not in correct country) {

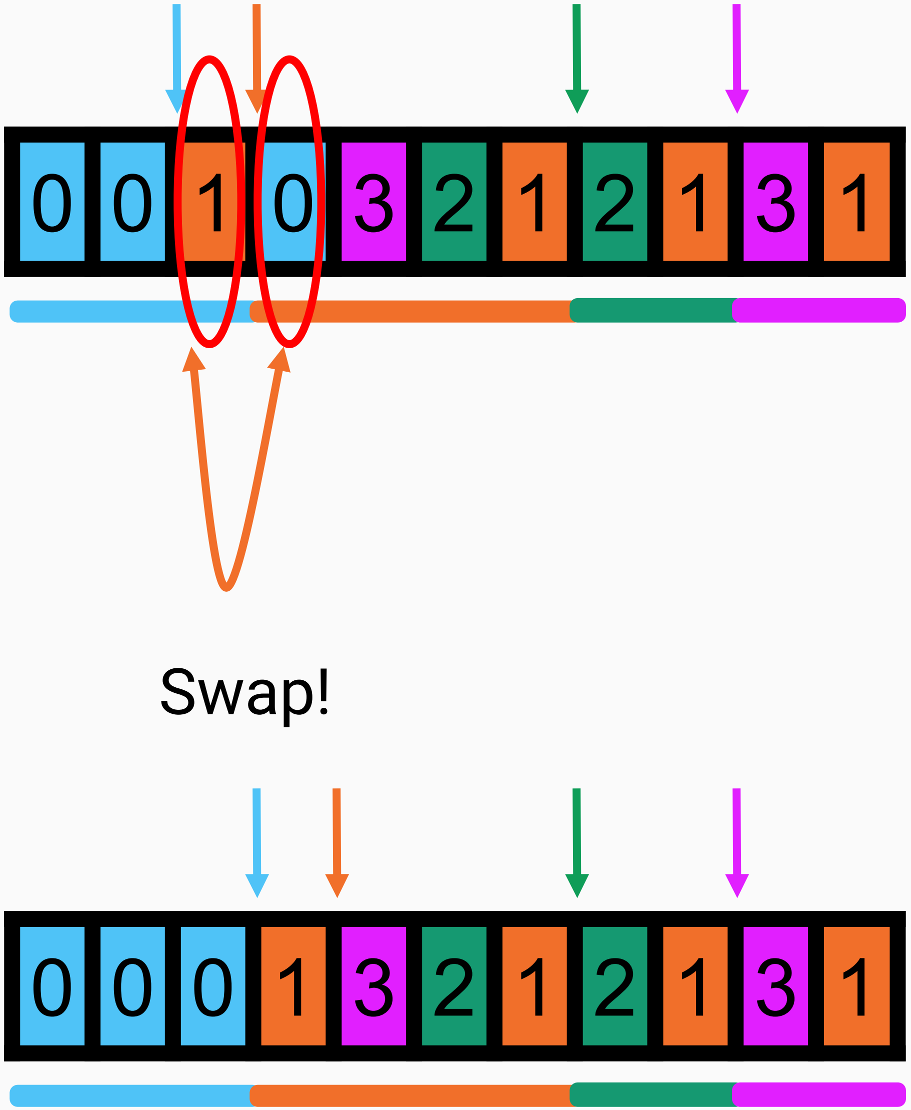
Swap item to location pointed to in target country

Increment target country pointer

}

Increment current country pointer

}



Serial In-place Radix Sort

Initialize pointer to beginning of each country

For each country:

While (pointer not at end of country) {

While(item pointed to is not in correct country) {

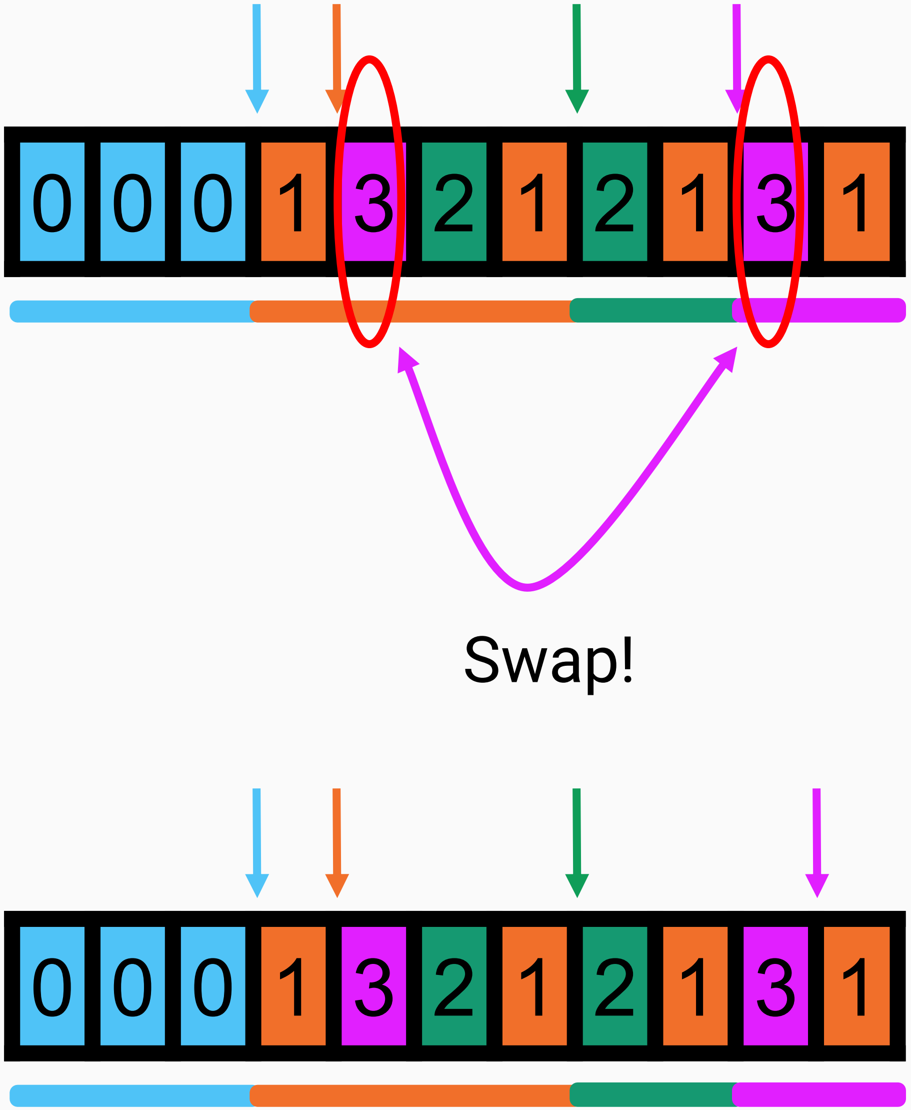
Swap item to location pointed to in target country

Increment target country pointer

}

Increment current country pointer

}



Serial In-place Radix Sort

Initialize pointer to beginning of each country

For each country:

While (pointer not at end of country) {

While(item pointed to is not in correct country) {

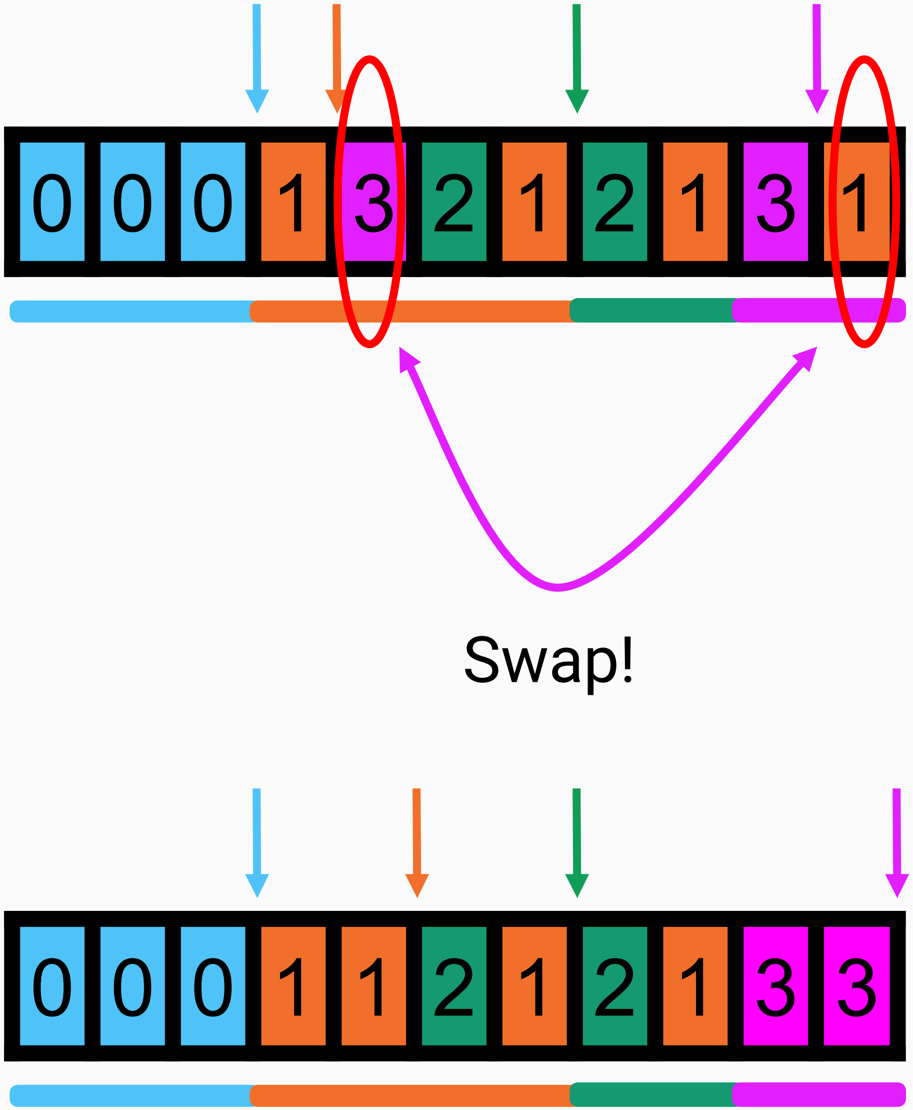
Swap item to location pointed to in target country

Increment target country pointer

}

Increment current country pointer

}



Serial In-place Radix Sort

Initialize pointer to beginning of each country

For each country:

While (pointer not at end of country) {

While(item pointed to is not in correct country) {

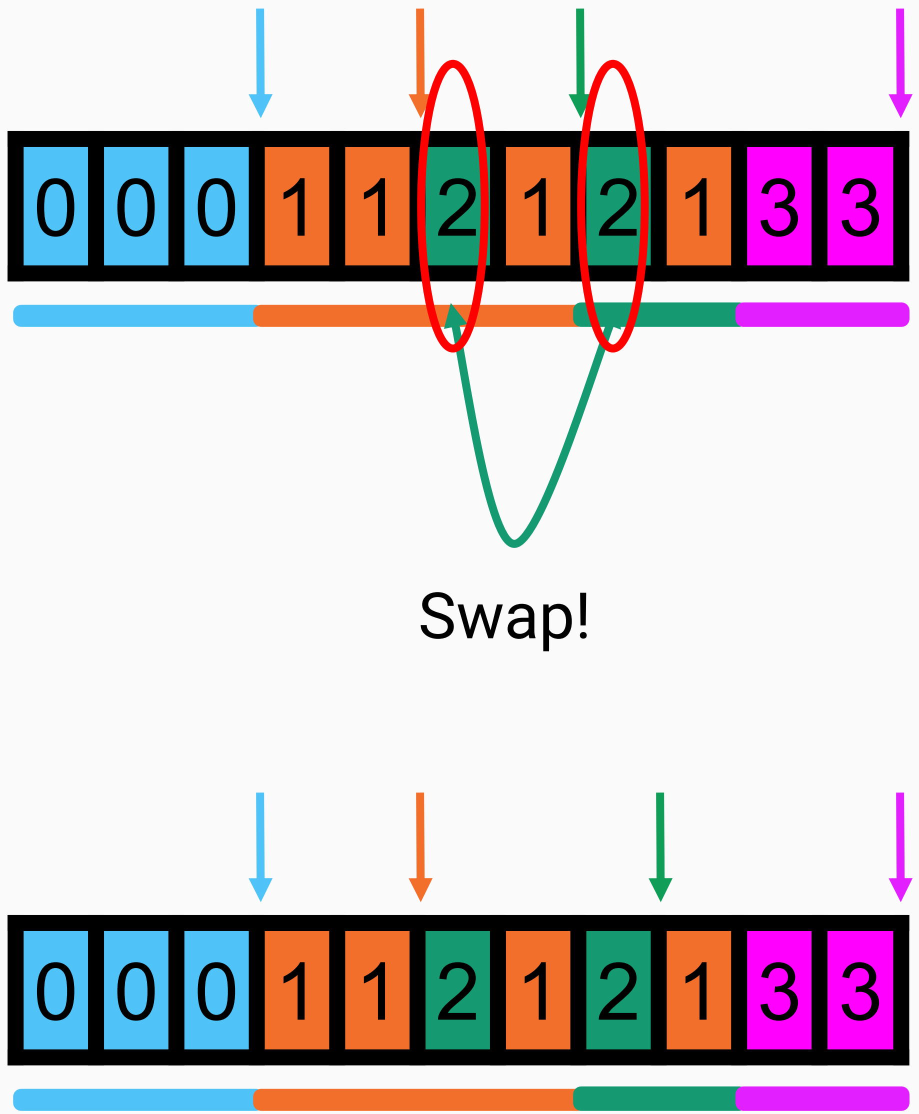
Swap item to location pointed to in target country

Increment target country pointer

}

Increment current country pointer

}



Serial In-place Radix Sort

Initialize pointer to beginning of each country

For each country:

While (pointer not at end of country) {

While(item pointed to is not in correct country) {

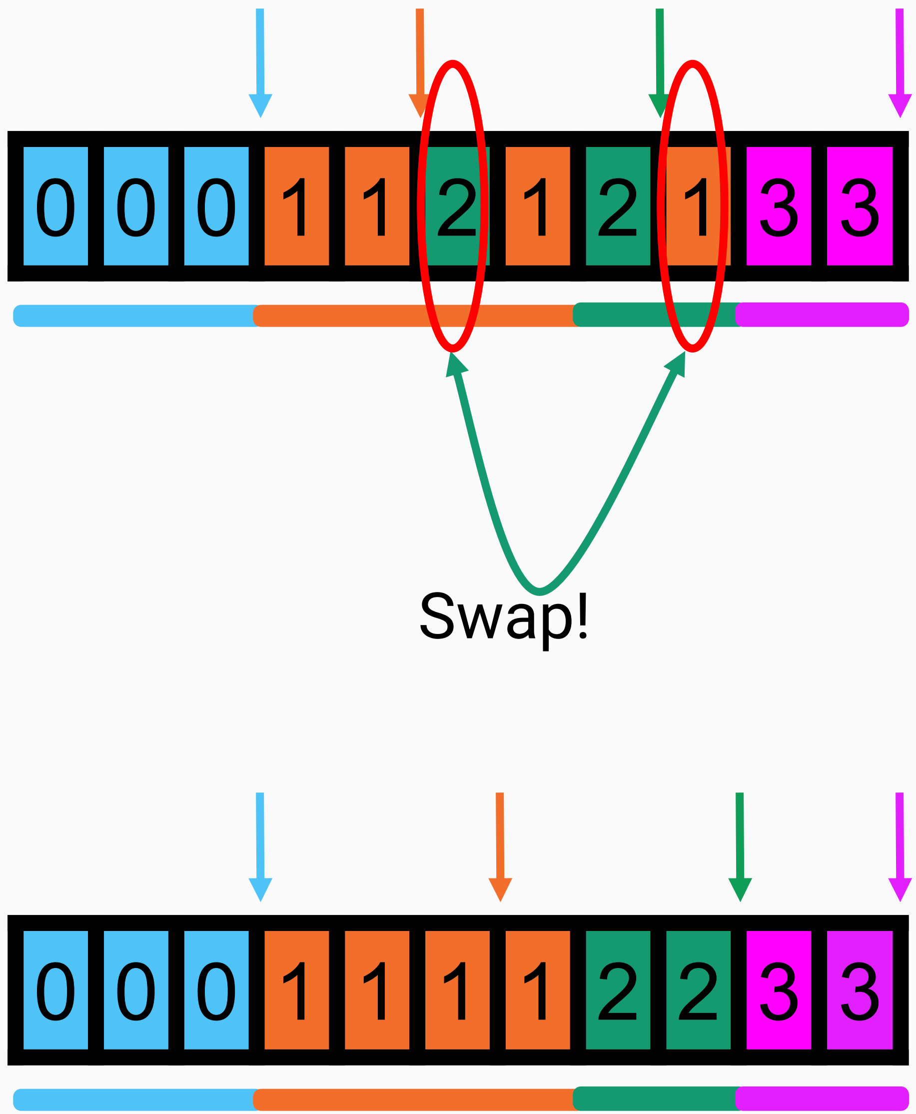
Swap item to location pointed to in target country

Increment target country pointer

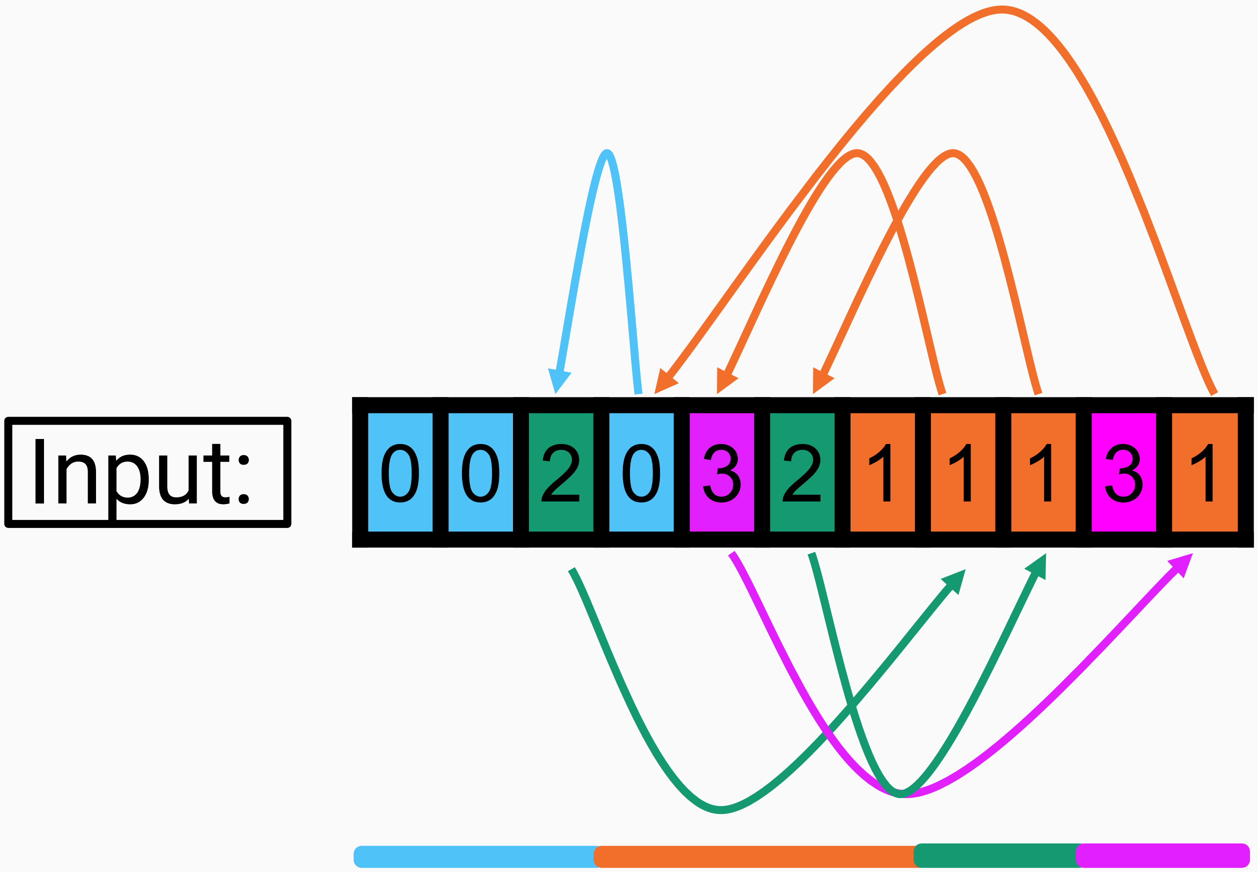
}

Increment current country pointer

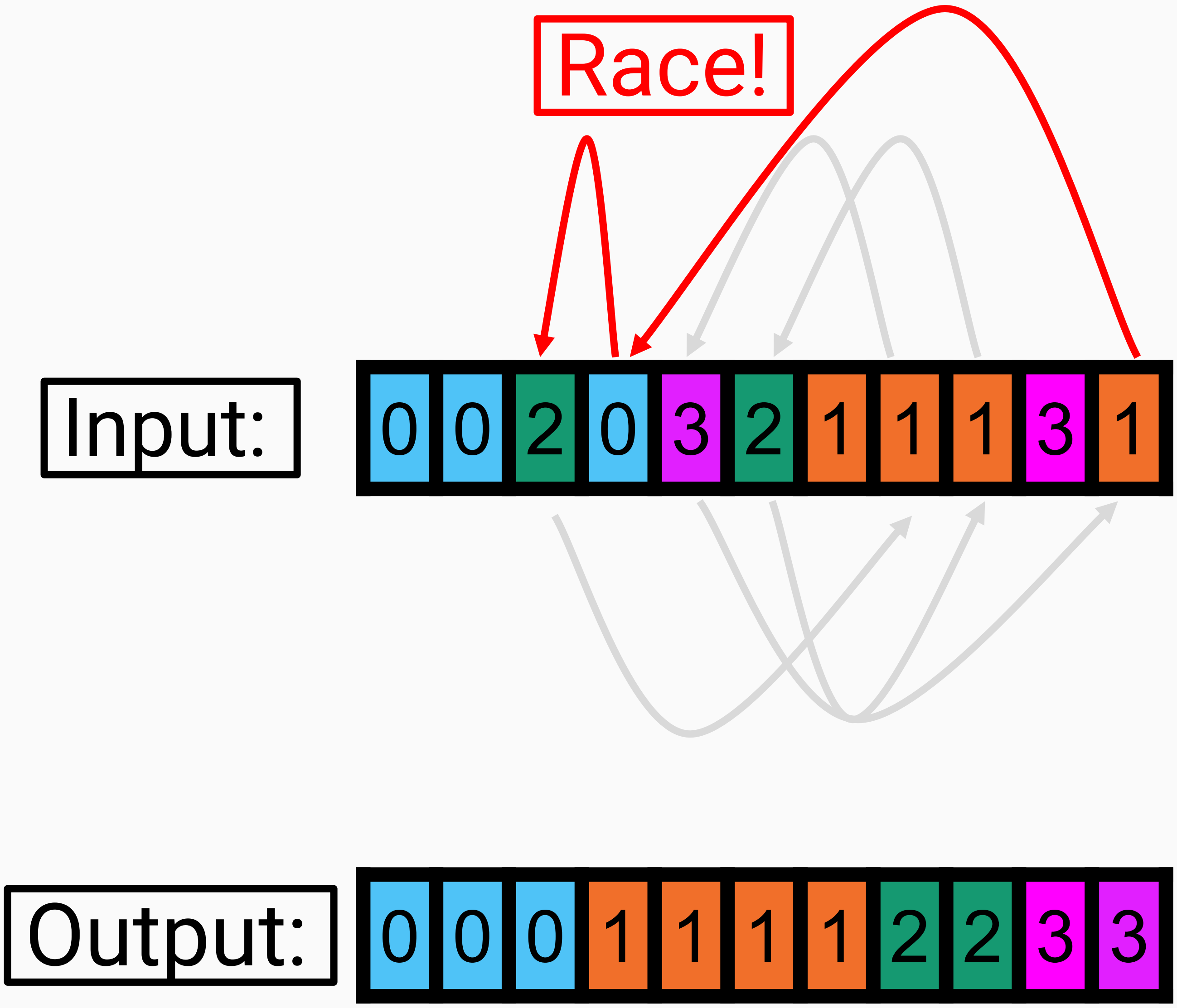
}



Why parallel in-place is hard?!



Why parallel in-place is hard?!



Related Work

PARADIS [Cho et. al 2015]

- Parallel in-place radix sort.
- Worst case span is O(n).

IPS4o [Axtmann et. al 2017]

- Parallel in-place comparison based sort.
- Work is O(nlog(n)).

Goal

A relaxed PIP algorithm for radix sort

For some parameter K:

a. Work: $O(n)$
b. Span: $O(\log(K) + n/K)$
c. Space: $O(K)$

(assuming fixed length integers)

Our Algorithm: Regions Sort

Regions Sort Overview

1. Local Sorting
   - Partially sort the input.

2. Regions Graph Building
   - Represent dependencies in partially sorted array with small amount of memory.

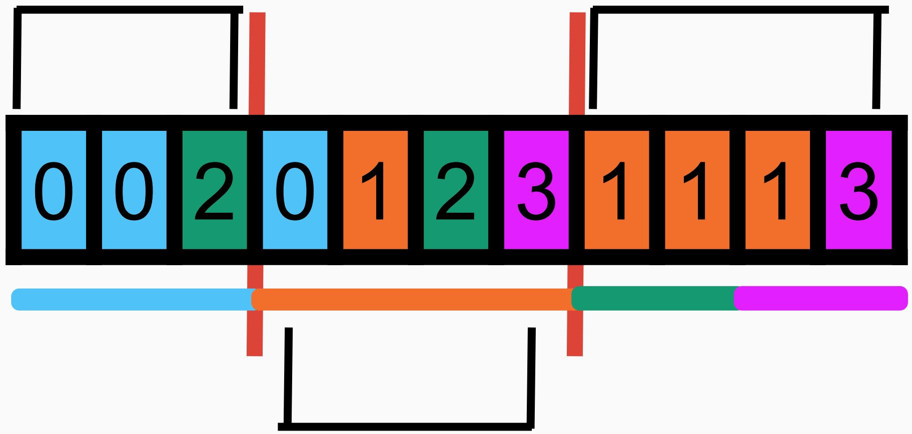
3. Global Sorting
   - Use regions graph to completely sort the input.

Local Sorting

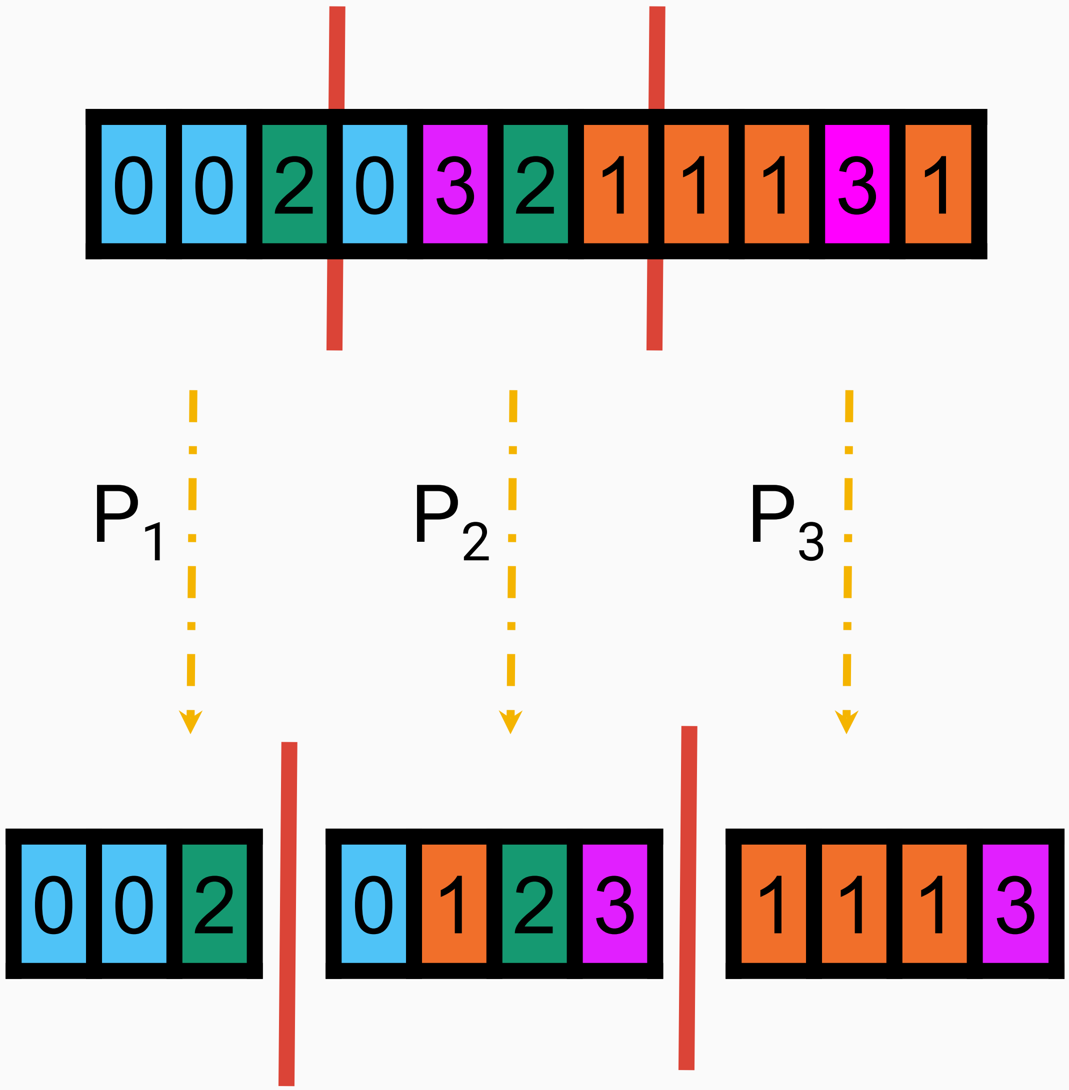
Key Idea:

Divide array into K *Blocks* and sort each block independently.

Block: sub-array of size n/K.



Local Sorting



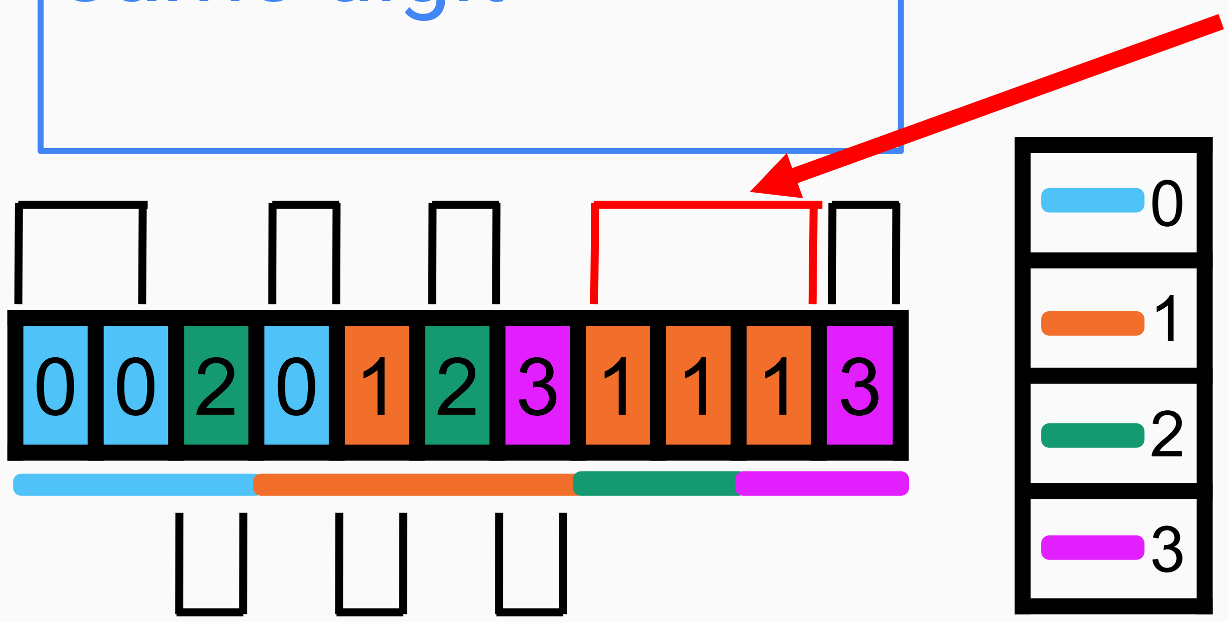
Sort using serial in-place radix sort

Regions Graph Building

Key Idea: Represent dependences in partially sorted array with small amount of memory.

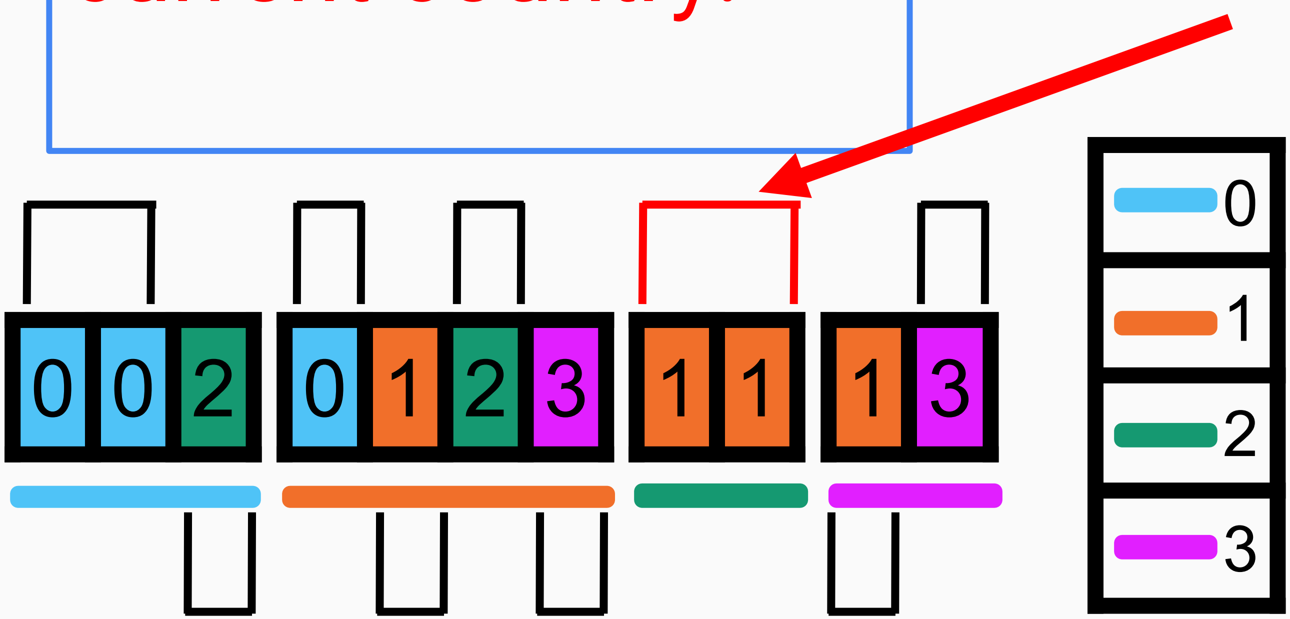
Regions Graph Building

Homogeneous sub-array: A subarray with the same digit

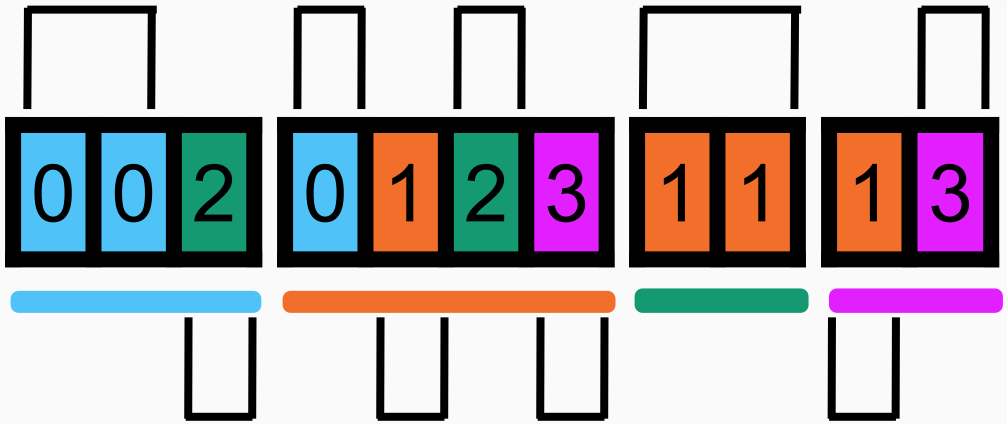


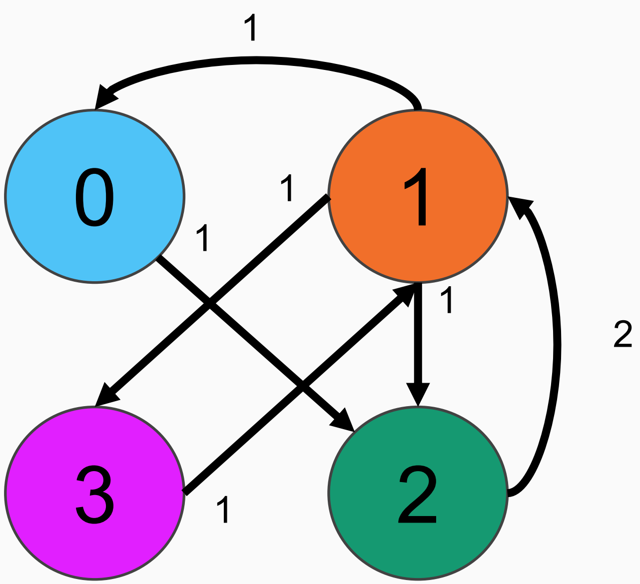
Regions Graph Building

Region: A homogeneous sub-array within same current country.



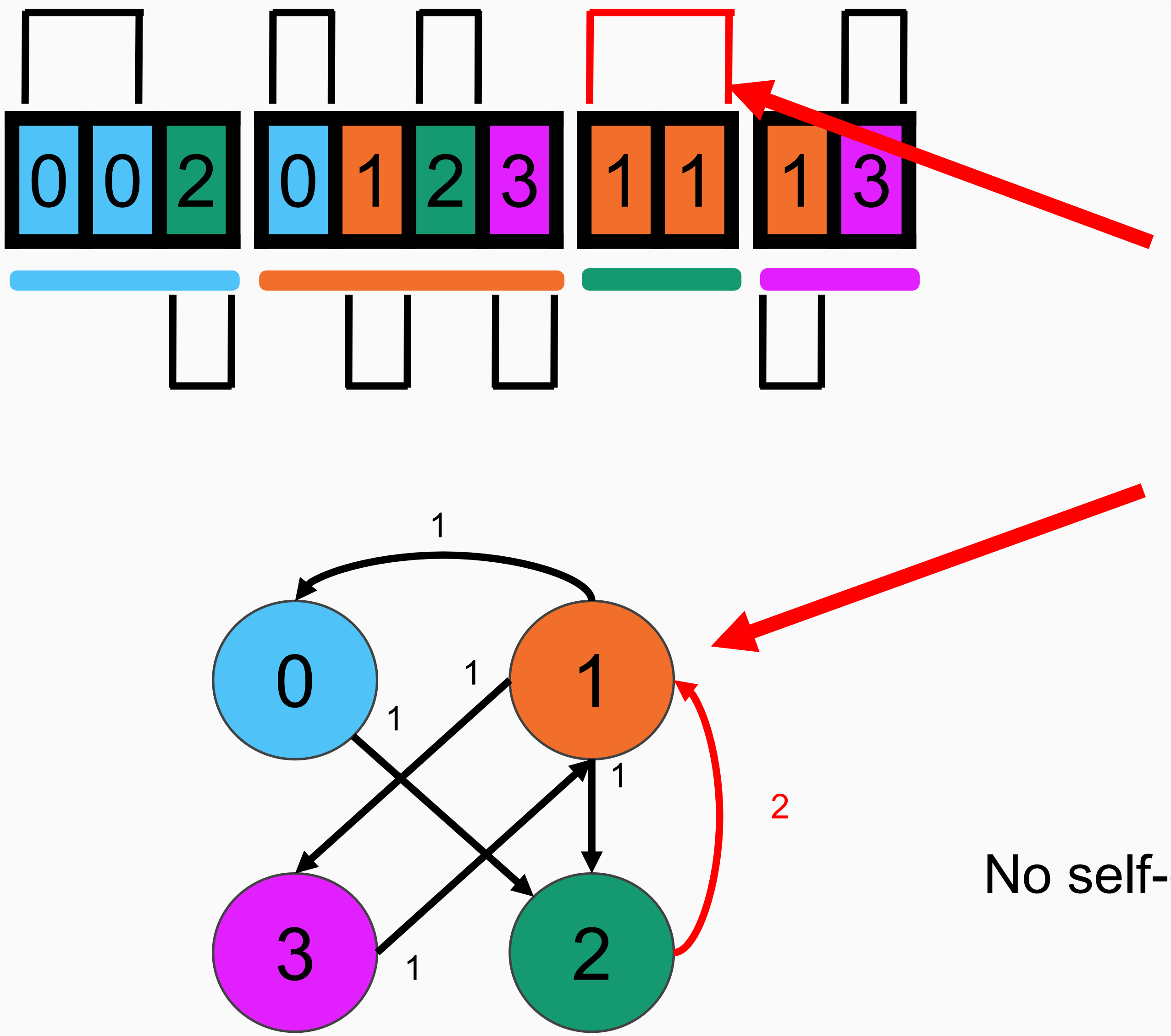
Regions Graph Building





Create edge of weight W from country x to country y if a region of W elements wants to go from country x to country y

Regions Graph Building



No self-edges

Global Sorting

Key Idea: Use regions graph to move regions to their target countries iteratively and updating the graph.

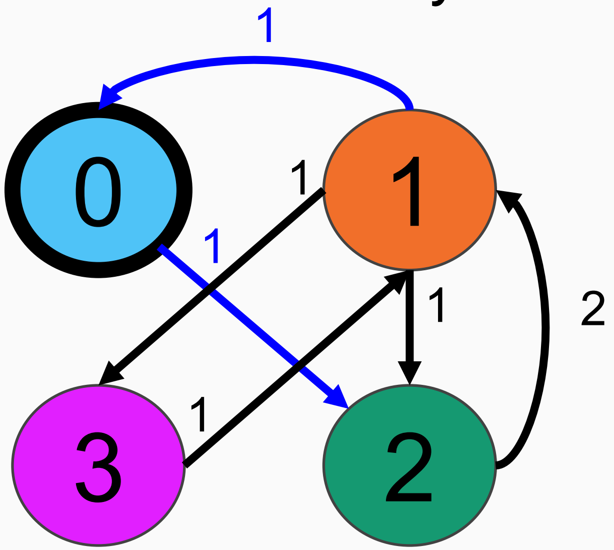
<u>Two Approaches:</u>

1. Cycle Finding
2. 2-Path Finding

Global Sorting

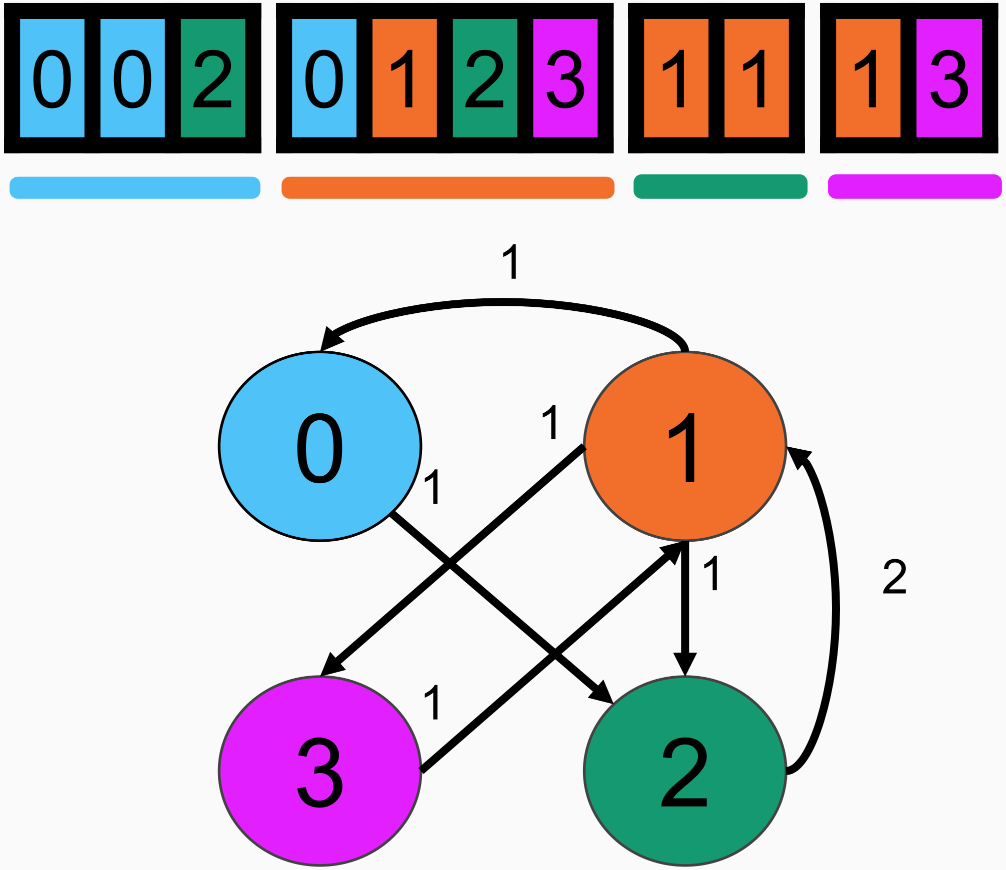
A 2-path consists of two edges:

- Incoming edge to node x corresponding to a region that can be moved into country x.
- Outgoing edge from node x corresponding to a region that is in country x and needs to be moved out of country x.



Global Sorting: 2-Path Finding

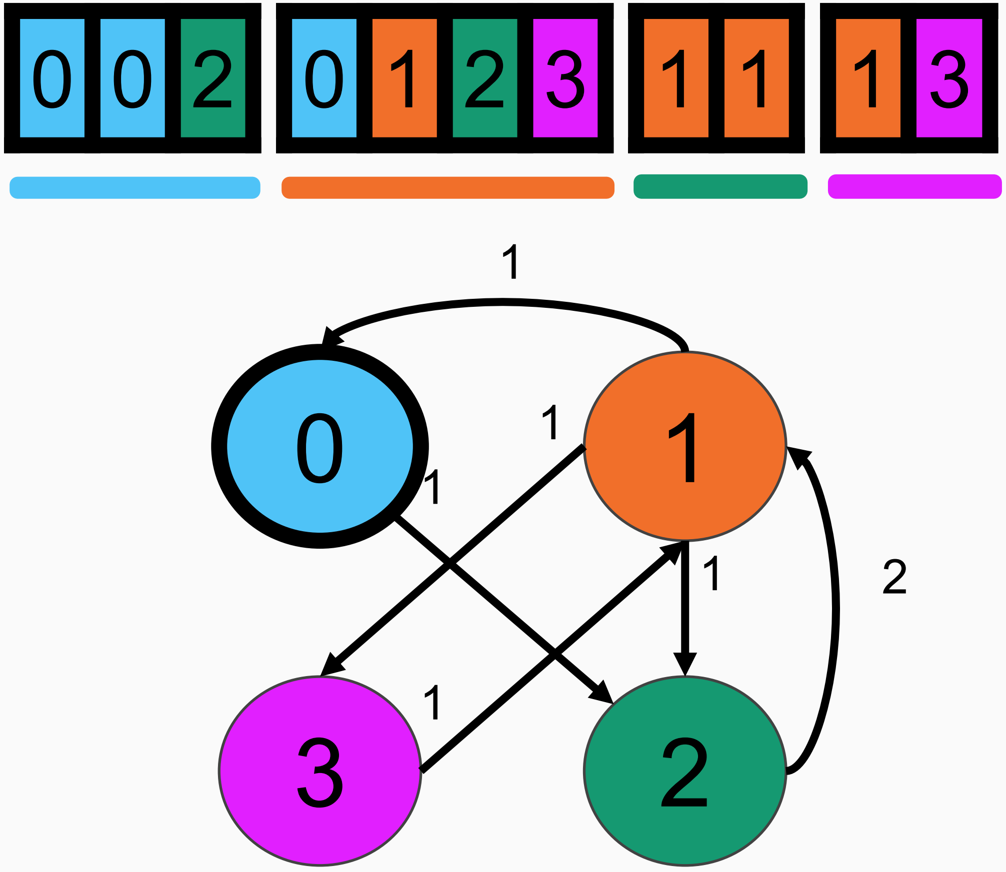
2-path Finding



Global Sorting: 2-Path Finding

2-path Finding

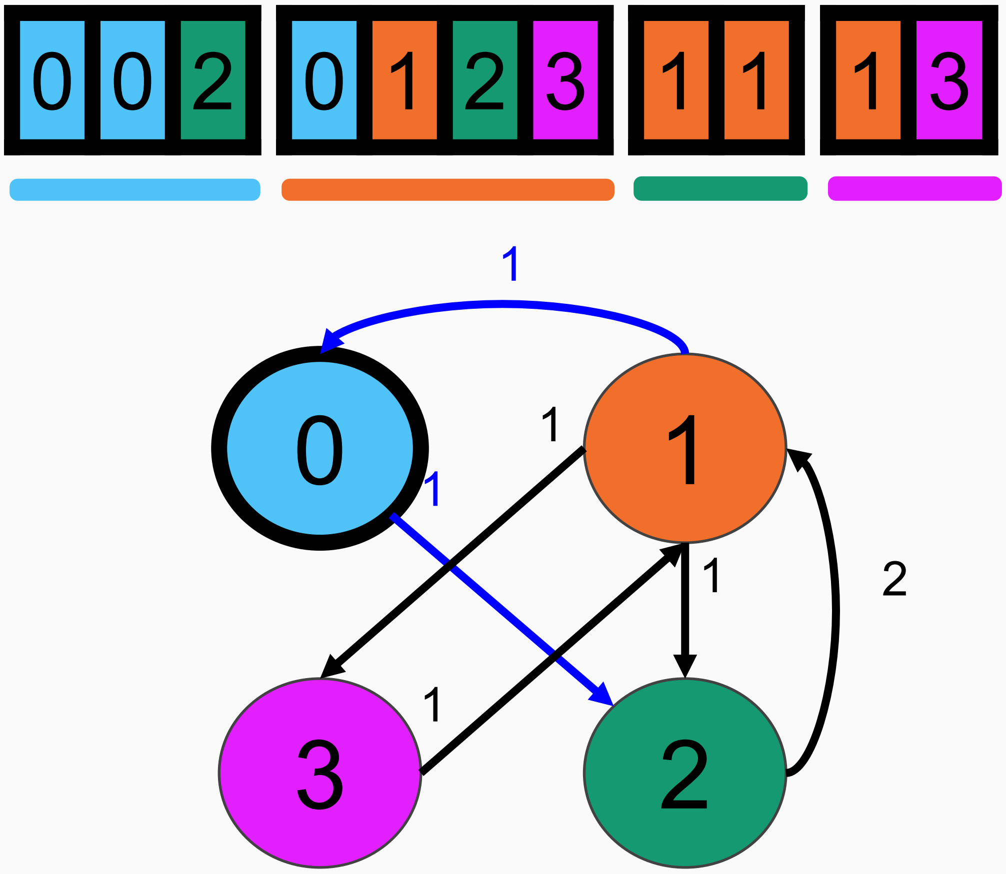
1. Choose a vertex.



Global Sorting: 2-Path Finding

2-path Finding

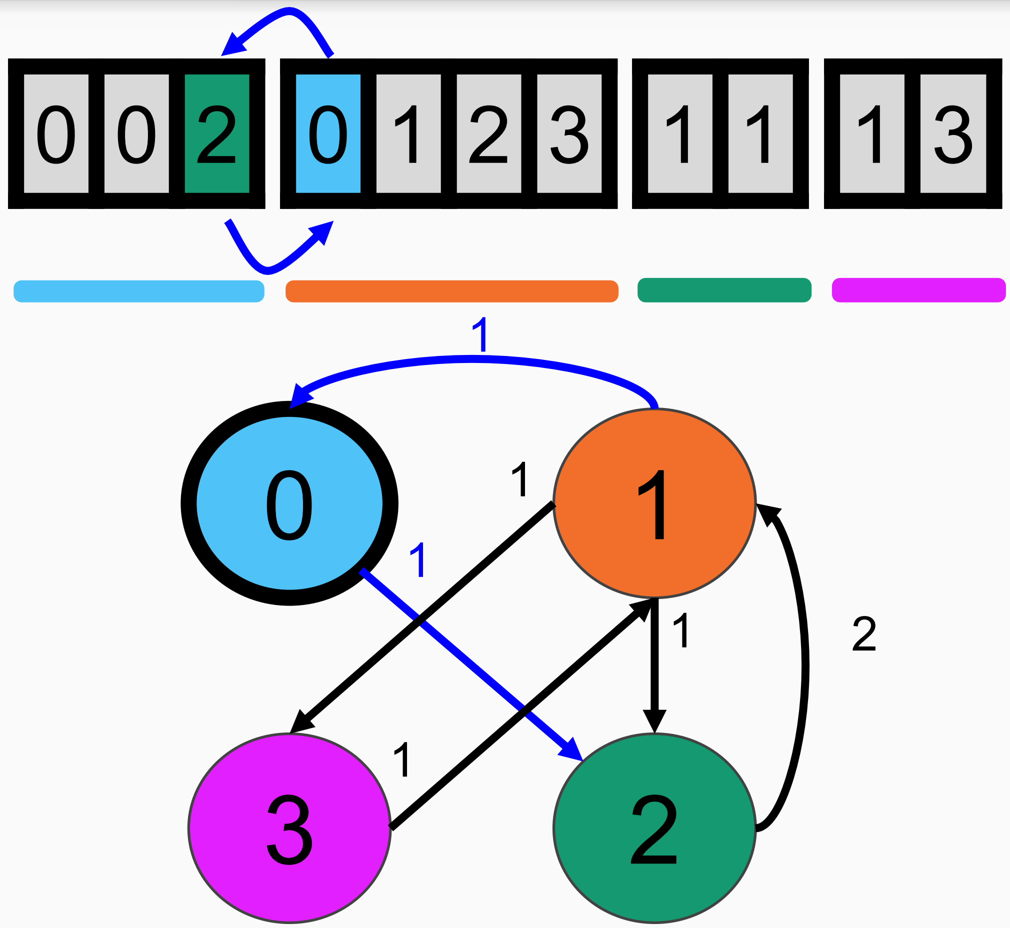
1. Choose a vertex.
2. Match incoming edges with outgoing edges.



Global Sorting: 2-Path Finding

2-path Finding

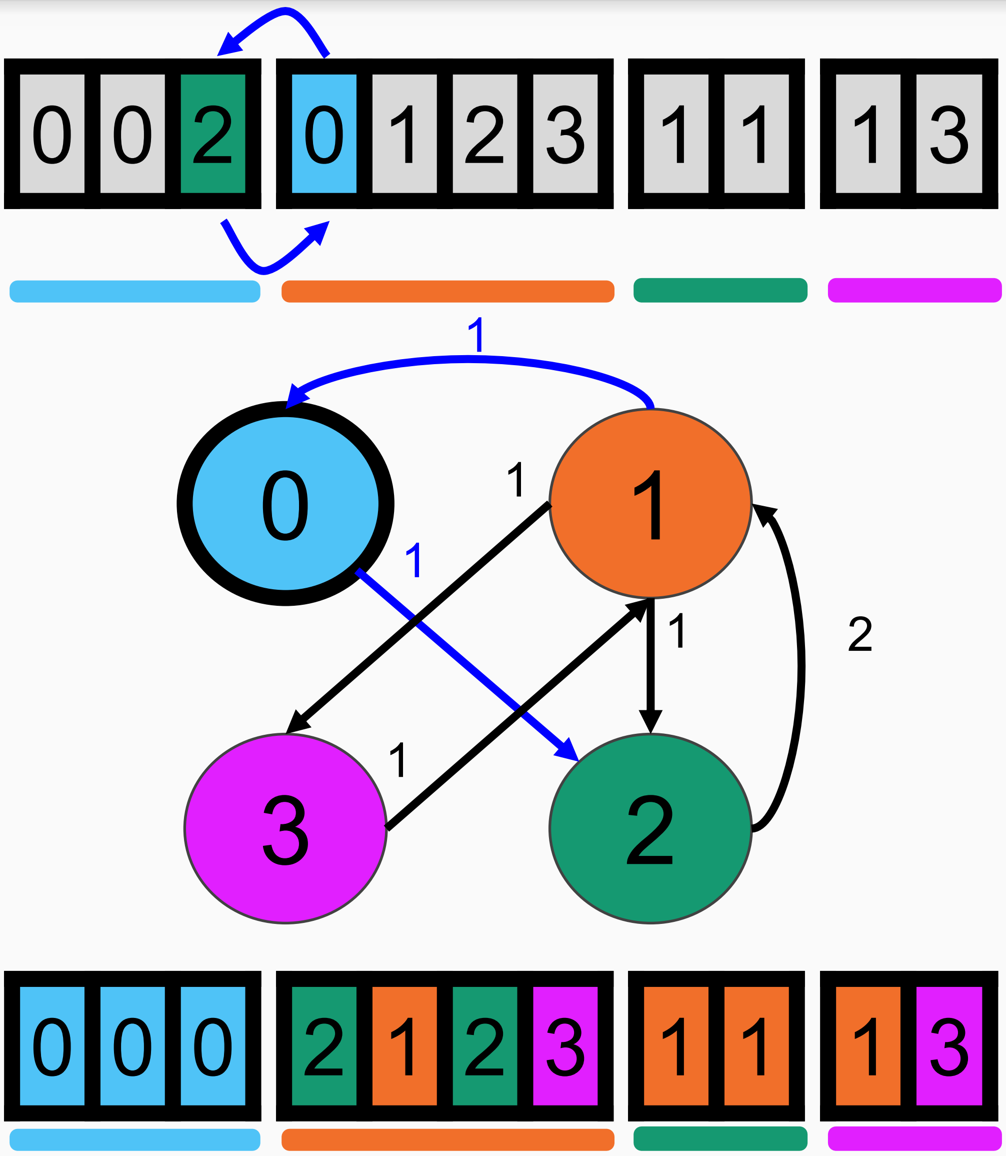
1. Choose a vertex.
2. Match incoming edges with outgoing edges.



Global Sorting: 2-Path Finding

2-path Finding

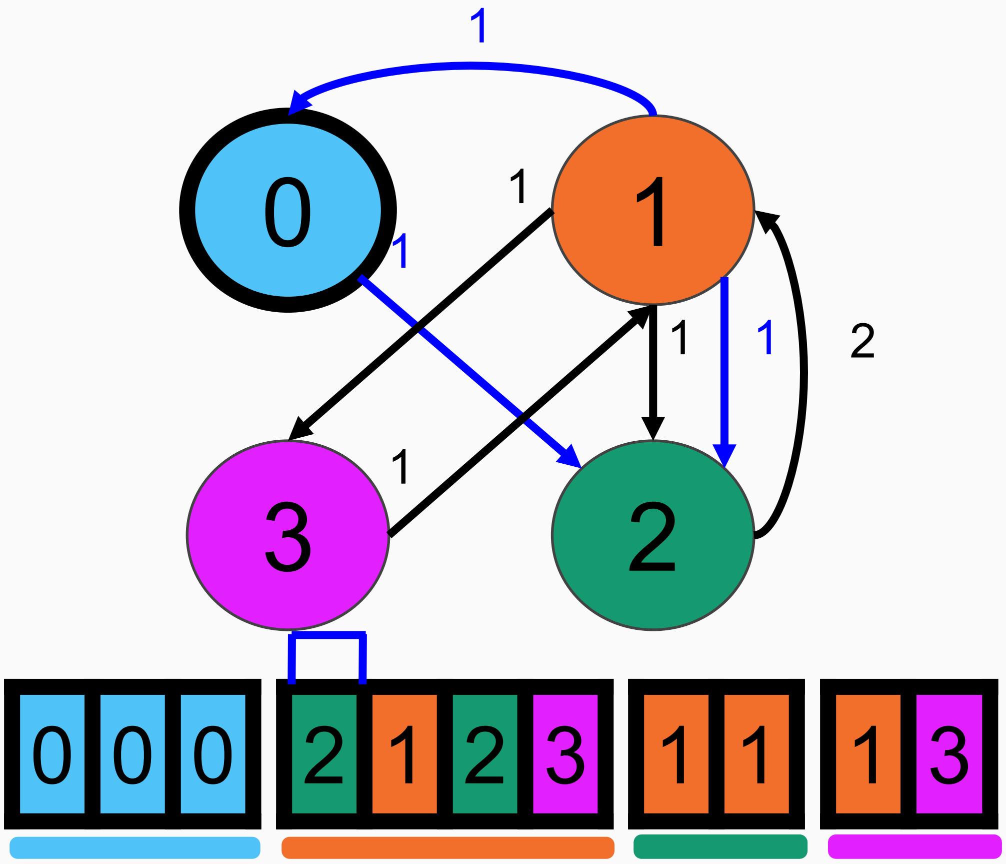
1. Choose a vertex.
2. Match incoming edges with outgoing edges.
3. Execute swaps.



Global Sort: 2-Path Finding

2-path Finding

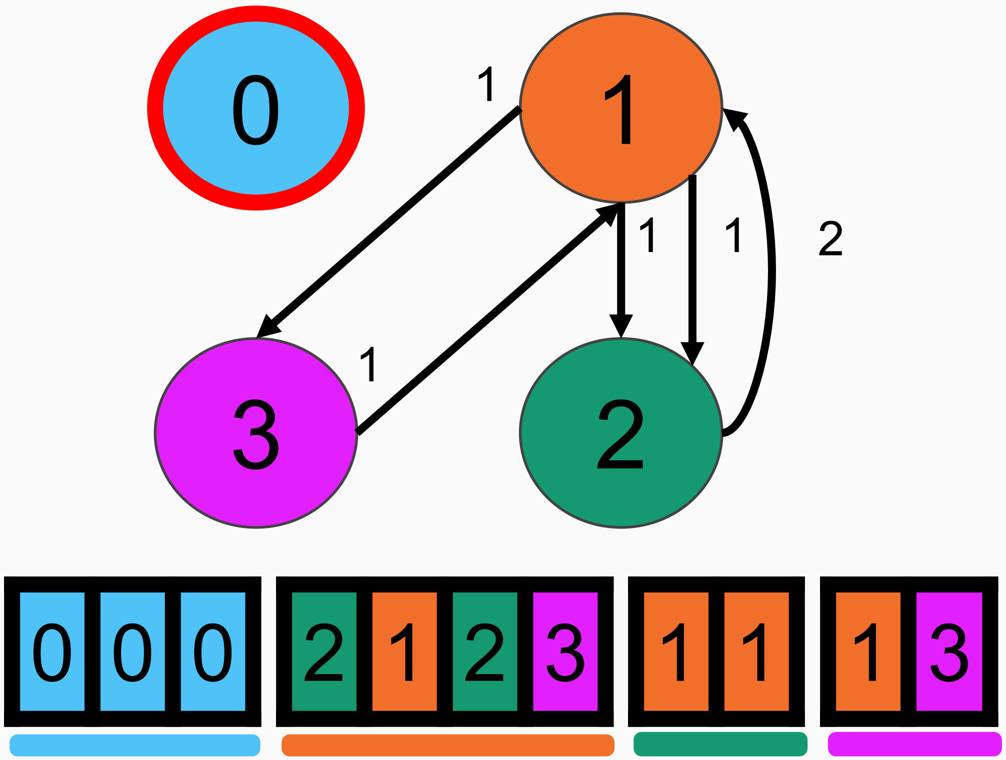
1. Choose a vertex.
2. Match incoming edges with outgoing edges.
3. Execute swaps.
4. Edit edges.



Global Sorting: 2-Path Finding

2-path Finding

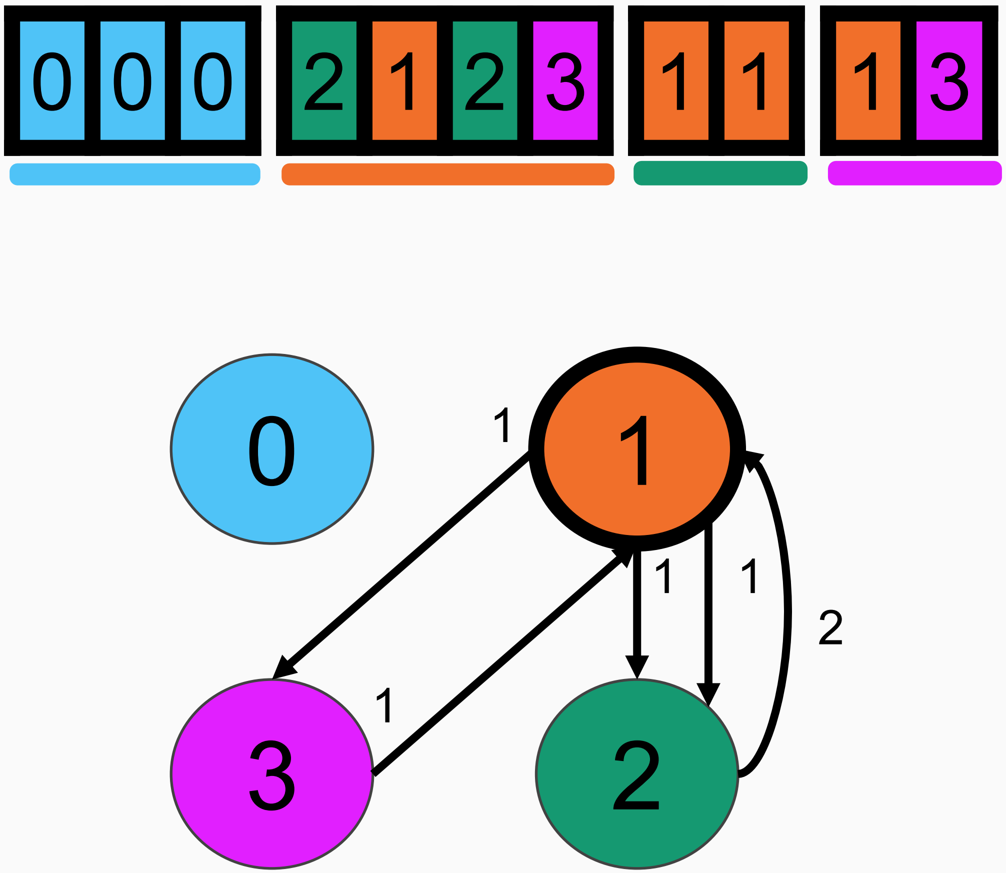
1. Choose a vertex.
2. Match incoming edges with outgoing edges.
3. Execute swaps.
4. Edit edges.



Global Sorting: 2-Path Finding

2-path Finding

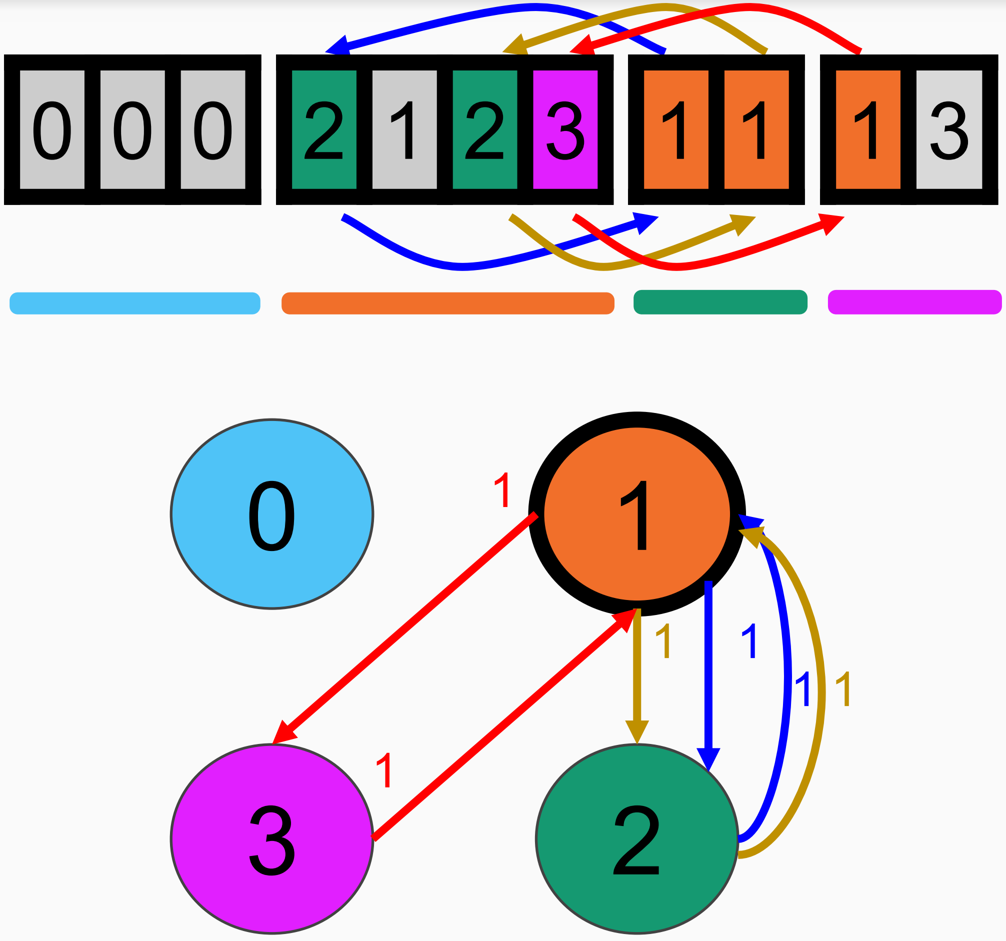
1. Choose a vertex.
2. Match incoming edges with outgoing edges.
3. Execute swaps.
4. Edit edges.



Global Sorting: 2-Path Finding

2-path Finding

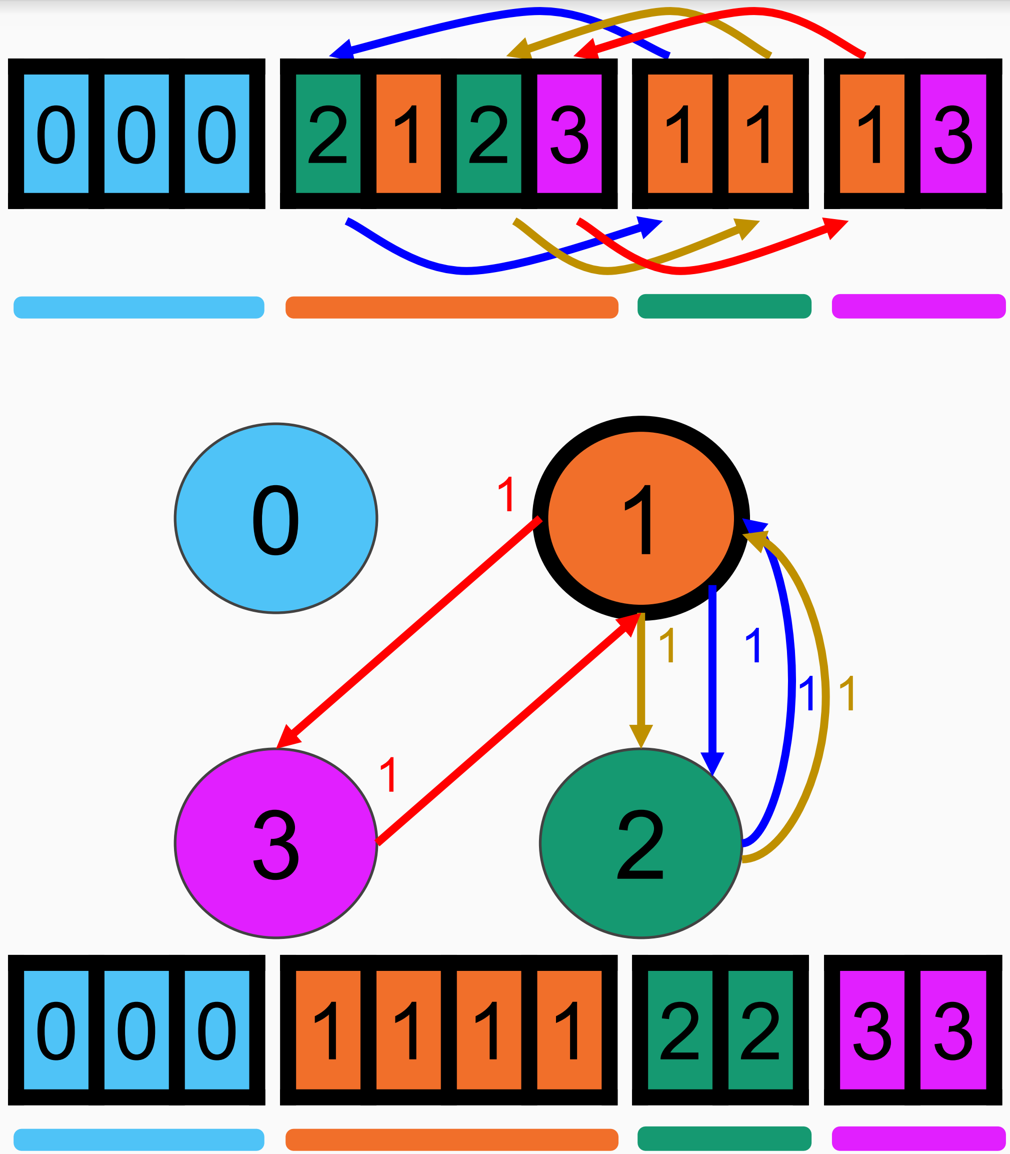
1. Choose a vertex.
2. Match incoming edges with outgoing edges.
3. Execute swaps.
4. Edit edges.



Global Sorting: 2-Path Finding

2-path Finding

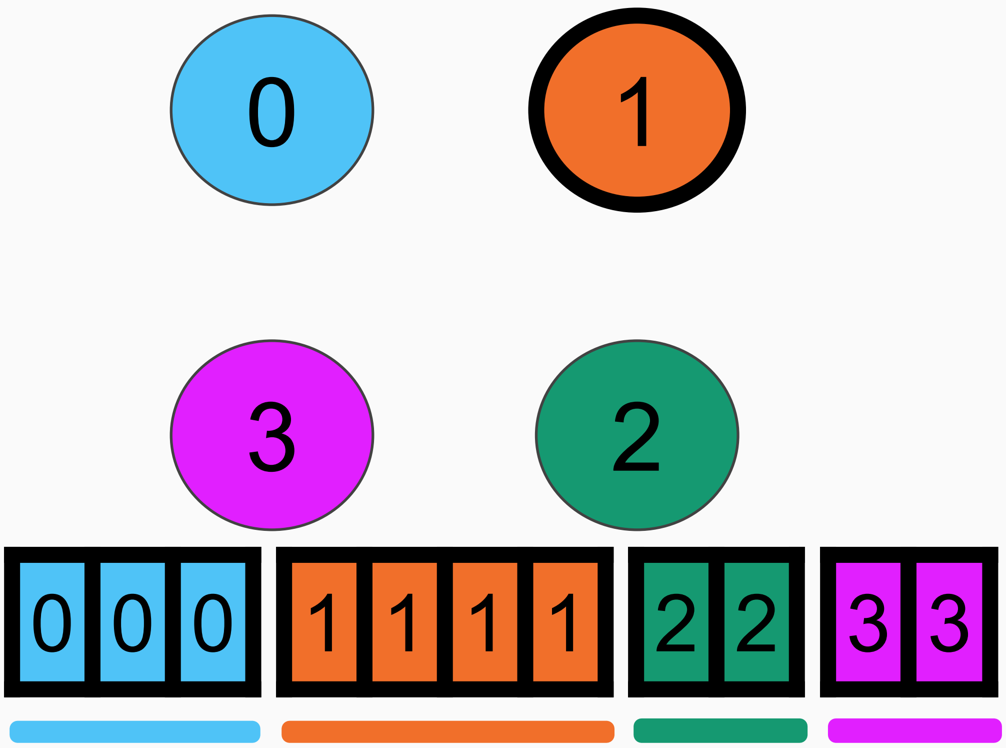
1. Choose a vertex.
2. Match incoming edges with outgoing edges.
3. Execute swaps.
4. Edit edges.



Global Sorting: 2-Path Finding

2-path Finding

1. Choose a vertex.
2. Match incoming edges with outgoing edges.
3. Execute swaps.
4. Edit edges.



Analysis

Analysis

1. Local Sorting
    a. Work: O(n)
    b. Span: O(log(K) + n/K)
    c. Space = O(KB)

- K is number of blocks
- B is number of buckets per block

Analysis

2. Build Regions Graph

a. Work = O(KB)
b. Span = O(log(KB))
c. Space = O(KB)

- Since #edges ≤ #regions ≤ KB
- K is number of blocks
- B is number of buckets per block

Analysis

3. Global Sorting

    a. Work = O(n)
    b. Span = O(B (log(KB) + B))
    c. Space = O(KB)

- O(n) swaps
- #nodes removed = O(B)
- #edges at each node removed is O(KB)

Analysis

Total for one level of recursion

$$Work = O(n)$$

$$Span = O(n/K + B\ (\log(KB) + B))$$

$$Space = O(KB)$$

Recursion

Recursion

- Each country is recursed on independently.
- Each country divided into number of blocks proportional to its size.
- Integers with range r need at most $\log_B(r)$ recursion levels to be fully sorted.
- For problem sizes smaller than B, we use comparison sort.

Algorithm: Recursion

Total on all levels

a. Work = $O(n \log(r))$
b. Span = $O((\log(K) + n/K) \log(r))$
c. Space = $O(P \log(r) + K)$

- Assuming $B = \Theta(1)$

Algorithm: Recursion

Total on all levels

a. Work = $O(n)$
b. Span = $O((\log(K) + n/K))$
c. Space = $O(P + K)$

- Assuming $B = \Theta(1)$
- Assuming $r = \Theta(1)$ (fixed length integers)

Alternative Approach: Cycle Finding

- Find Cycle in Regions Graph
- Execute Cycle to move elements
- Remove edge with min weight, and decrease weight of all other edges by this weight
- Repeat until all edges are deleted

Evaluation

Evaluation: Control Algorithms

State of the art parallel sorting algorithms:

- __gnu_parallel::sort (MCSTL, included in gcc) [Singler et. al 2007]
  - Not fully in-place; uses parallel mergesort
- RADULS (parallel out-of-place radix sort) [Kokot et al. 2017]
- PBBS parallel out-of-place radix sort [Shun et. al 2012]
- PBBS parallel out-of-place sample sort [Shun et. al 2012]
- Ska Sort (serial in-place radix sort)
- IPS4o (parallel in-place sample sort) [Axtmann et al. 2017]
- PARADIS (parallel in-place radix sort) not publicly available

Input distribution:

- Uniform
- Skewed
- Equal, and almost sorted

Evaluation: Our Algorithms

Our Algorithms

Cycle finding

$K = P$

$B = 256$

2-path finding

$K = 5000$

$B = 256$

Evaluation: Test Environment

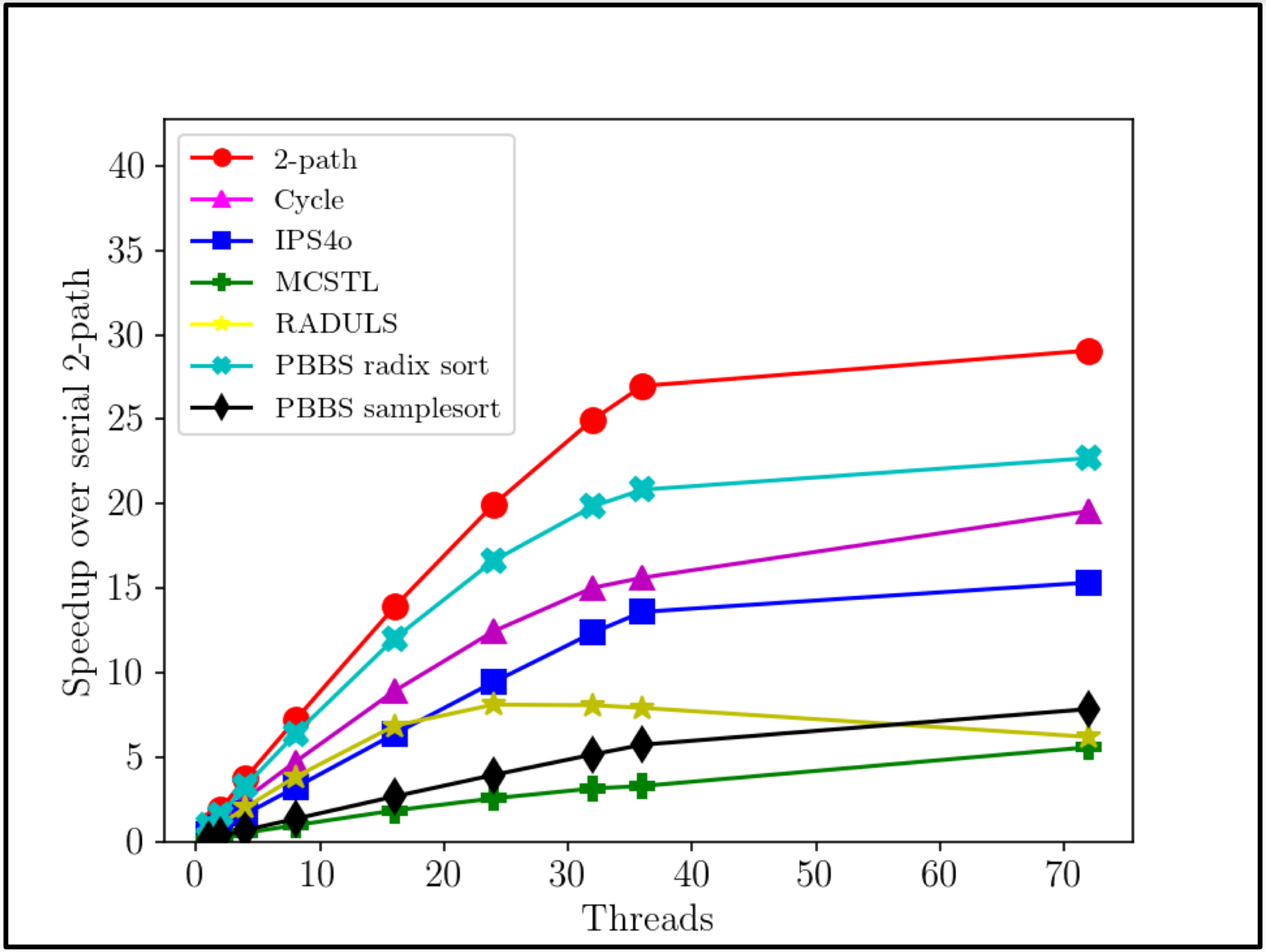
- AWS c5.9xlarge
- Intel Xeon Platinum 8000 series
- 72 vCPU (36 cores with hyperthreading)
- 144 GB RAM
- All code compiled with g++-7 with Cilk Plus

Comparison with other algorithms

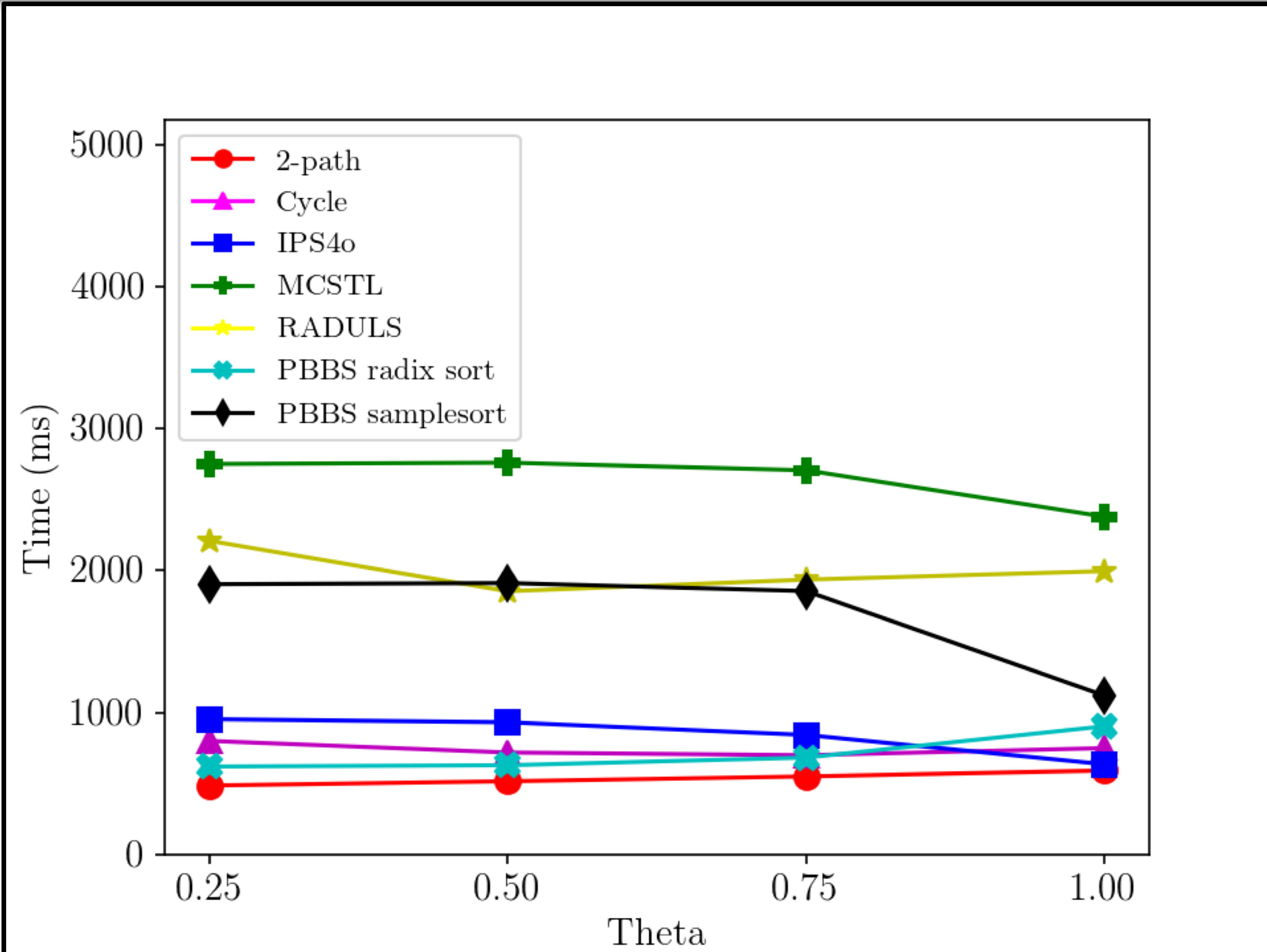
Regions Sort performance on various inputs with 1 billion integers:

- Between 1.1-3.6x faster than IPS4o, the fastest parallel sample sort, except on one input (1.02x slower).
- Between 1.2-4.4x faster than the fastest out-of-place Radix Sort (PBBS).
- 1.3x slower to 9.4x faster than RADULS.
- About 2x faster than PARADIS based on their reported numbers on same number of cores

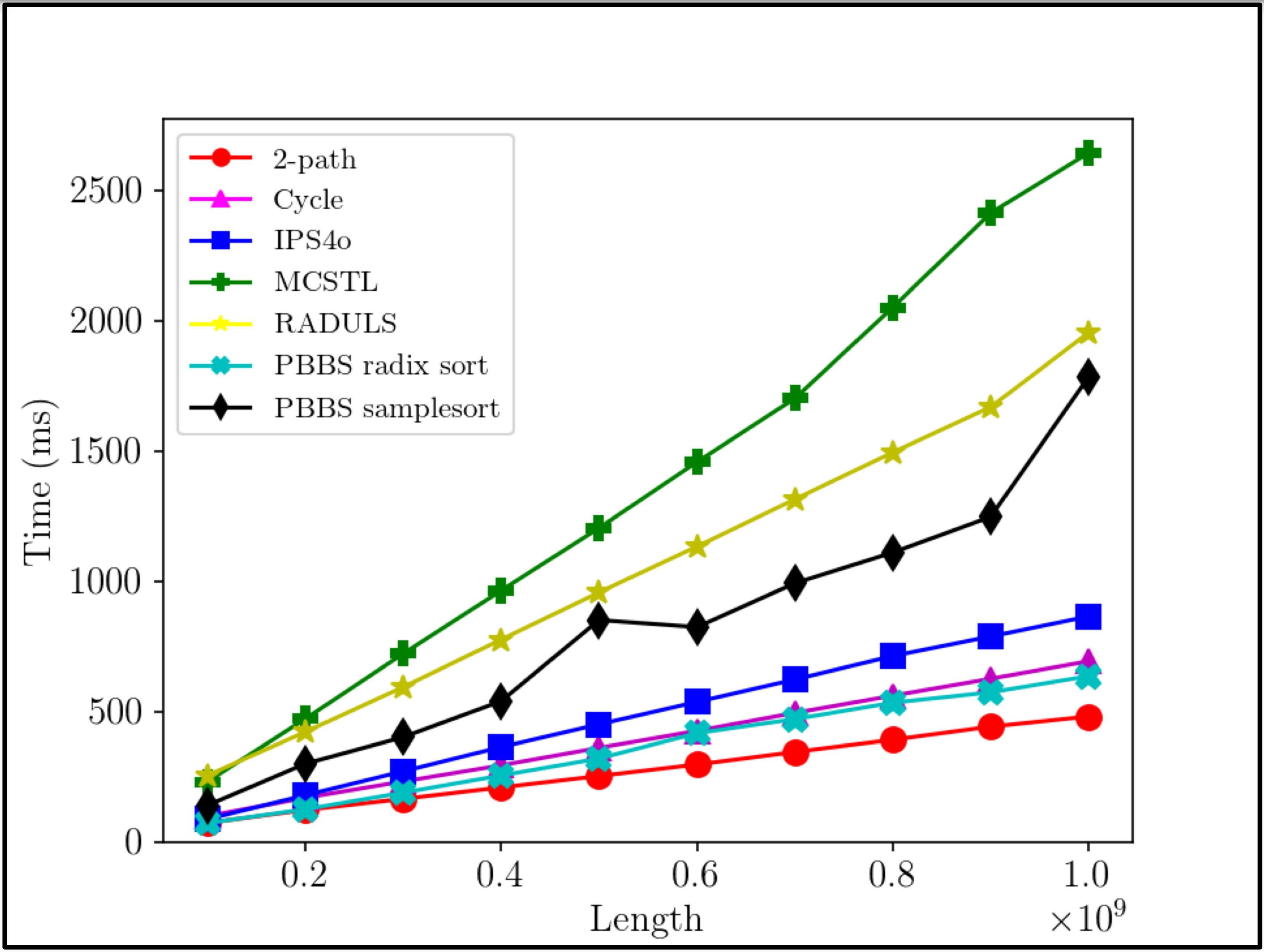
Speedup over serial 2-path: 1 billion random integers



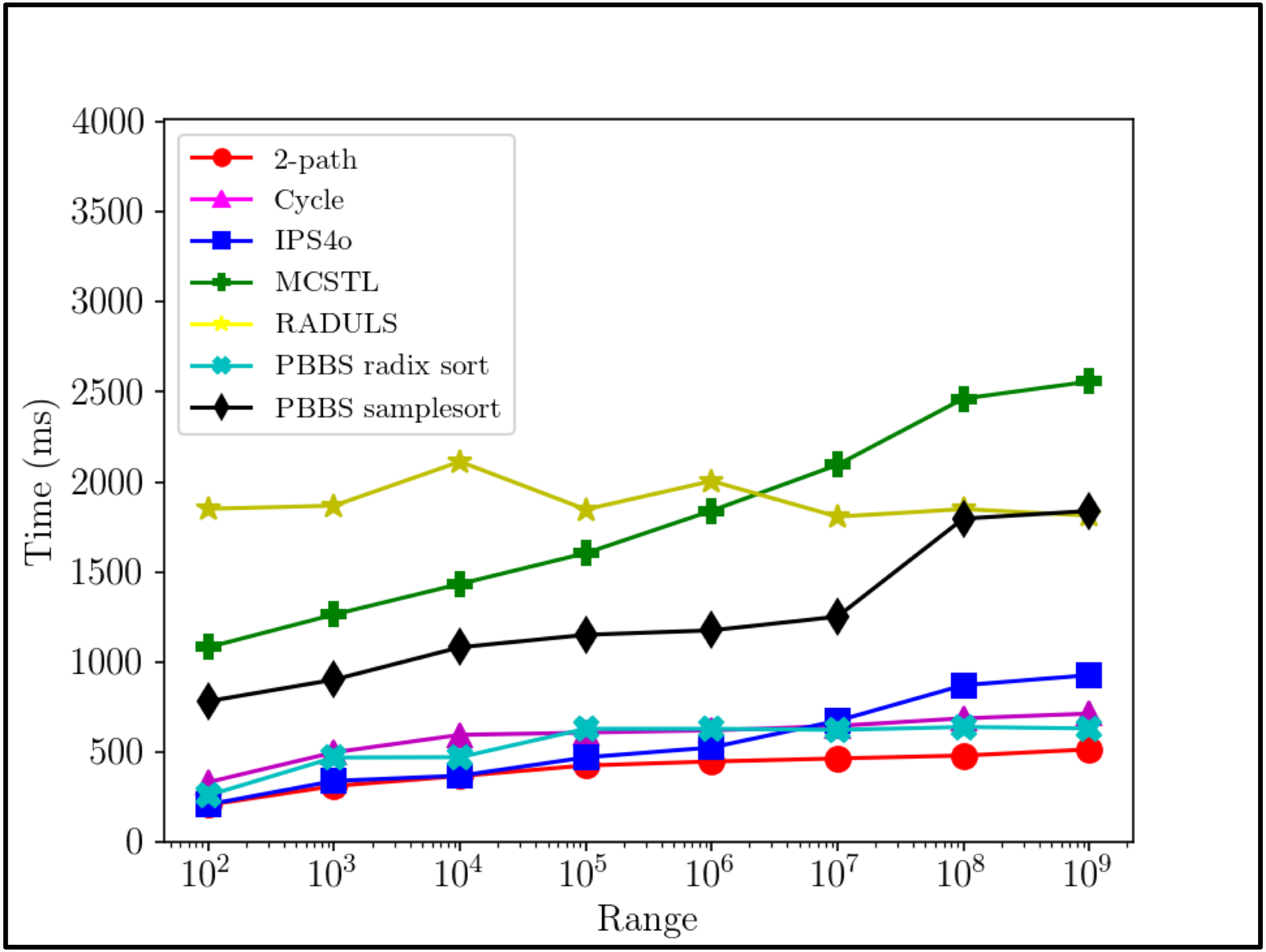
Distribution independence: 1 billion integers from Zipf



Regions Sort: fastest across all input sizes (Random)



Input Range - Uniform Sequence (1 billion integers)



Conclusion

Our contributions:

- Regions Sort: the first parallel in-place radix sort with strong theoretical guarantees.
- Empirical evidence showing high scalability and distribution independence.
- Almost always faster than state-of-the-art parallel sorting algorithms in our experiments.