Theoretically-Efficient and Practical Parallel In-Place Radix Sorting

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

Affiliation: Massachusetts Institute of Technology

Agenda

- Introduction
    - Motivation
    - Related Work
- Regions Sort
    - Algorithm Design
    - Theoretical Analysis
- Experiments
    - Setup
    - Results

Motivation

Why Radix Sort?

Takes O(n) work for fixed length integers.

Comparison-based sorts take **Ω**(nlog(n)) work.

In-Place Algorithms

What are in-place algorithms?

- Require at most sublinear auxiliary space.

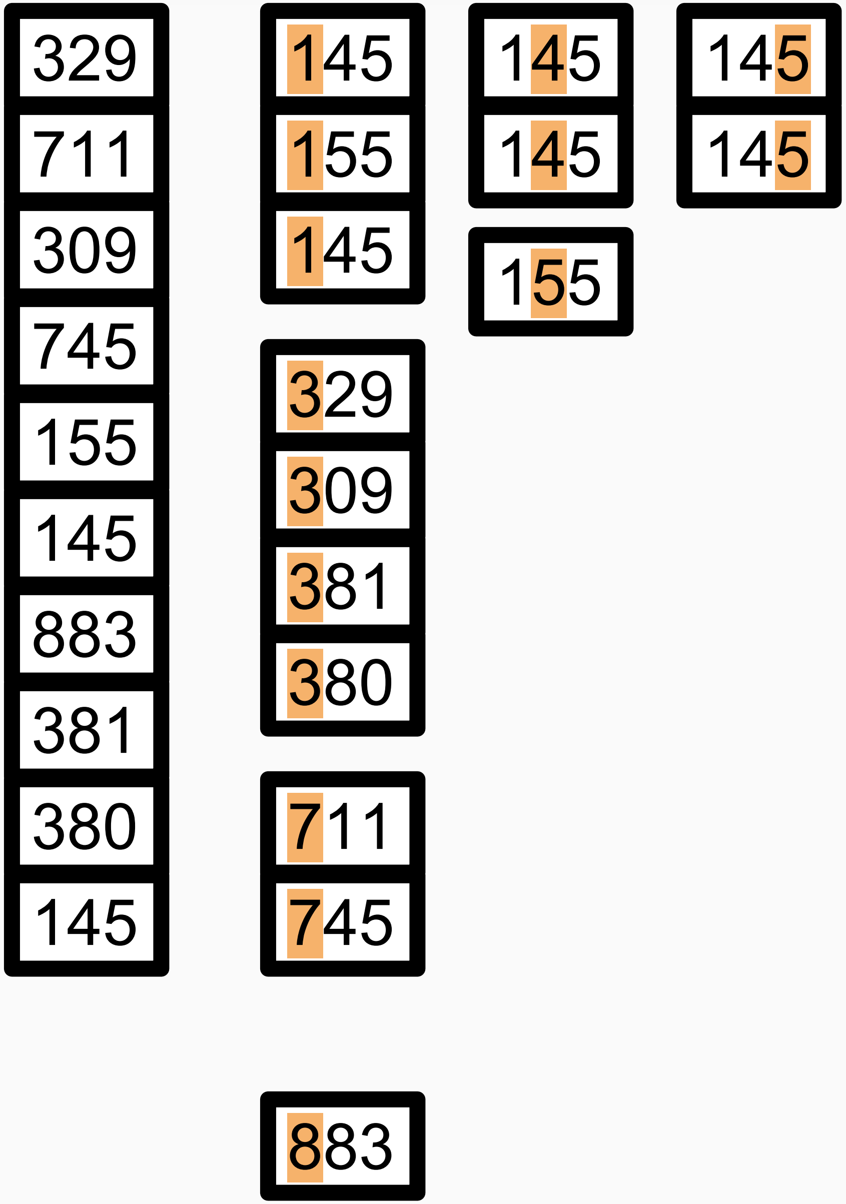
Why in-place?

- Smaller memory footprint!
- Potentially better utilization of cache.

(Most Significant Digit First) Radix Sort

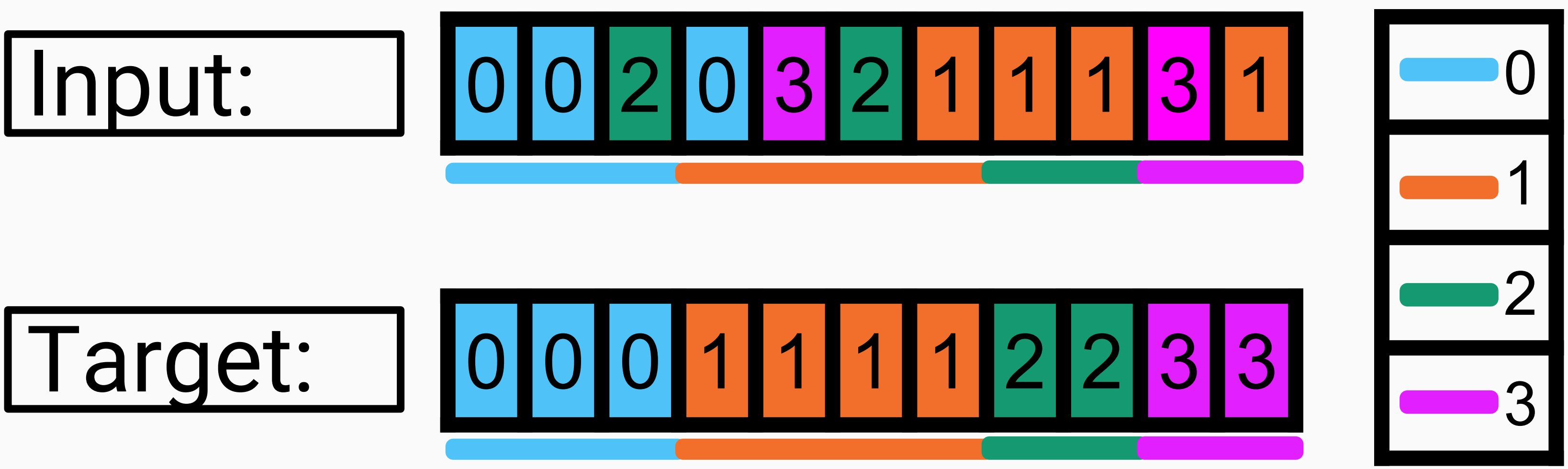
Radix Sort

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



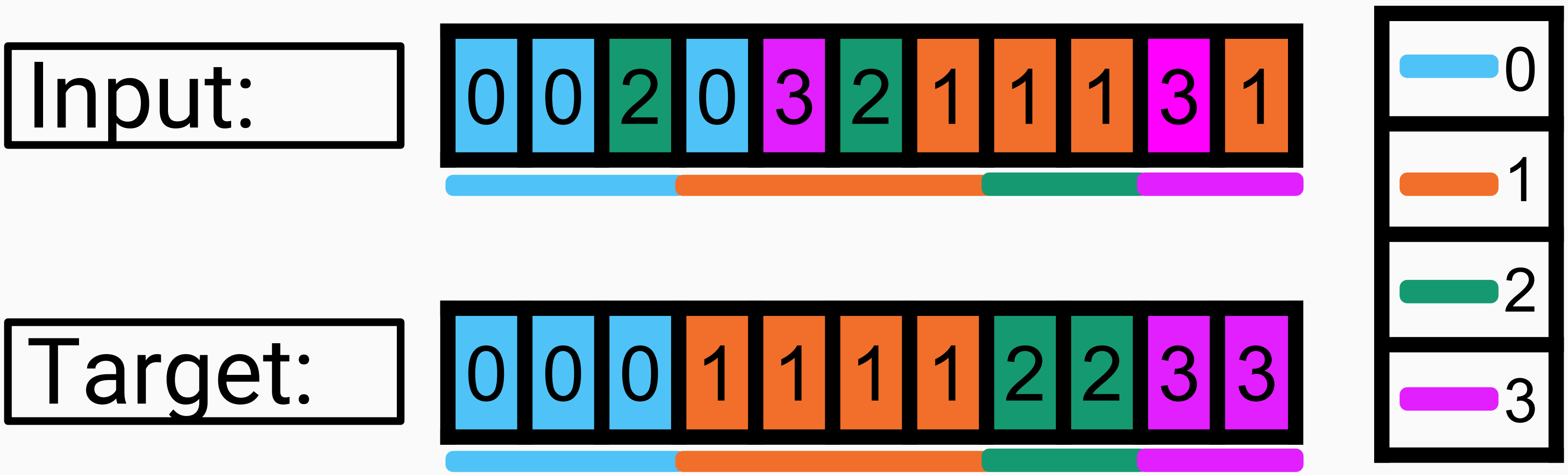
Terminology: Country

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.

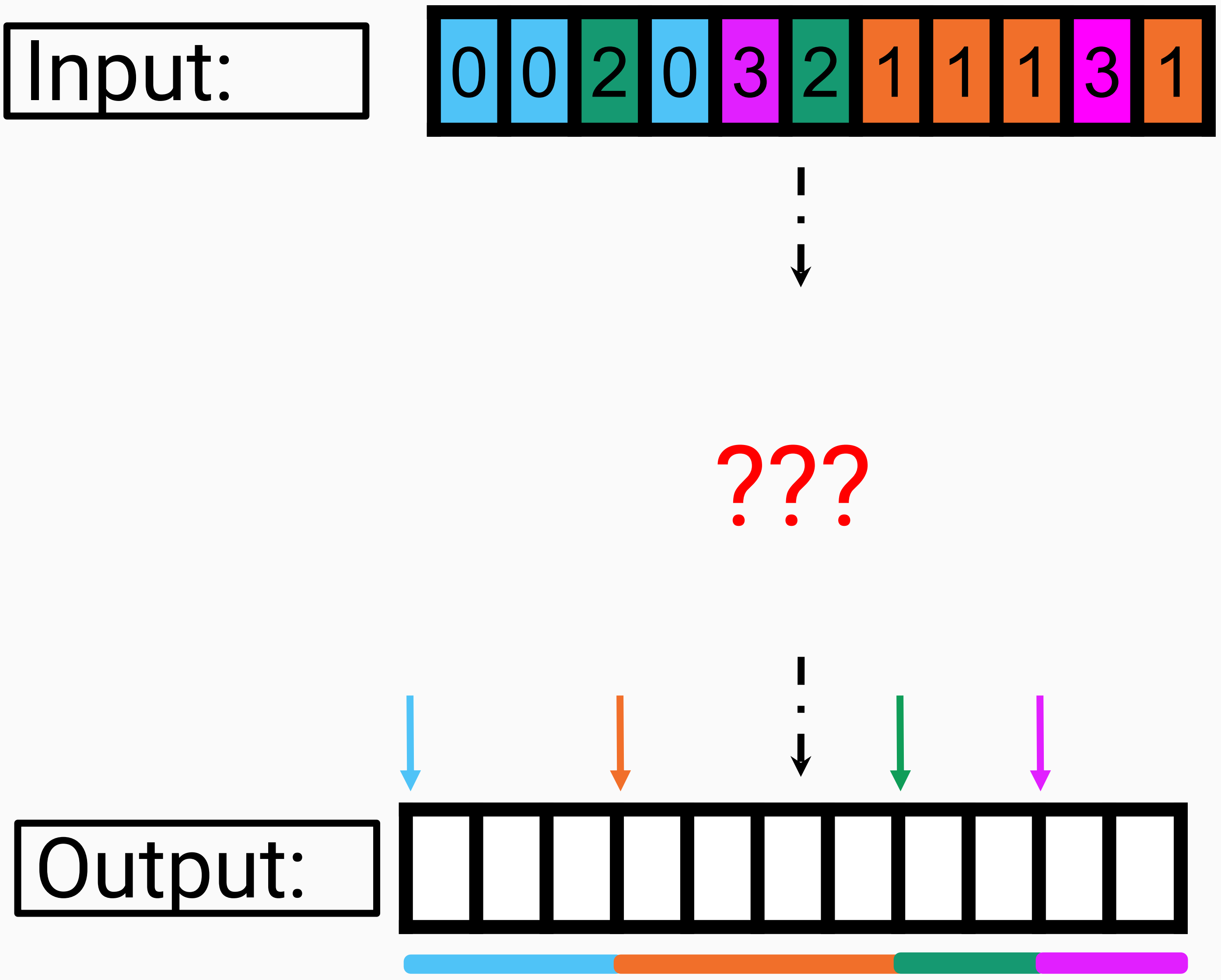


Serial In-place Radix Sort

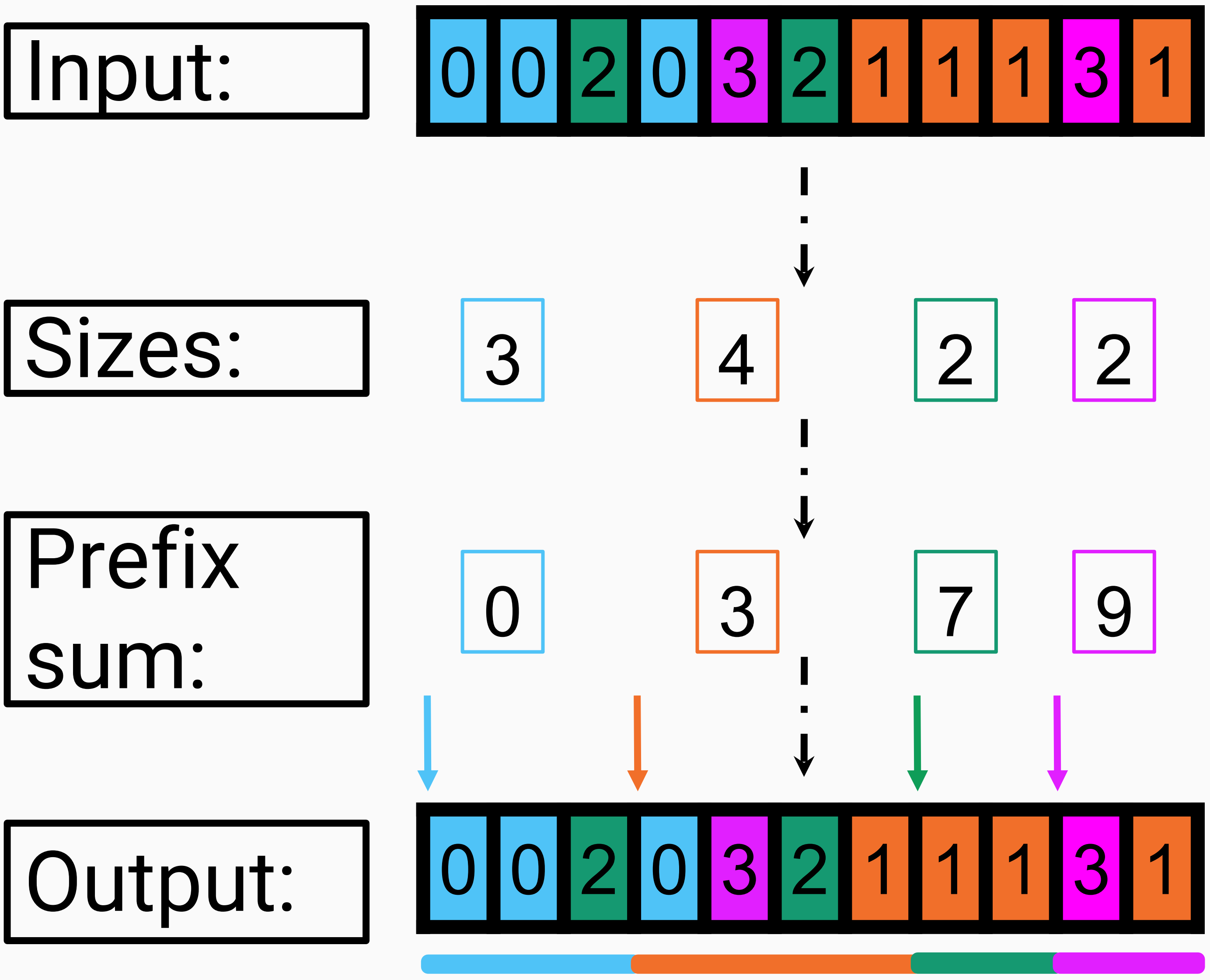
1. Find start location of each country (Histogram Building).

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

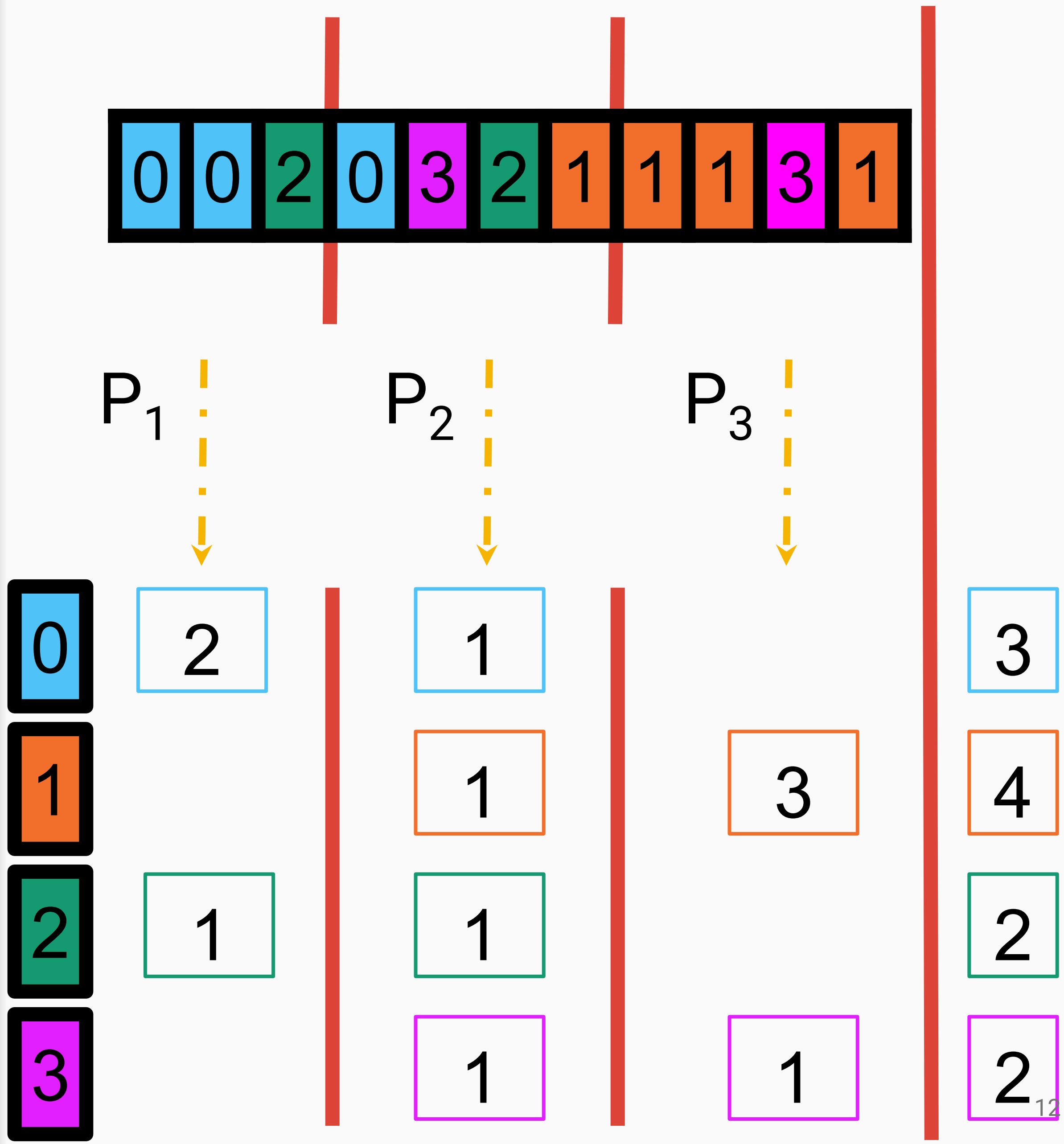
Histogram Building



Histogram Building



Parallel Histogram Building



Serial In-place Radix Sort

In-Place Radix Sort

For each bucket:

While (pointer not at end) {

While(item bucket != current bucket) {

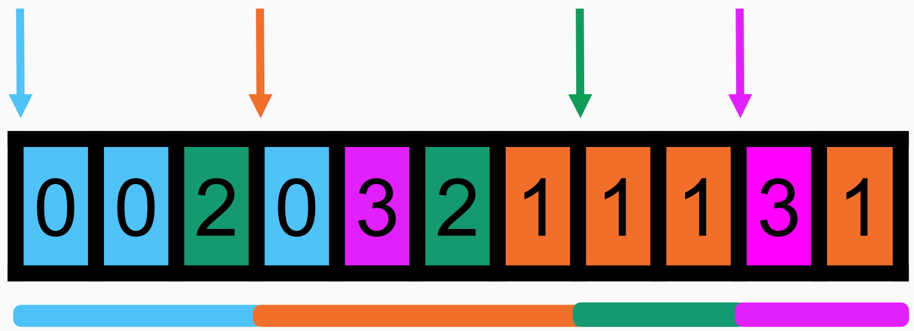
Swap item to target bucket

Update target bucket pointer

}

Update current bucket pointer

}



Serial In-place Radix Sort

In-Place Radix Sort

For each bucket:

While (pointer not at end) {

While(item bucket != current bucket) {

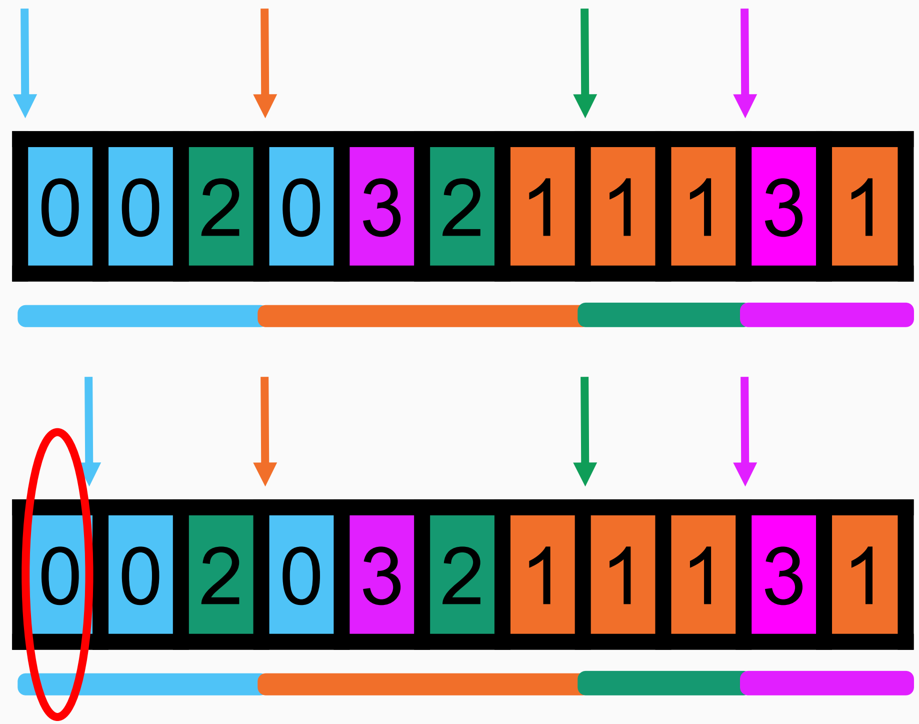
Swap item to target bucket

Update target bucket pointer

}

Update current bucket pointer

}



Serial In-place Radix Sort

In-Place Radix Sort

For each bucket:

While (pointer not at end) {

While(item bucket != current bucket) {

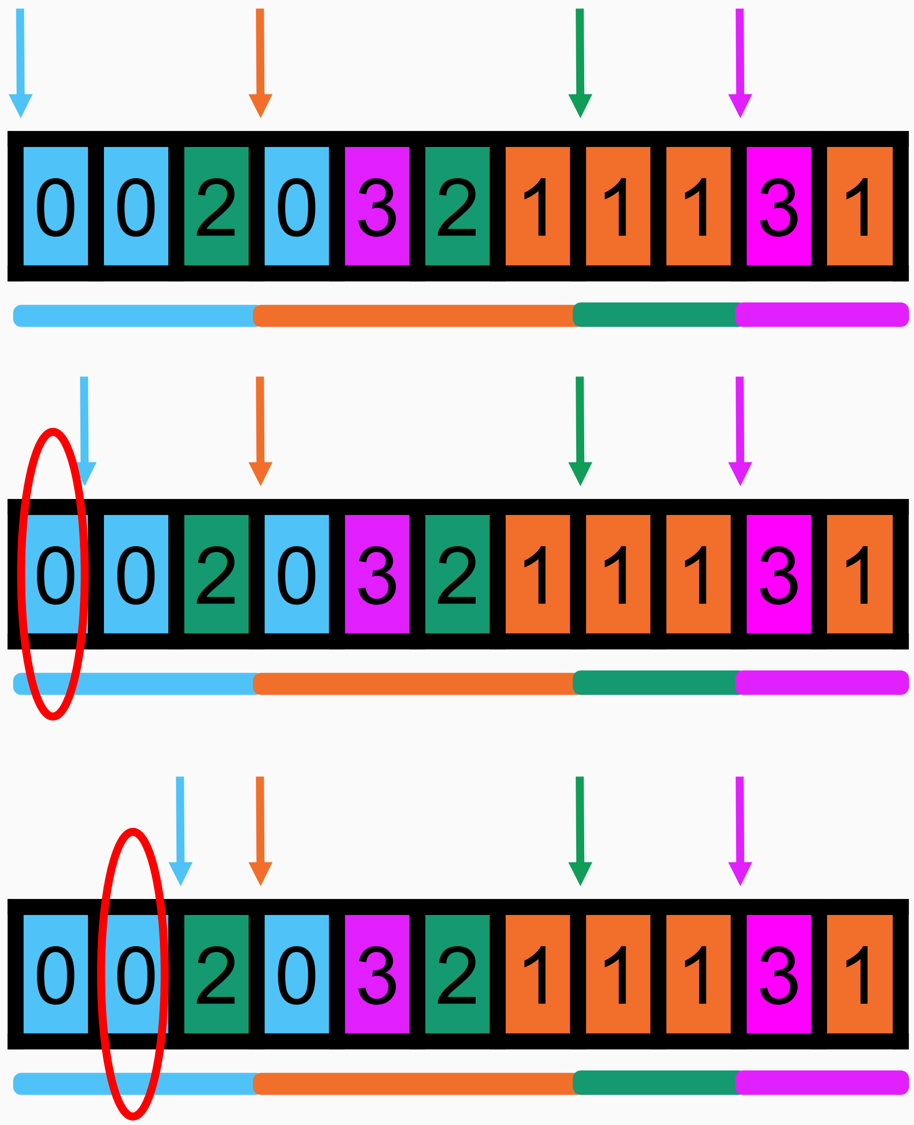
Swap item to target bucket

Update target bucket pointer

}

Update current bucket pointer

}



Serial In-place Radix Sort

In-Place Radix Sort

For each bucket:

While (pointer not at end) {

While(item bucket != current bucket) {

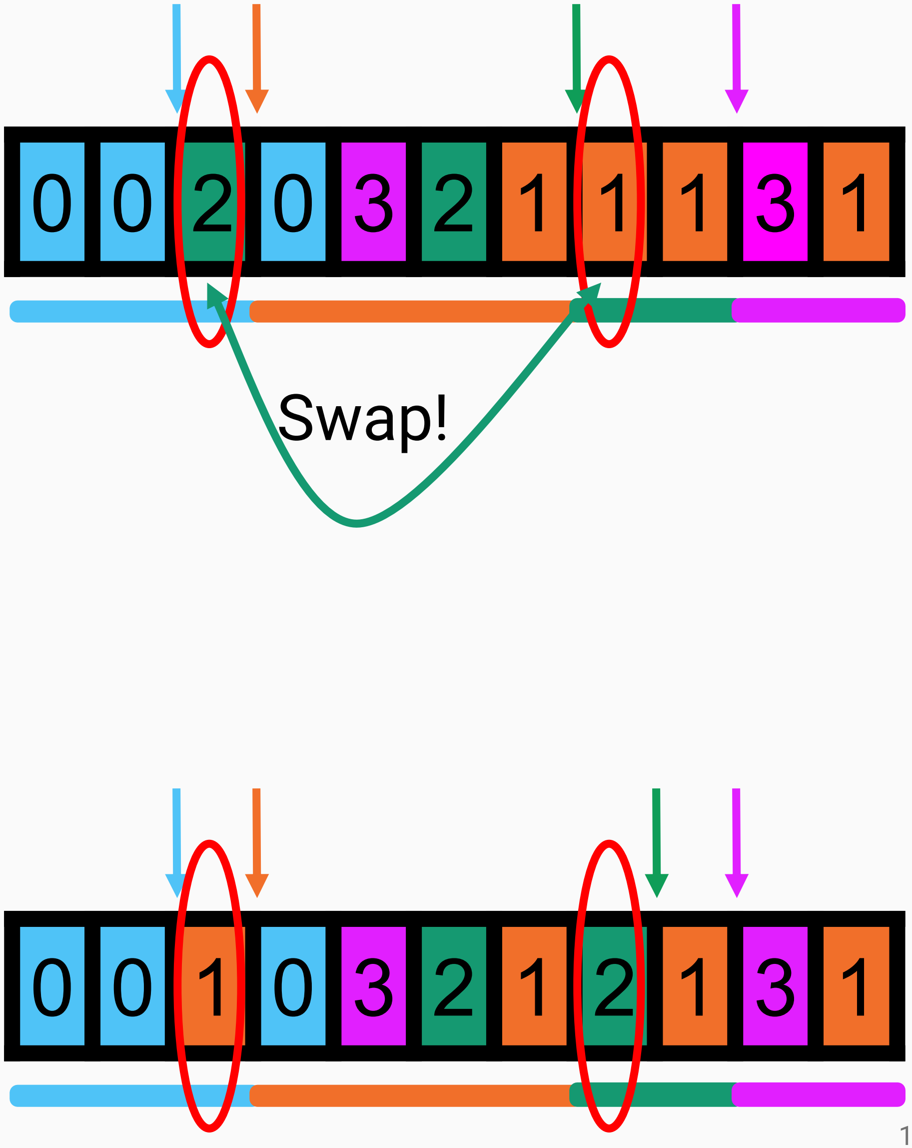
Swap item to target bucket

Update target bucket pointer

}

Update current bucket pointer

}



Serial In-place Radix Sort

In-Place Radix Sort

For each bucket:

While (pointer not at end) {

While(item bucket != current bucket) {

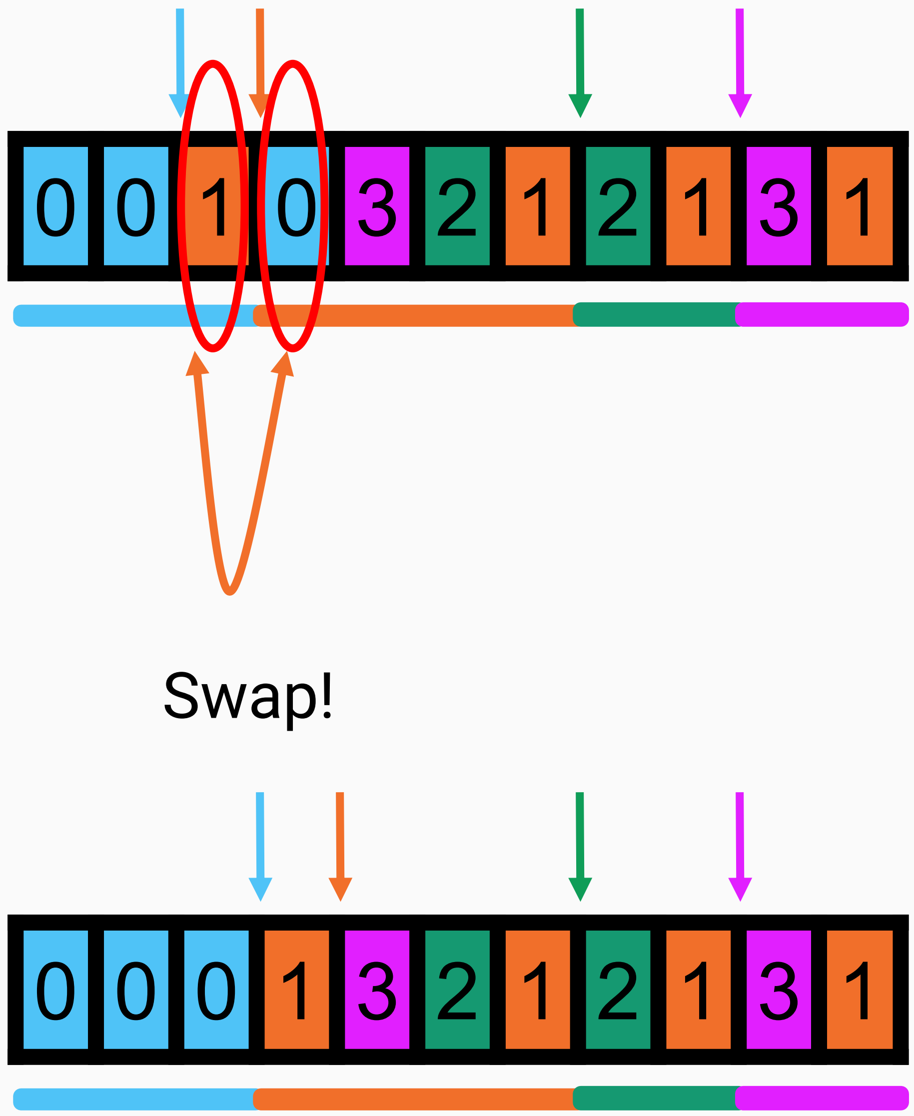
Swap item to target bucket

Update target bucket pointer

}

Update current bucket pointer

}



Serial In-place Radix Sort

In-Place Radix Sort

For each bucket:

While (pointer not at end) {

While(item bucket != current bucket) {

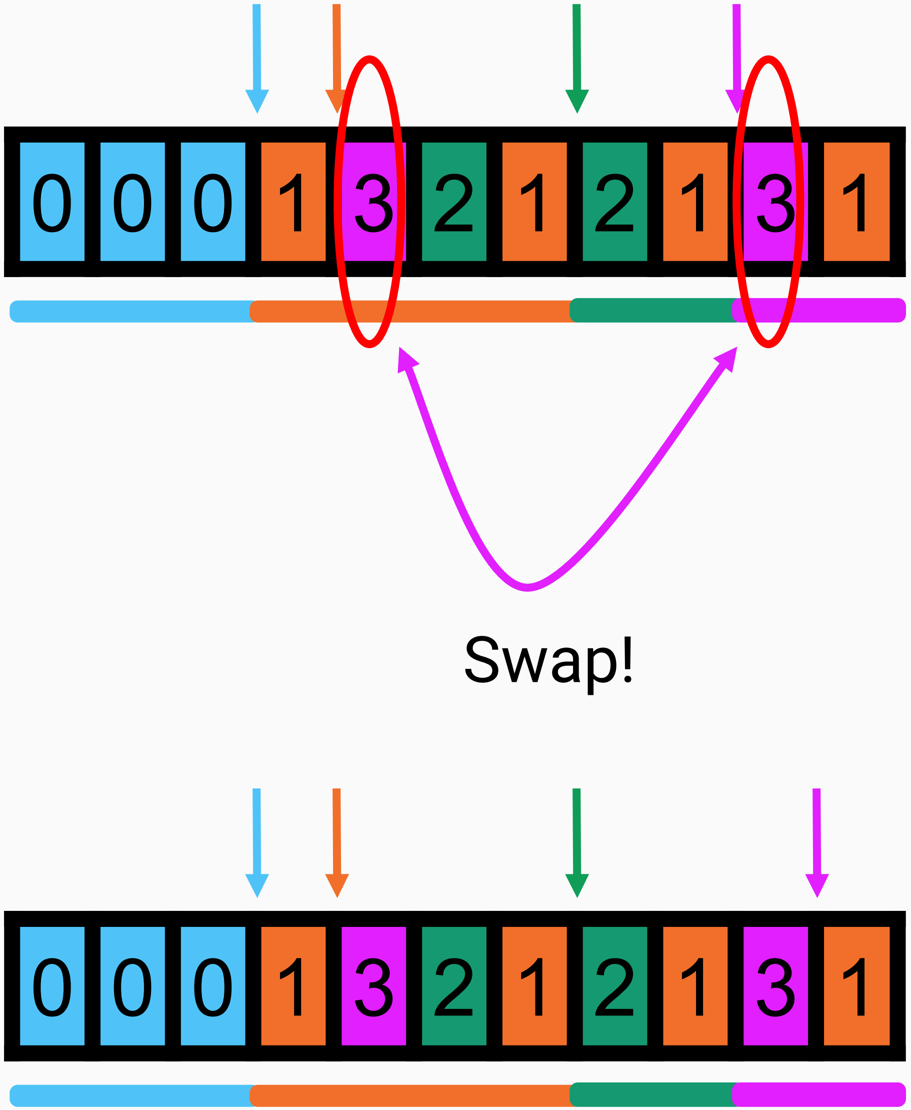
Swap item to target bucket

Update target bucket pointer

}

Update current bucket pointer

}



Serial In-place Radix Sort

In-Place Radix Sort

For each bucket:

While (pointer not at end) {

While(item bucket != current bucket) {

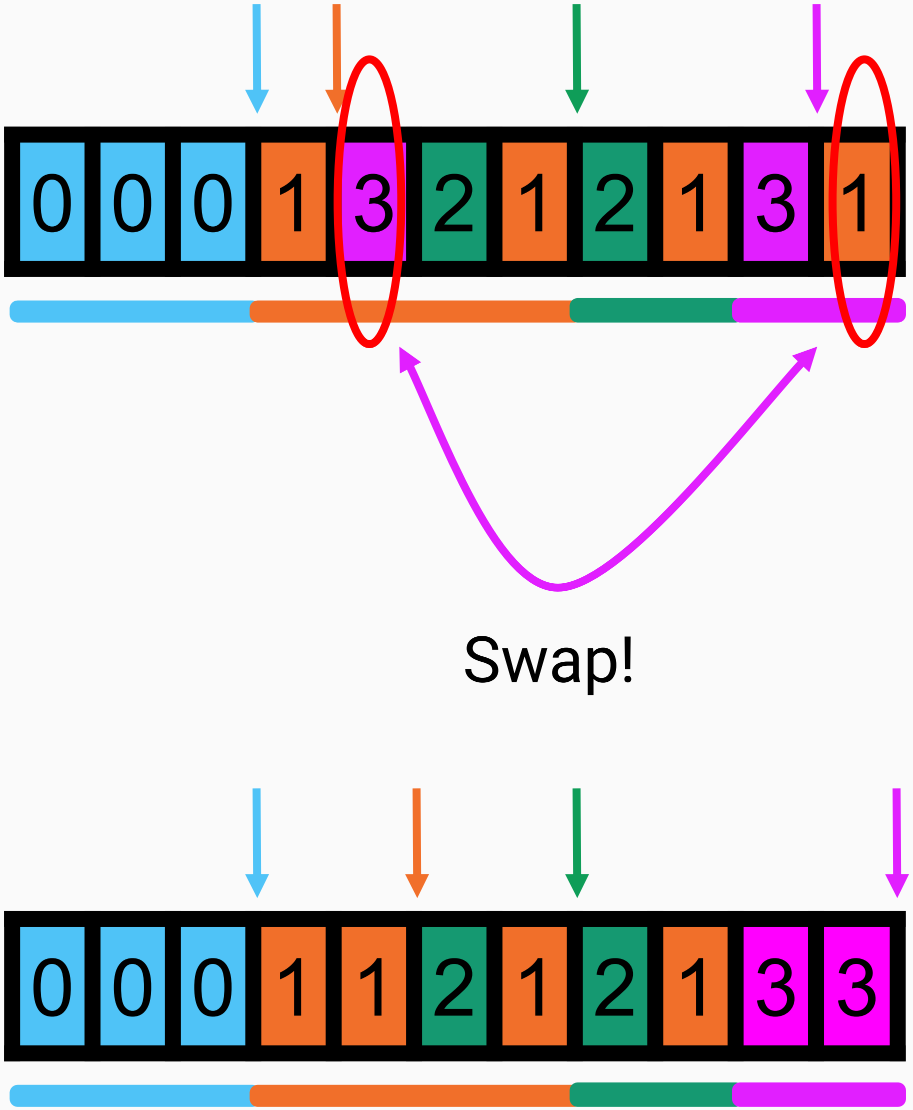
Swap item to target bucket

Update target bucket pointer

}

Update current bucket pointer

}



Serial In-place Radix Sort

In-Place Radix Sort

For each bucket:

While (pointer not at end) {

While(item bucket != current bucket) {

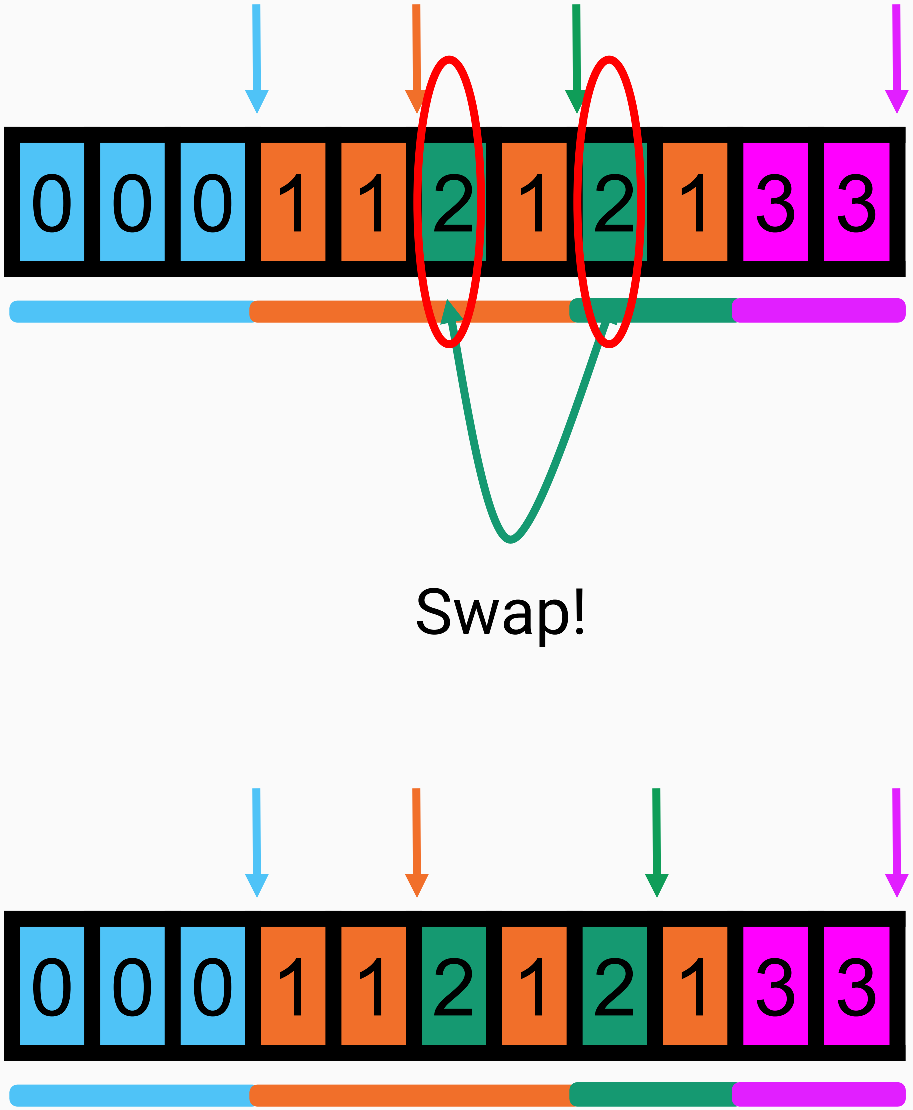
Swap item to target bucket

Update target bucket pointer

}

Update current bucket pointer

}



Serial In-place Radix Sort

In-Place Radix Sort

For each bucket:

While (pointer not at end) {

While(item bucket != current bucket) {

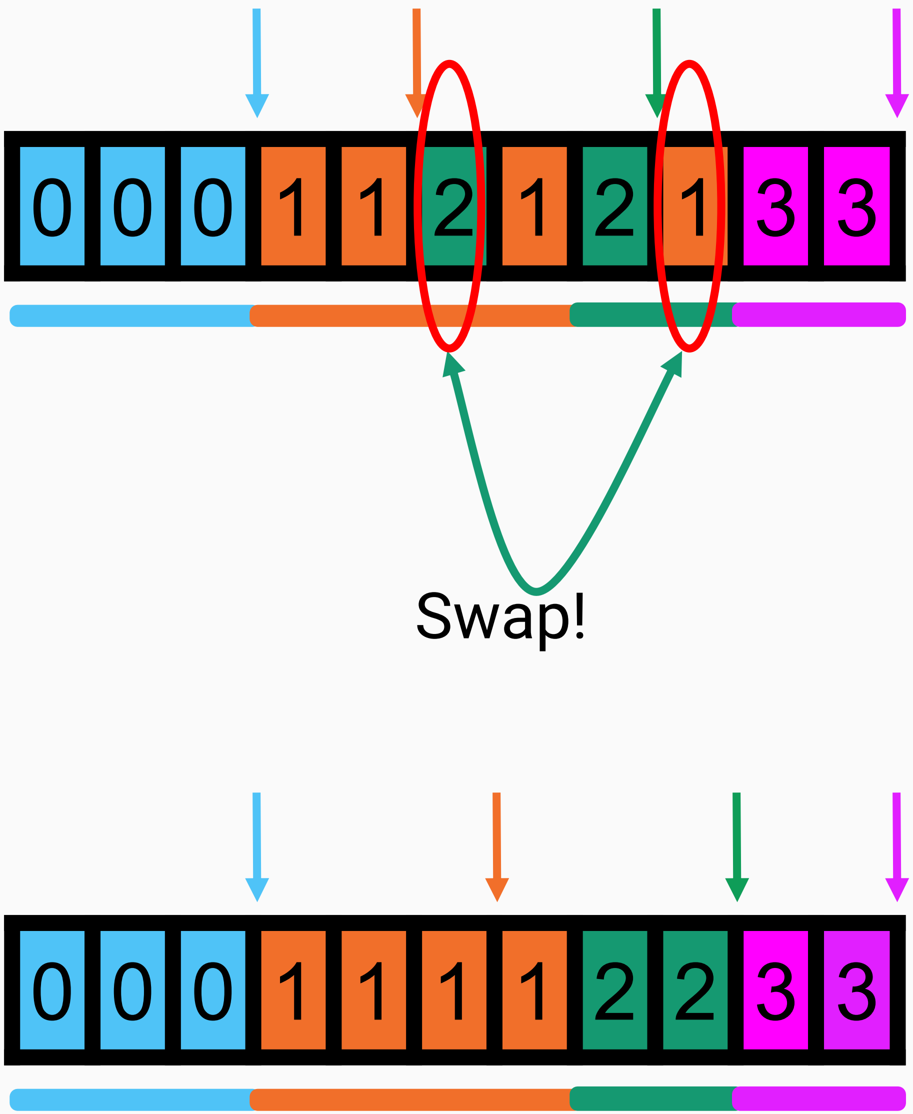
Swap item to target bucket

Update target bucket pointer

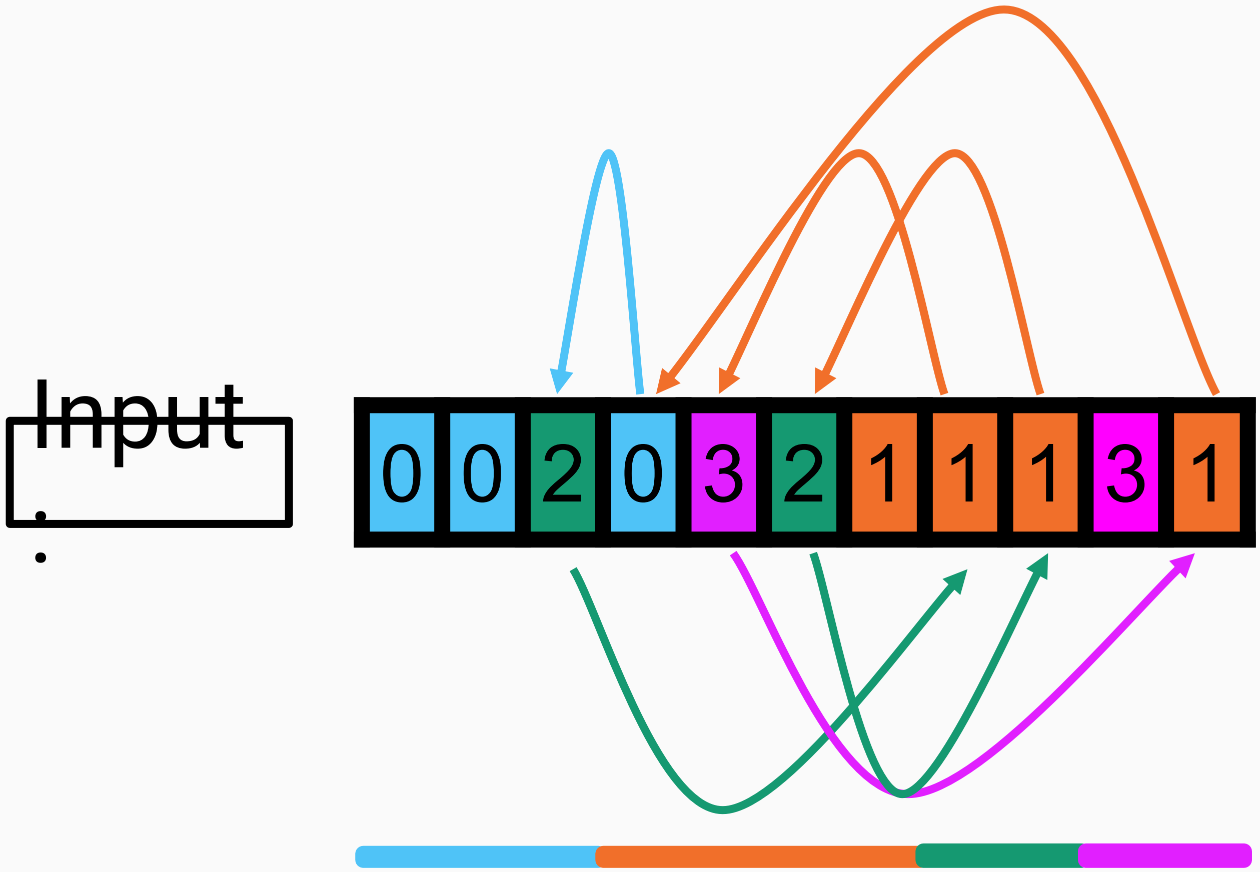
}

Update current bucket pointer

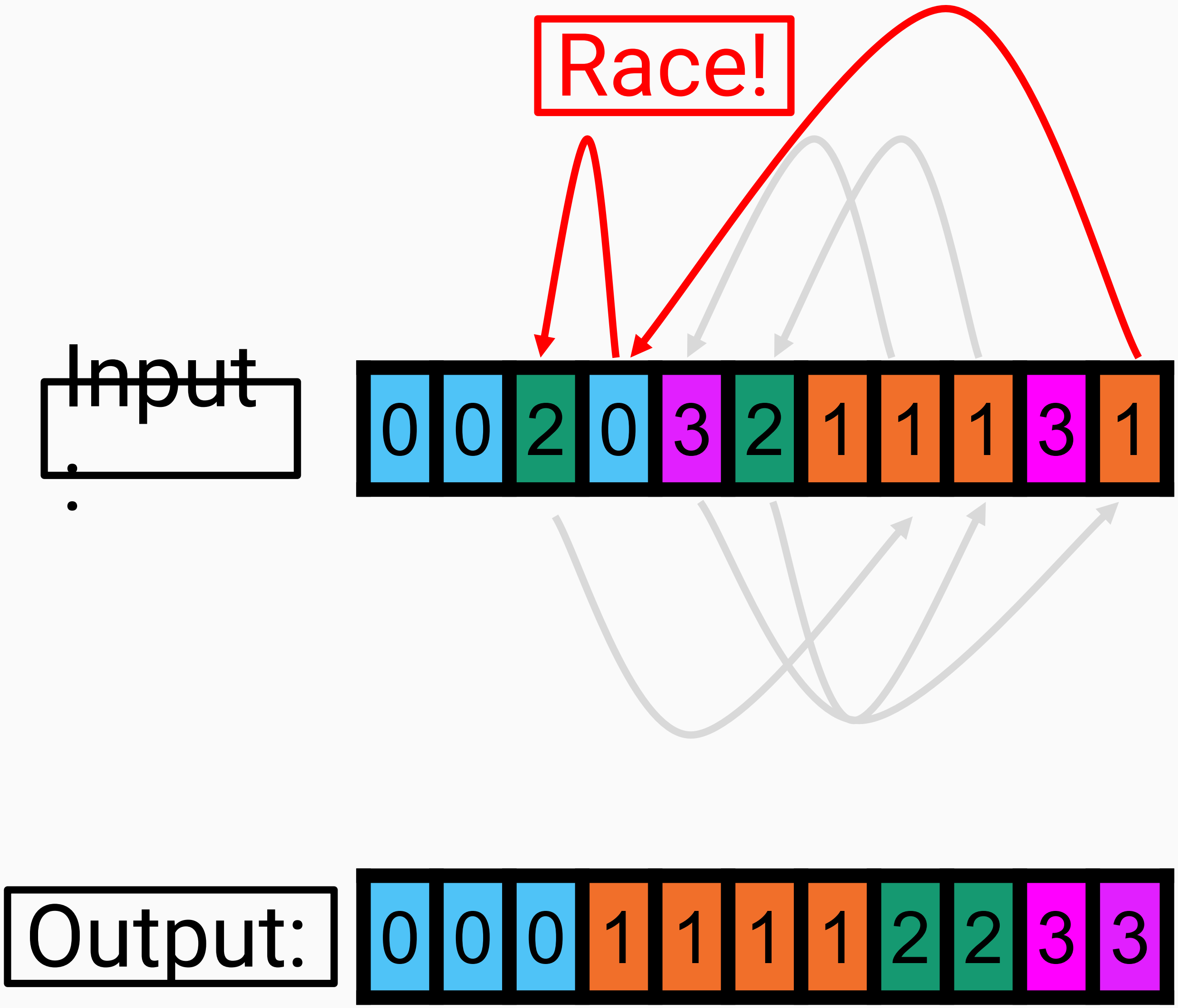
}



Why parallel in-place is hard?!



Why parallel in-place is hard?!



Model

Assuming Work-Span Model:

a. Work: is number of operations.
b. Span: longest dependence in the computation.

Parallelism = Work/Span.

Time = O(Work/Processors + Span).

Related Work

1. PARADIS [Cho et. al 2015]
    - Parallel in-place radix sort.
    - Worst case span is O(n).

1. IPS4o [Axtmann et. al 2017]
    - Parallel in-place comparison based sort.
    - Work is O(nlog(n)).

1. RADULS [Kokot et. al 2018]
    - Parallel radix sort with auxiliary memory linear in input size.

Goal

Using P processors:

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

For 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 input with small amount of memory.

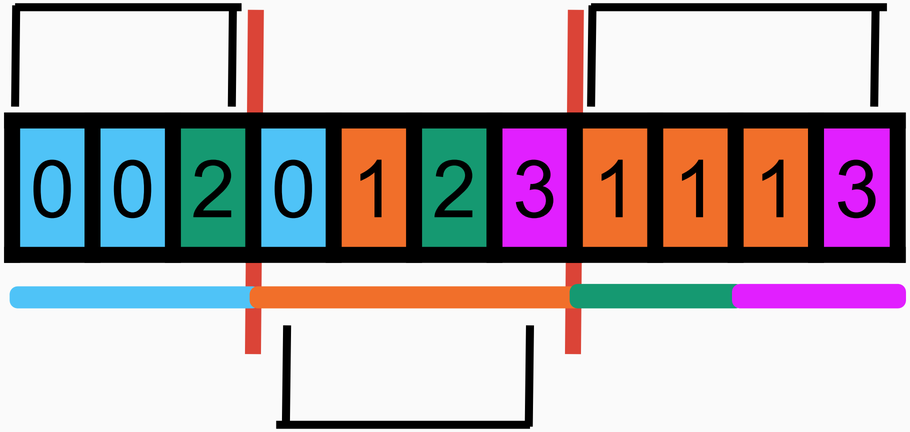
3. Global Sorting
    - Use regions graph to completely sort the input.

Local Sorting

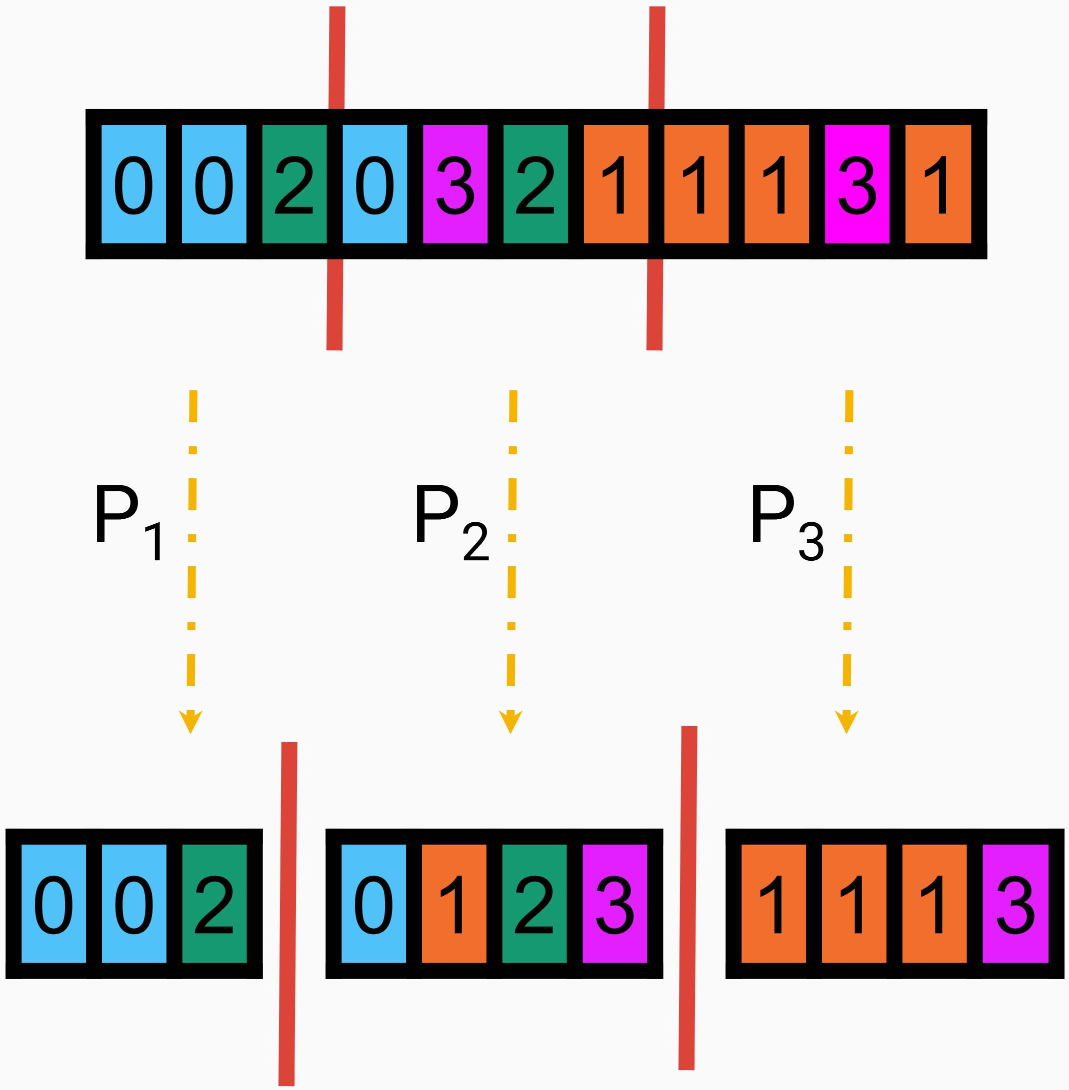
Key Idea:

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

Block: sub-array of size n/K.



Local Sorting

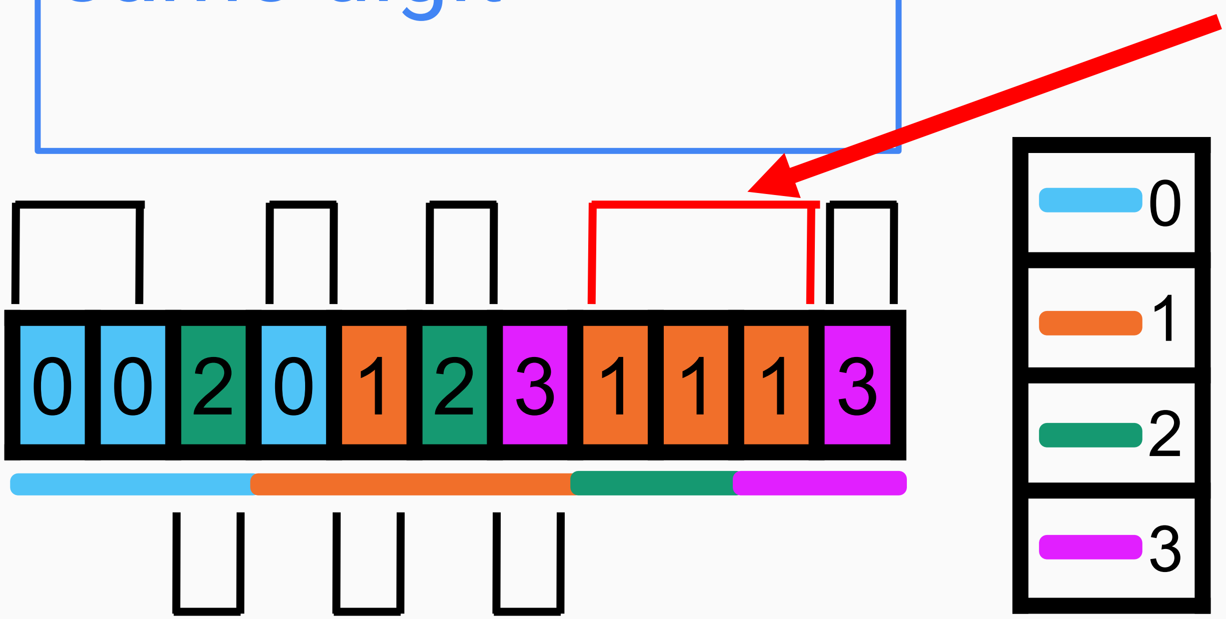


Regions Graph Building

Key Idea: Represent dependencies in partially sorted input with small amount of memory.

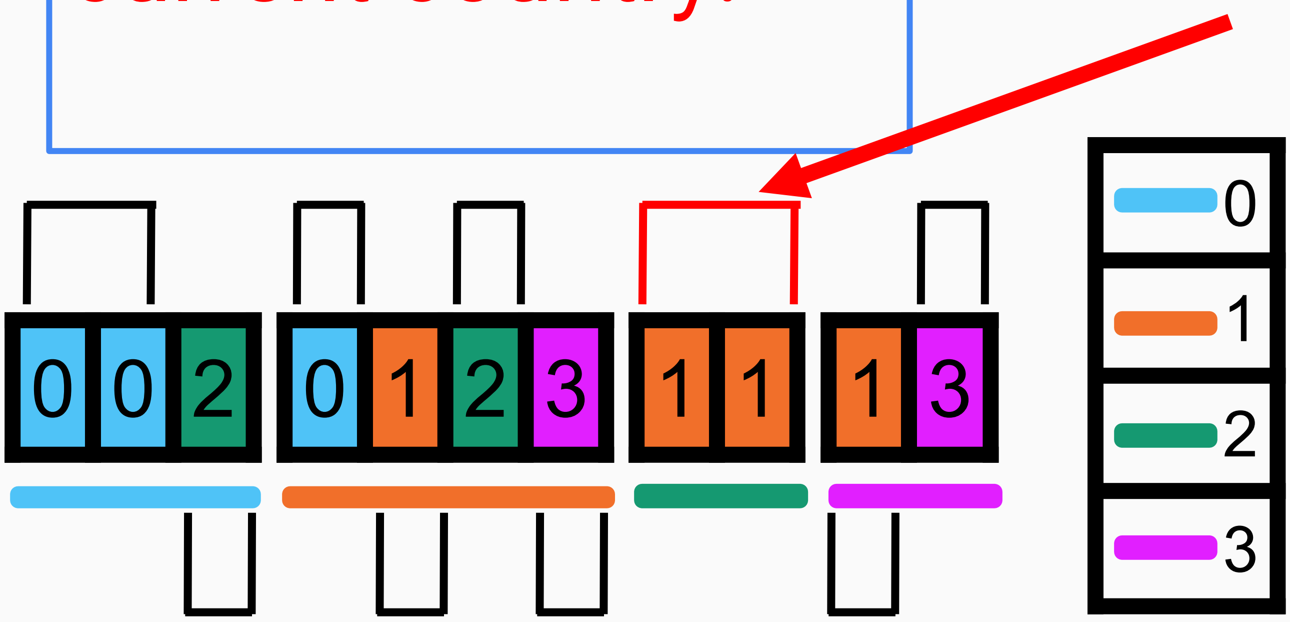
Regions Graph Building

Homogeneous sub-array: A subarray with the same digit

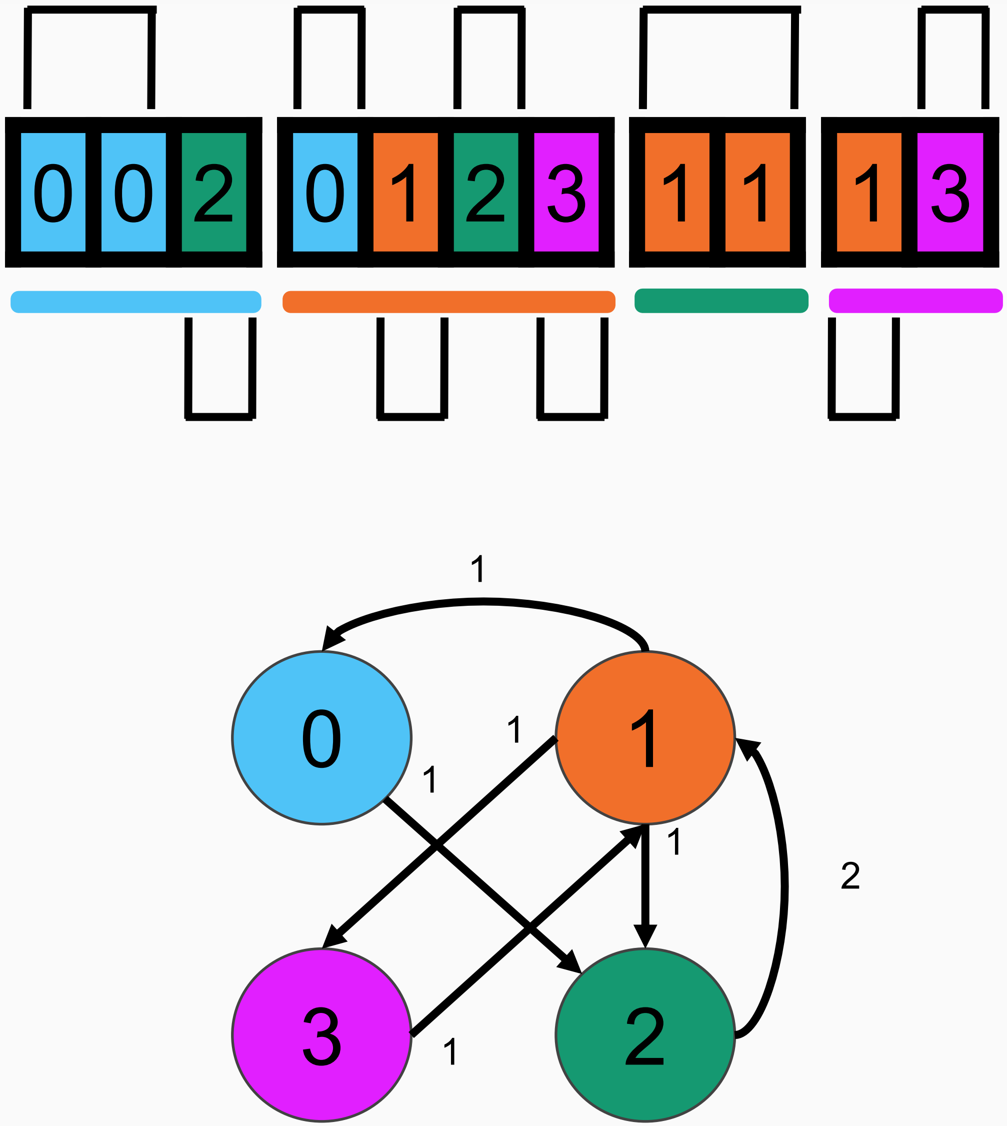


Regions Graph Building

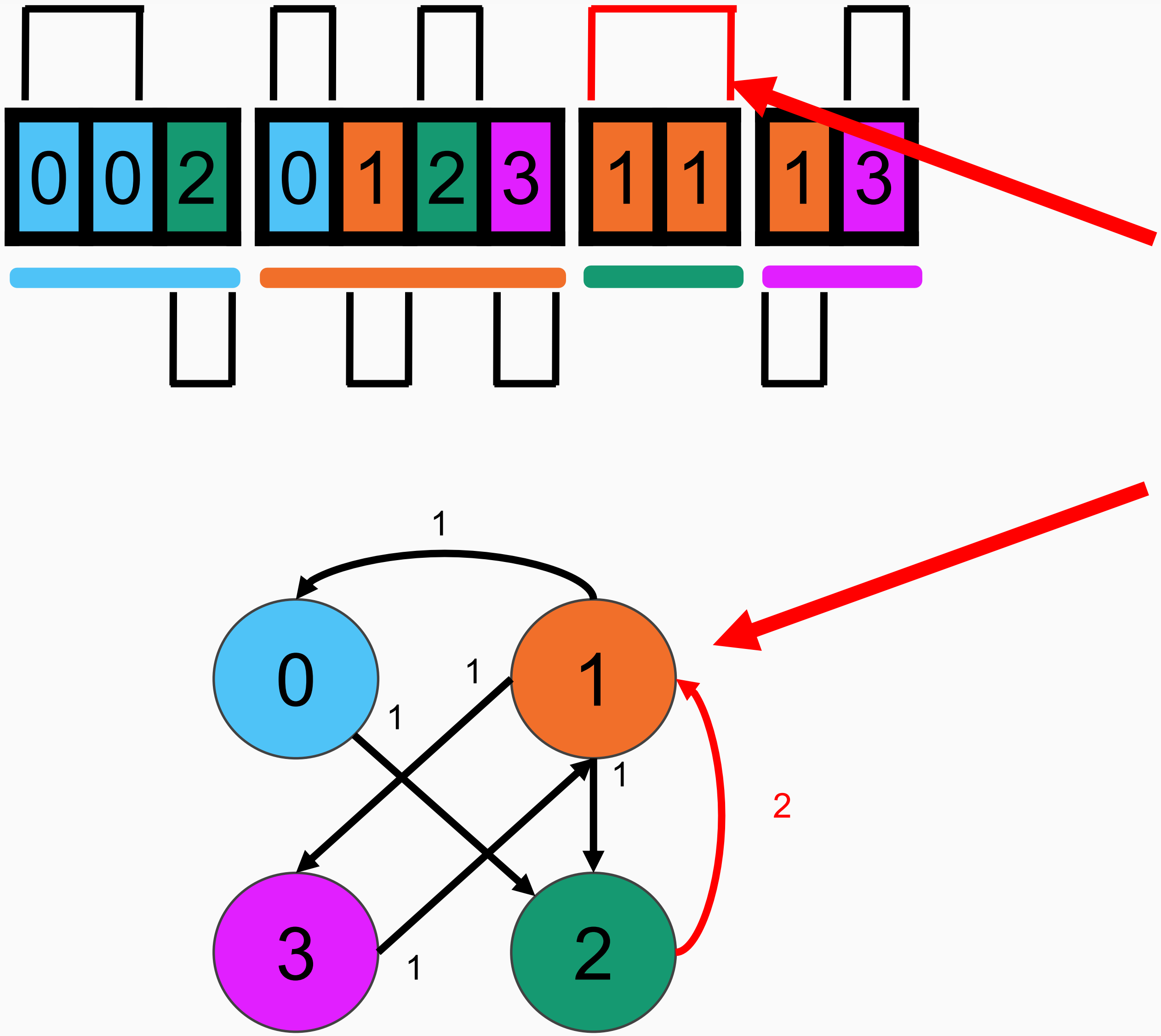
Region: A homogeneous sub-array within same current country.



Regions Graph Building



Regions Graph Building



Global Sorting

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

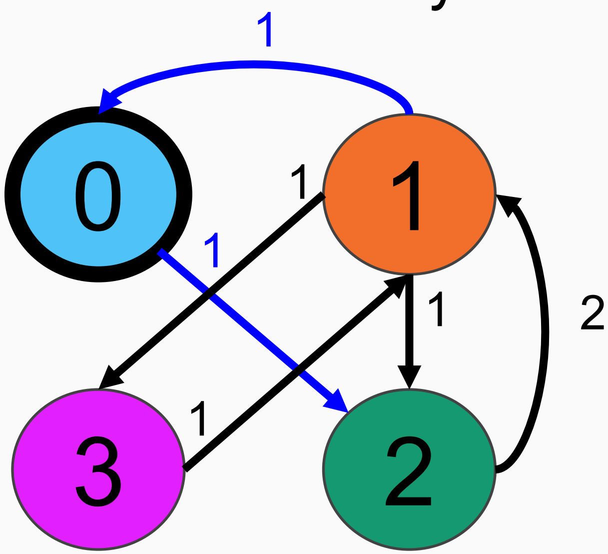
Two Approaches:

1. Cycle Finding
2. 2-Path Finding

Global Sorting

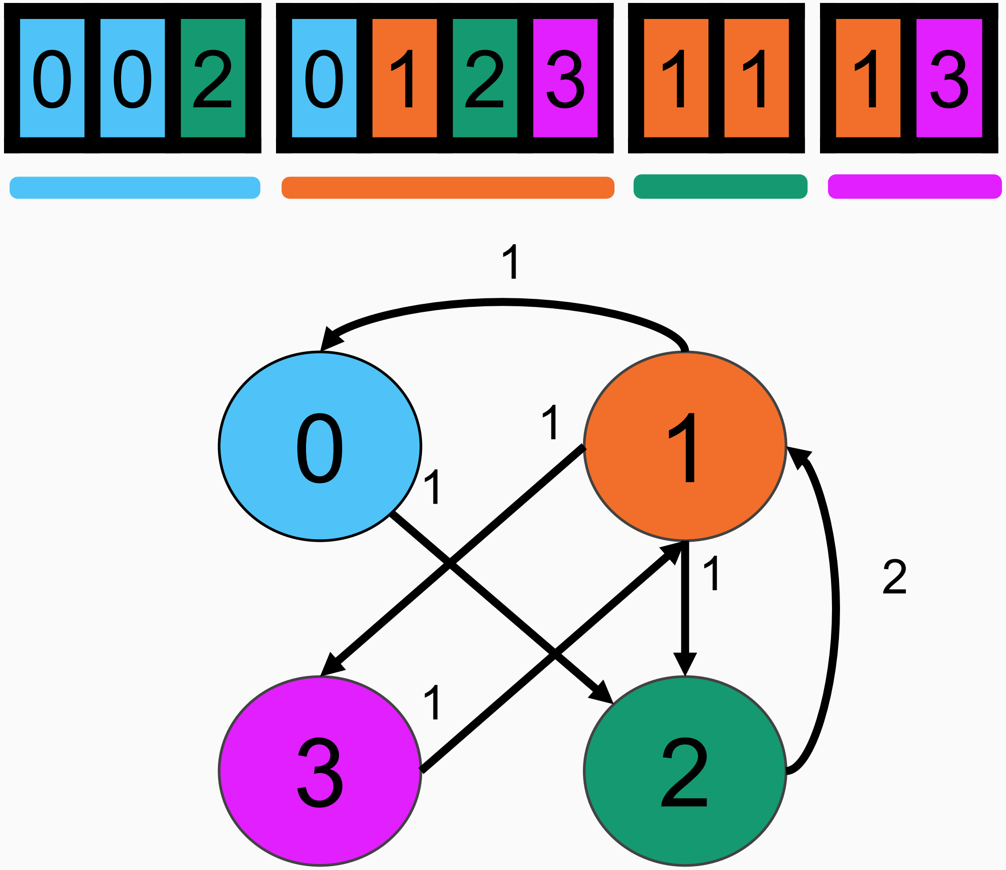
A 2-path consists of two edges:

- 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

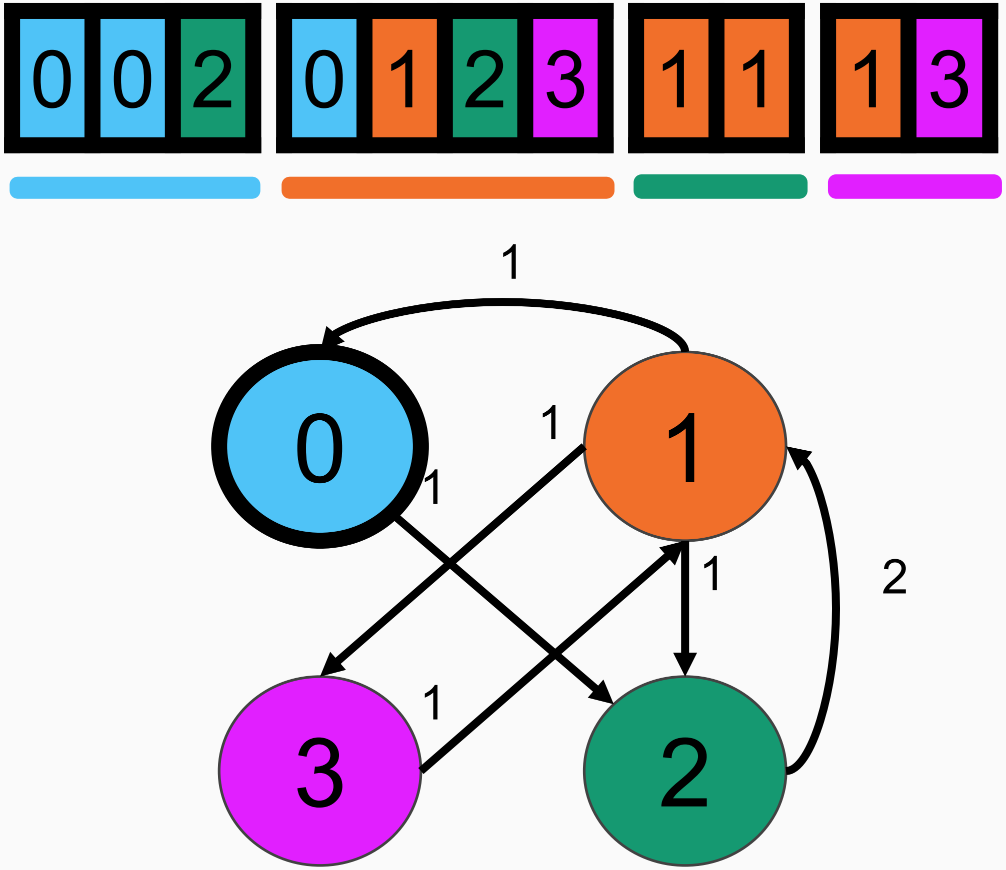
2-path Finding



Global Sorting: 2-Path Finding

2-path Finding

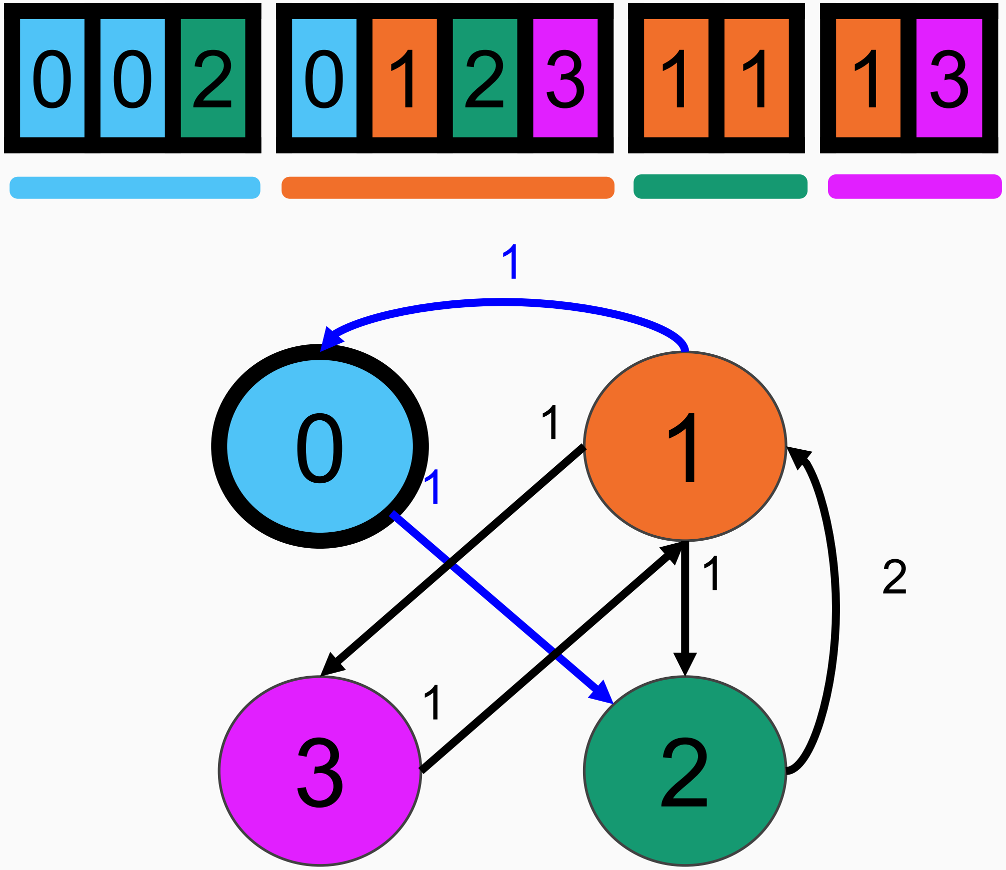
1. Choose a vertex.



Global Sorting: 2-Path Finding

2-path Finding

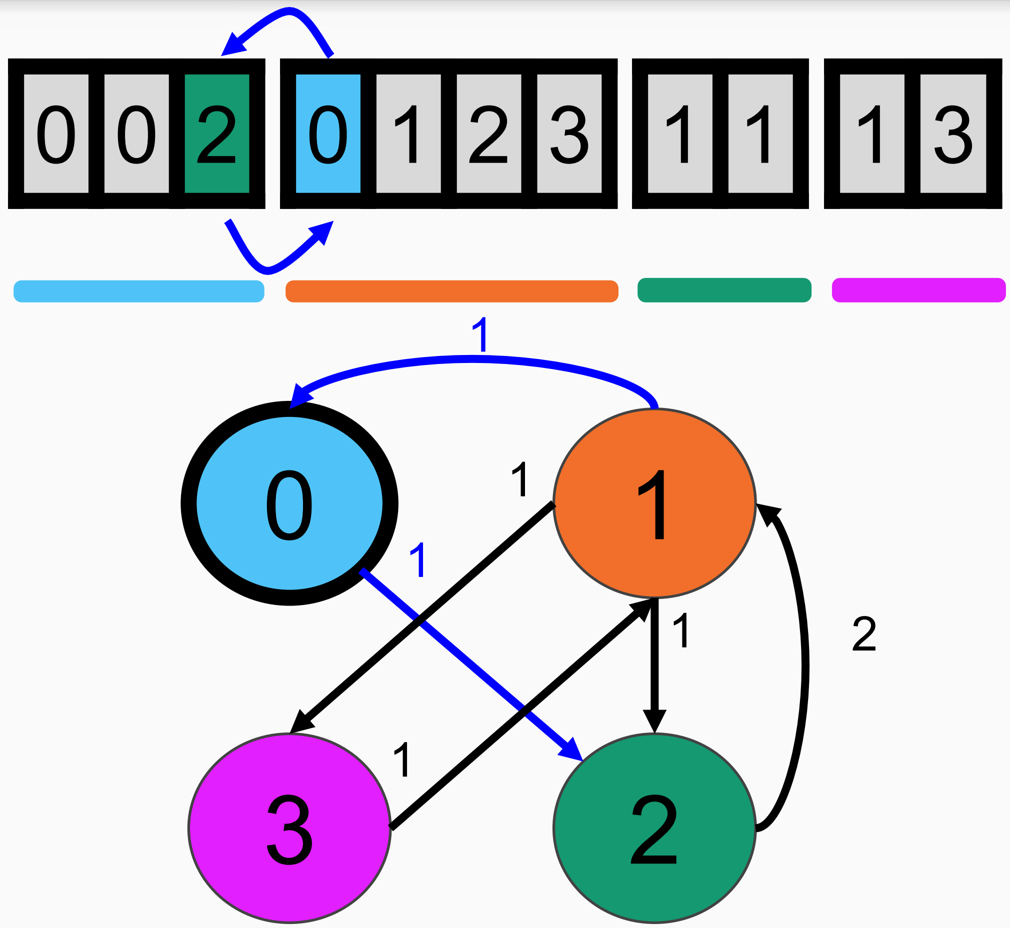
1. Choose a vertex.
2. Match incoming edges with outgoing edges.



Global Sorting: 2-Path Finding

2-path Finding

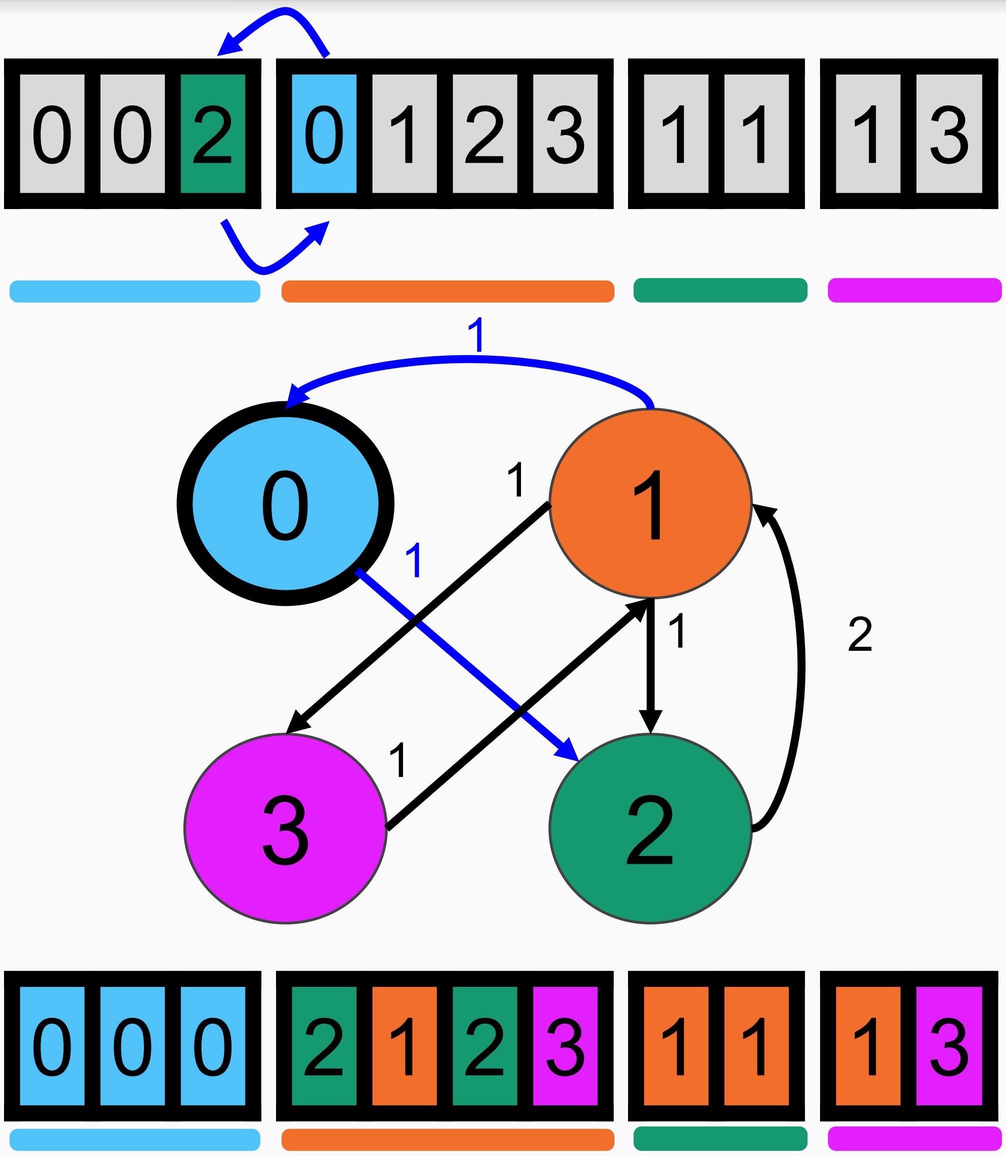
1. Choose a vertex.
2. Match incoming edges with outgoing edges.



Global Sorting: 2-Path Finding

2-path Finding

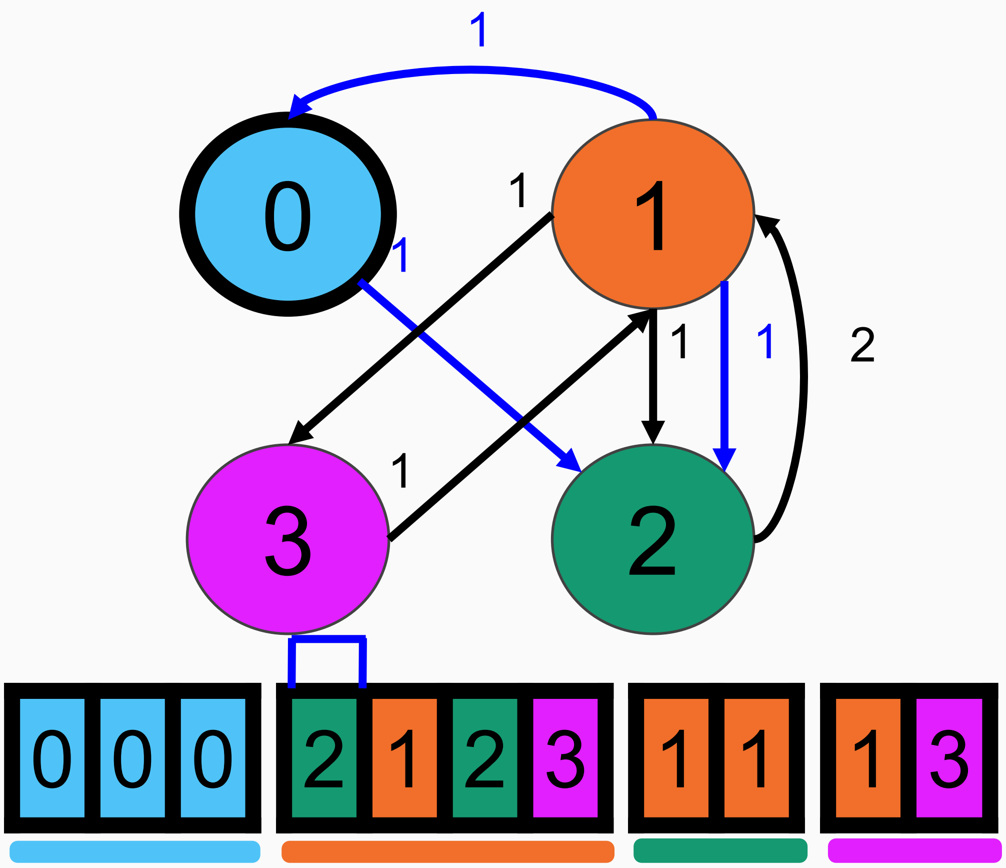
1. Choose a vertex.
2. Match incoming edges with outgoing edges.
3. Execute swaps.



Global Sort: 2-Path Finding

2-path Finding

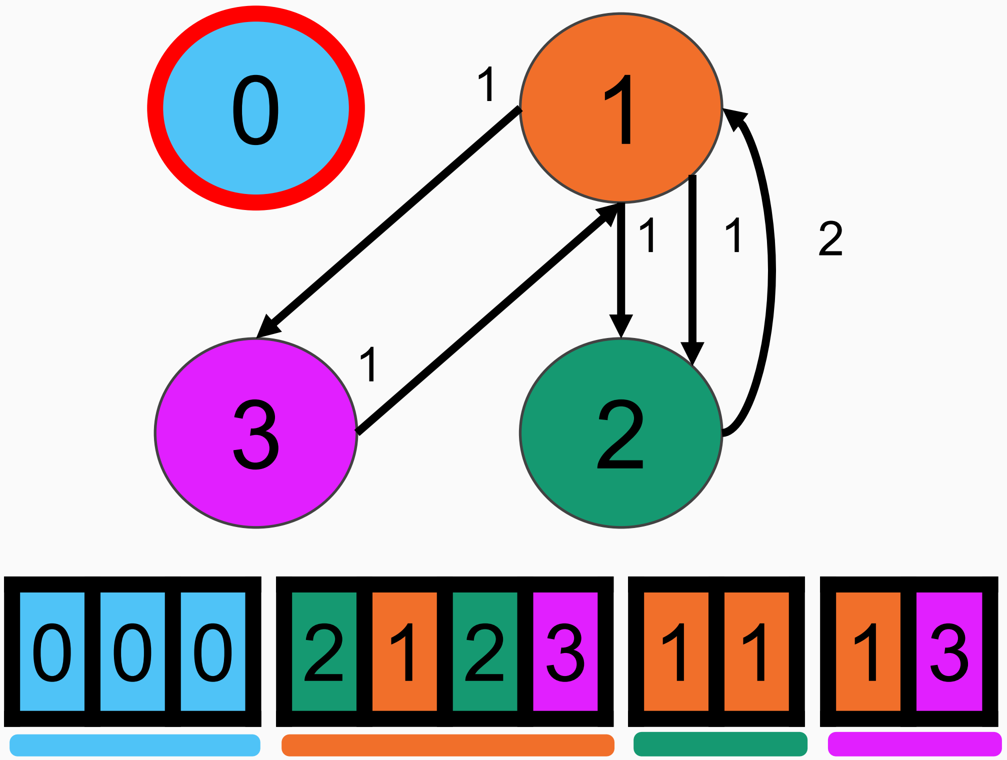
1. Choose a vertex.
2. Match incoming edges with outgoing edges.
3. Execute swaps.
4. Edit edges.



Global Sorting: 2-Path Finding

2-path Finding

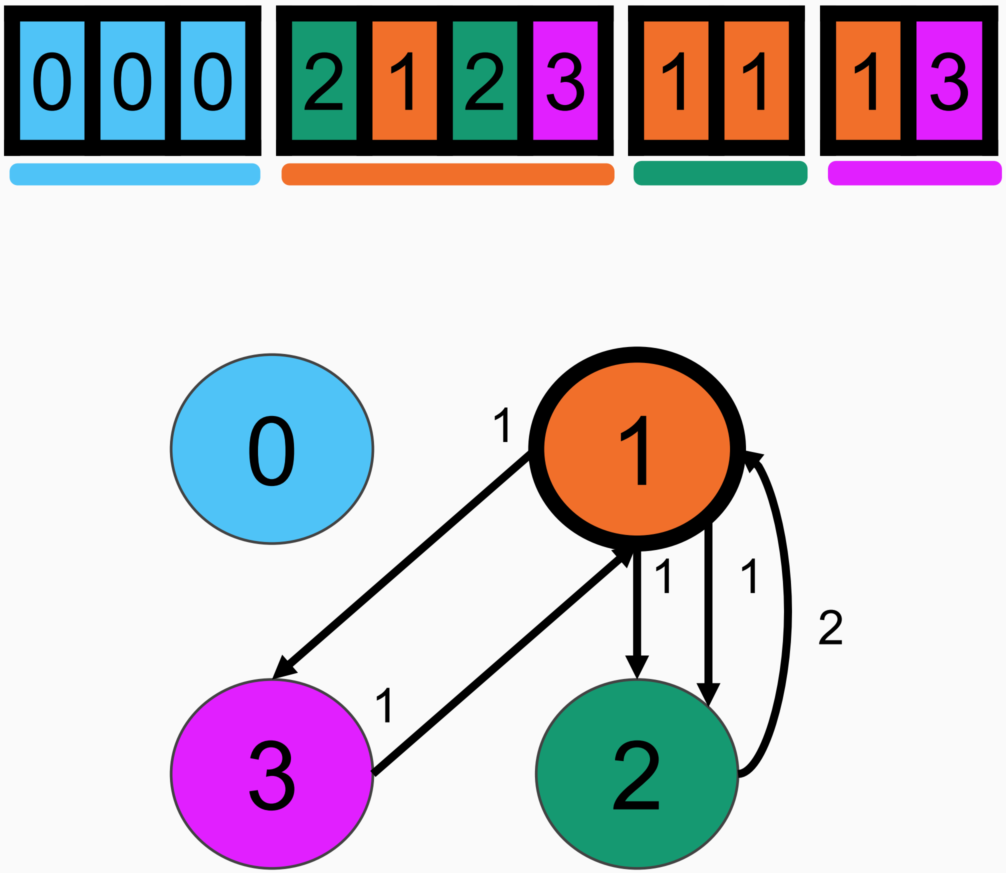
1. Choose a vertex.
2. Match incoming edges with outgoing edges.
3. Execute swaps.
4. Edit edges.



Global Sorting: 2-Path Finding

2-path Finding

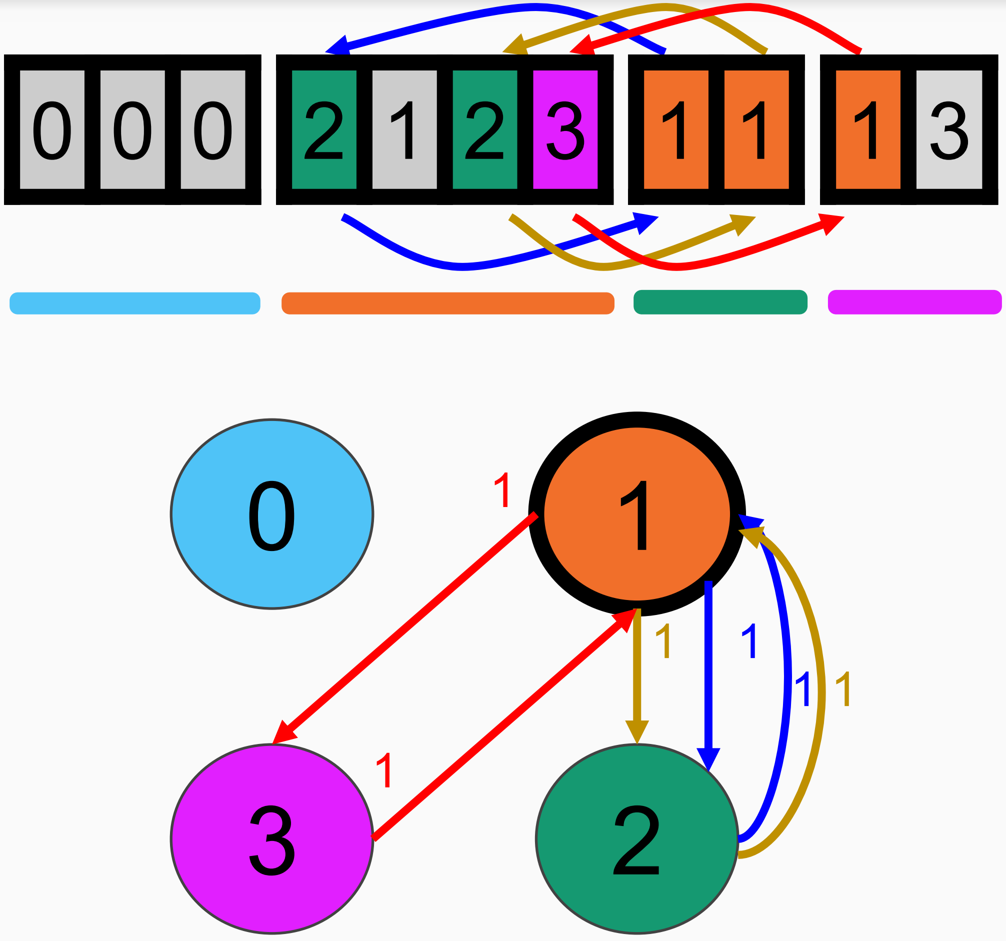
1. Choose a vertex.
2. Match incoming edges with outgoing edges.
3. Execute swaps.
4. Edit edges.



Global Sorting: 2-Path Finding

2-path Finding

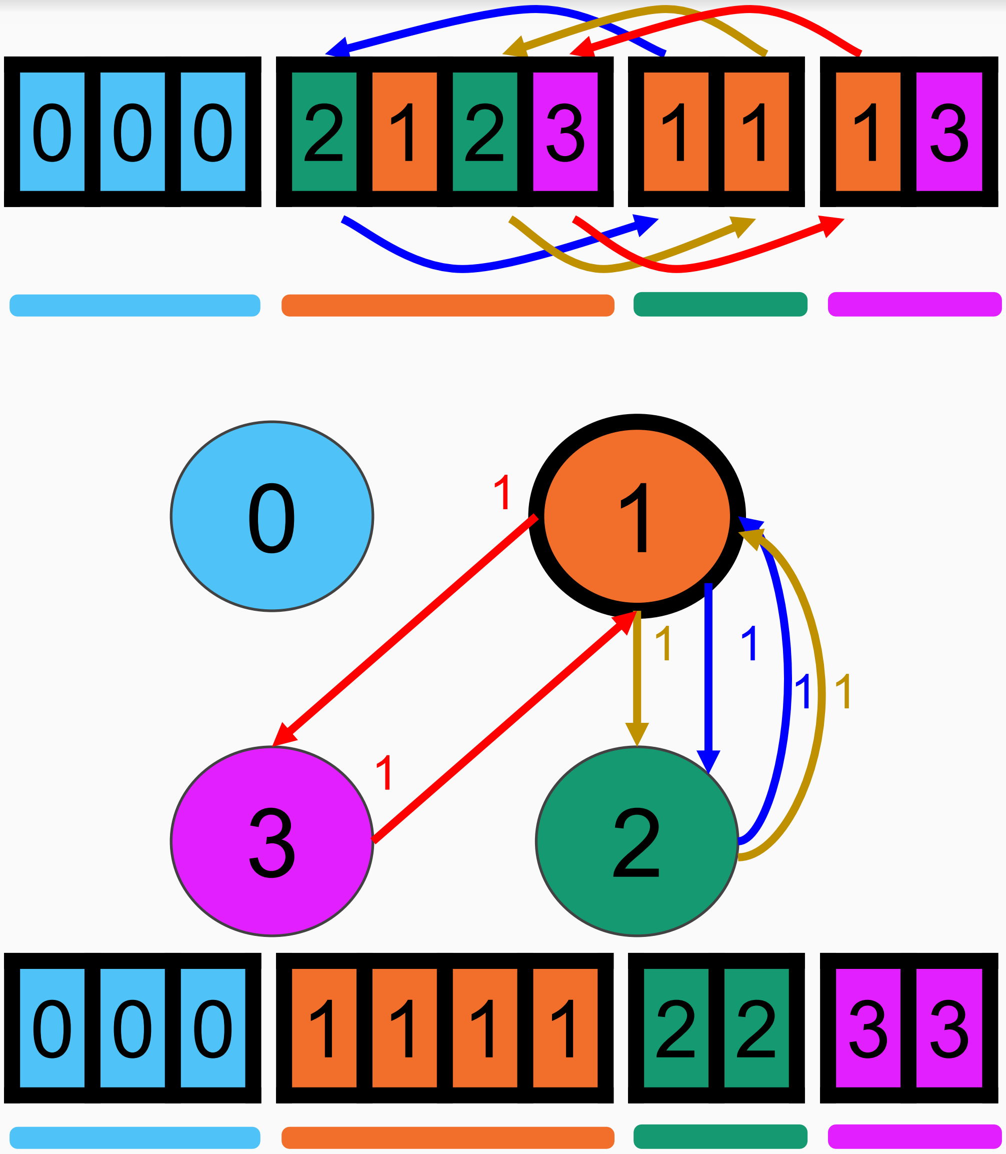
1. Choose a vertex.
2. Match incoming edges with outgoing edges.
3. Execute swaps.
4. Edit edges.



Global Sorting: 2-Path Finding

2-path Finding

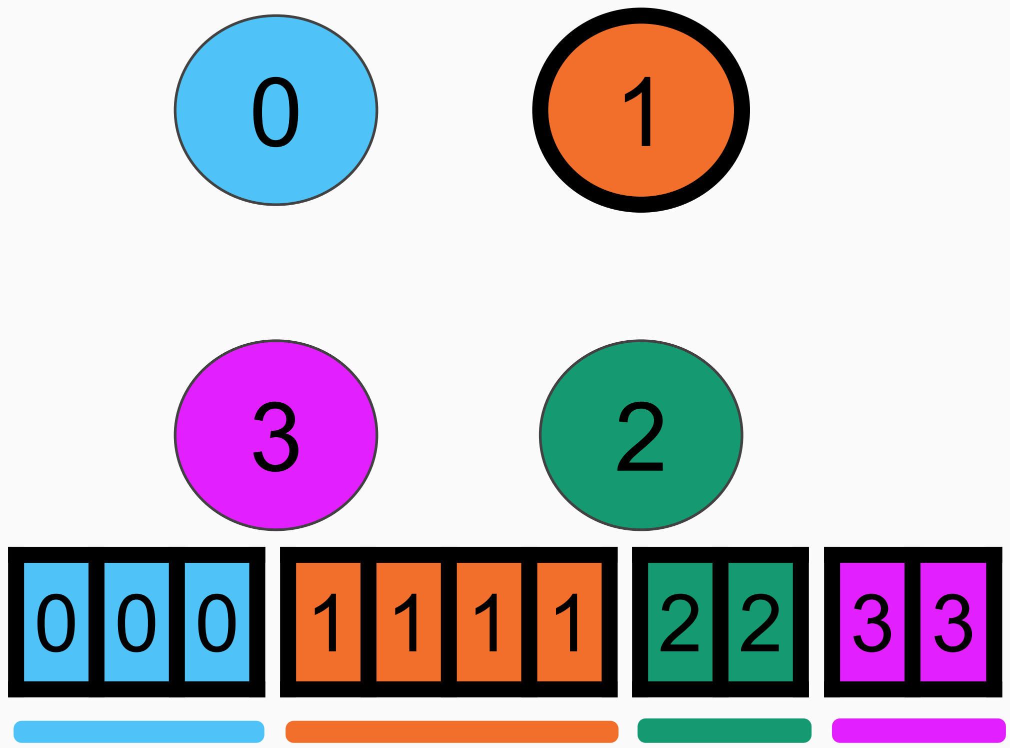
1. Choose a vertex.
2. Match incoming edges with outgoing edges.
3. Execute swaps.
4. Edit edges.



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(KBlog(n)).

- 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(KBlog(n))

- Since #regions = #edges = O(KB).
- K is number of blocks.
- B is number of buckets per block.

Analysis

3. Global Sorting

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

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

Analysis

Total for one level of recursion

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

- $B = \Theta(1)$
- $K = \Theta(P)$
- $KB = O(n)$

Algorithm: Recursion

Total on all levels

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

- $B = \Theta(1)$
- $K = \Theta(P)$
- $KB = O(n)$
- $r = \Theta(1)$ (fixed length integers)

Cycle Finding

- Find Cycle in Regions Graph
- Execute Cycle to move elements
- Remove edge with min weight
- Repeat until all edges are consumed

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 publically 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

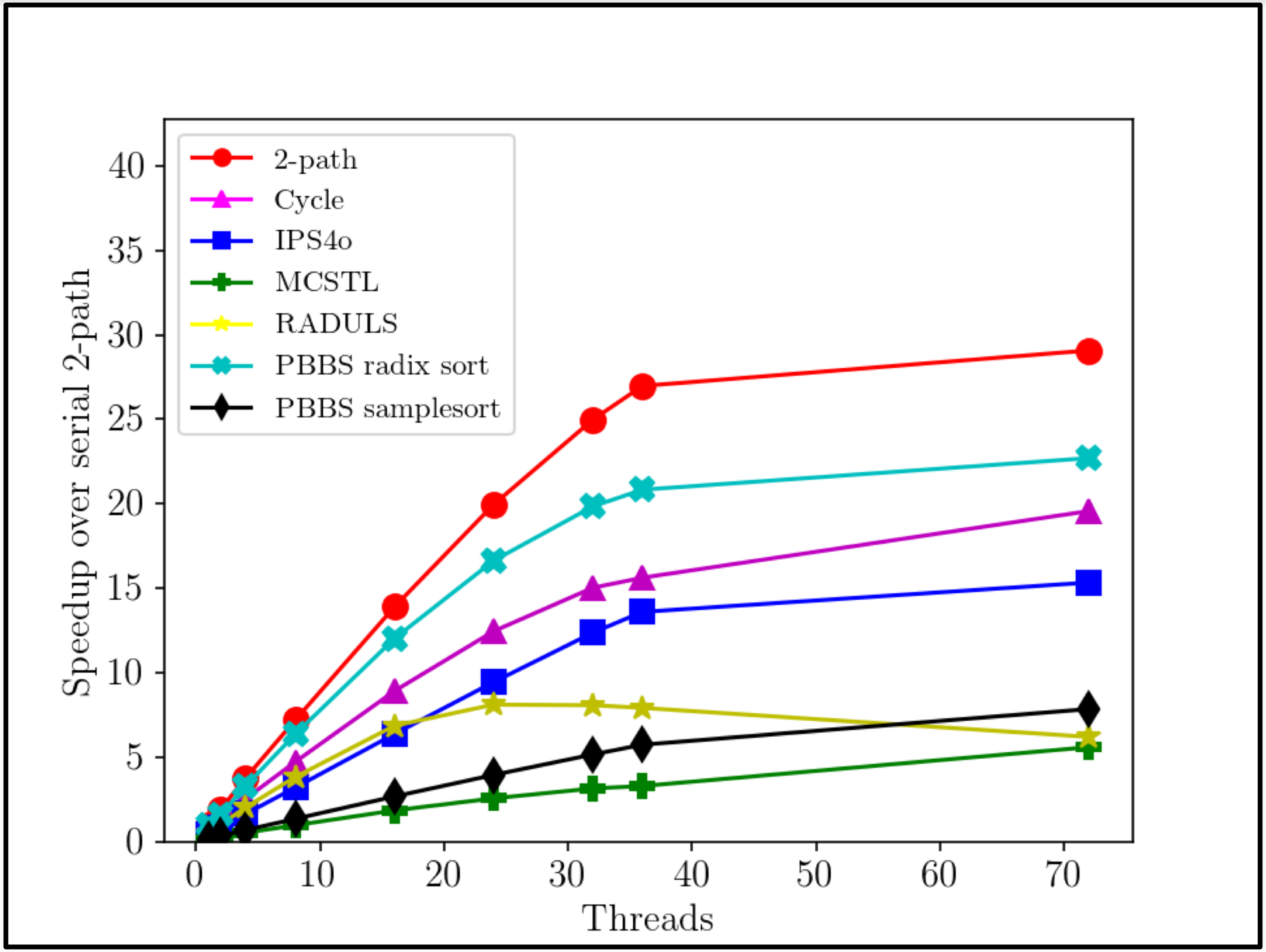
- AWS c5.9xlarge
- Intel Xeon Platinum 8000 series
- 72 vCPU (~36 cores with hyperthreading)
- 144 GB RAM
- All code compiled with g++-7 with Cilk Plus

Comparison with other algorithms

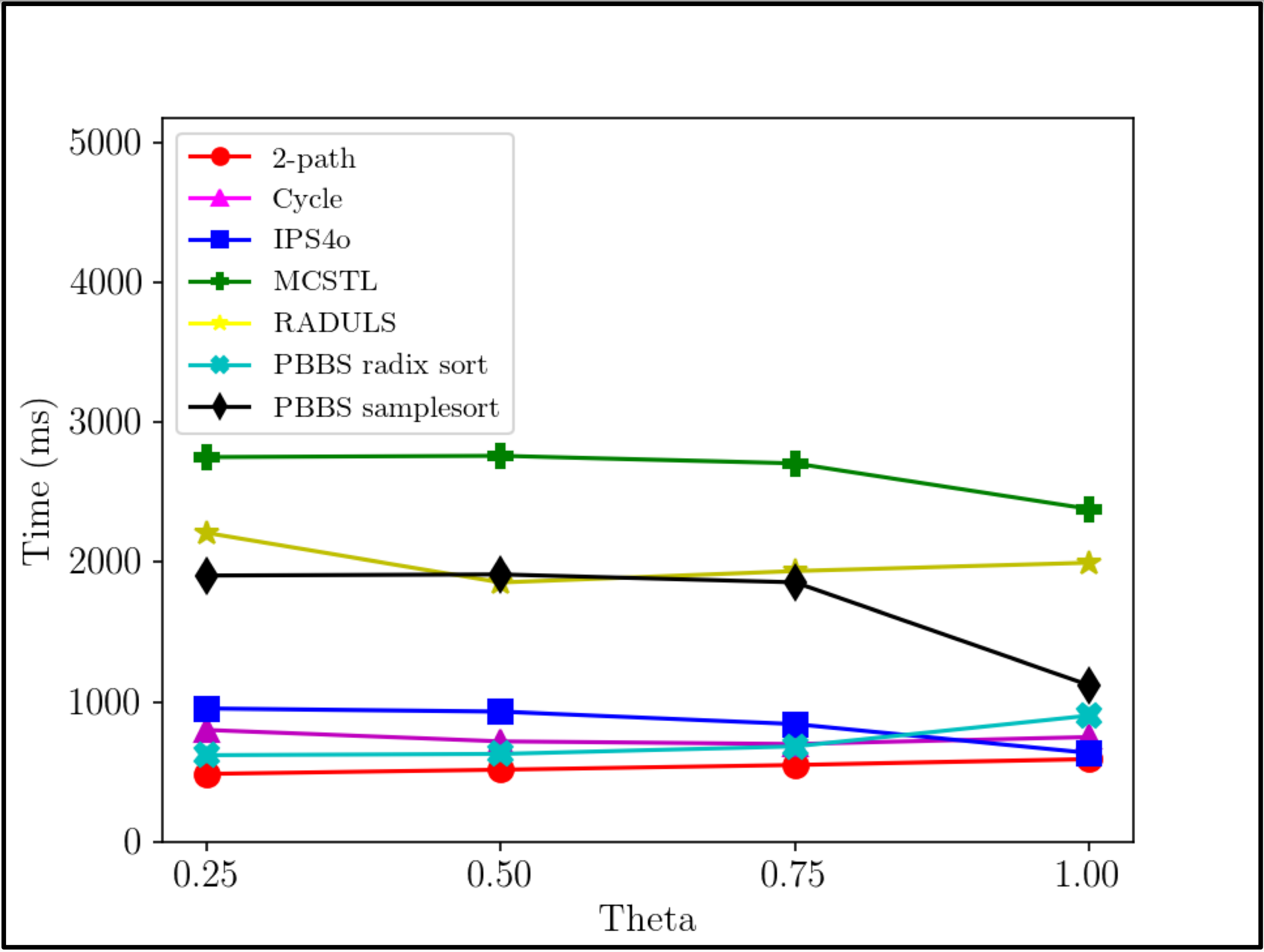
Regions Sort performance on various inputs with 1 billion integers:

- Between 1.1 - 3.6x faster than IPS4o, the fastest parallel sample sort, except on one input (1.02x slower).
- Between 1.2 - 4.4 x 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.

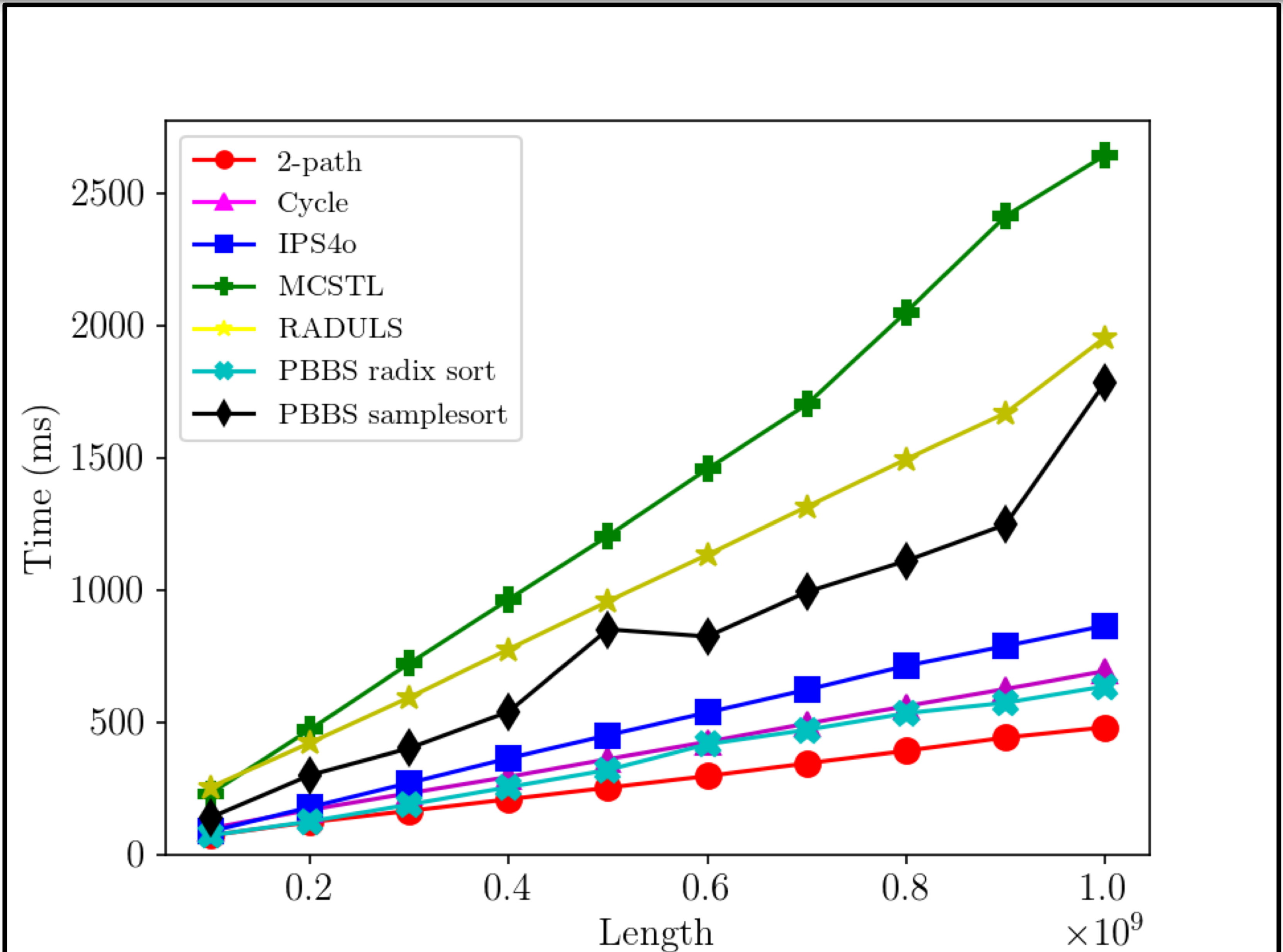
Speedup over serial 2-path: 1 billion random integers



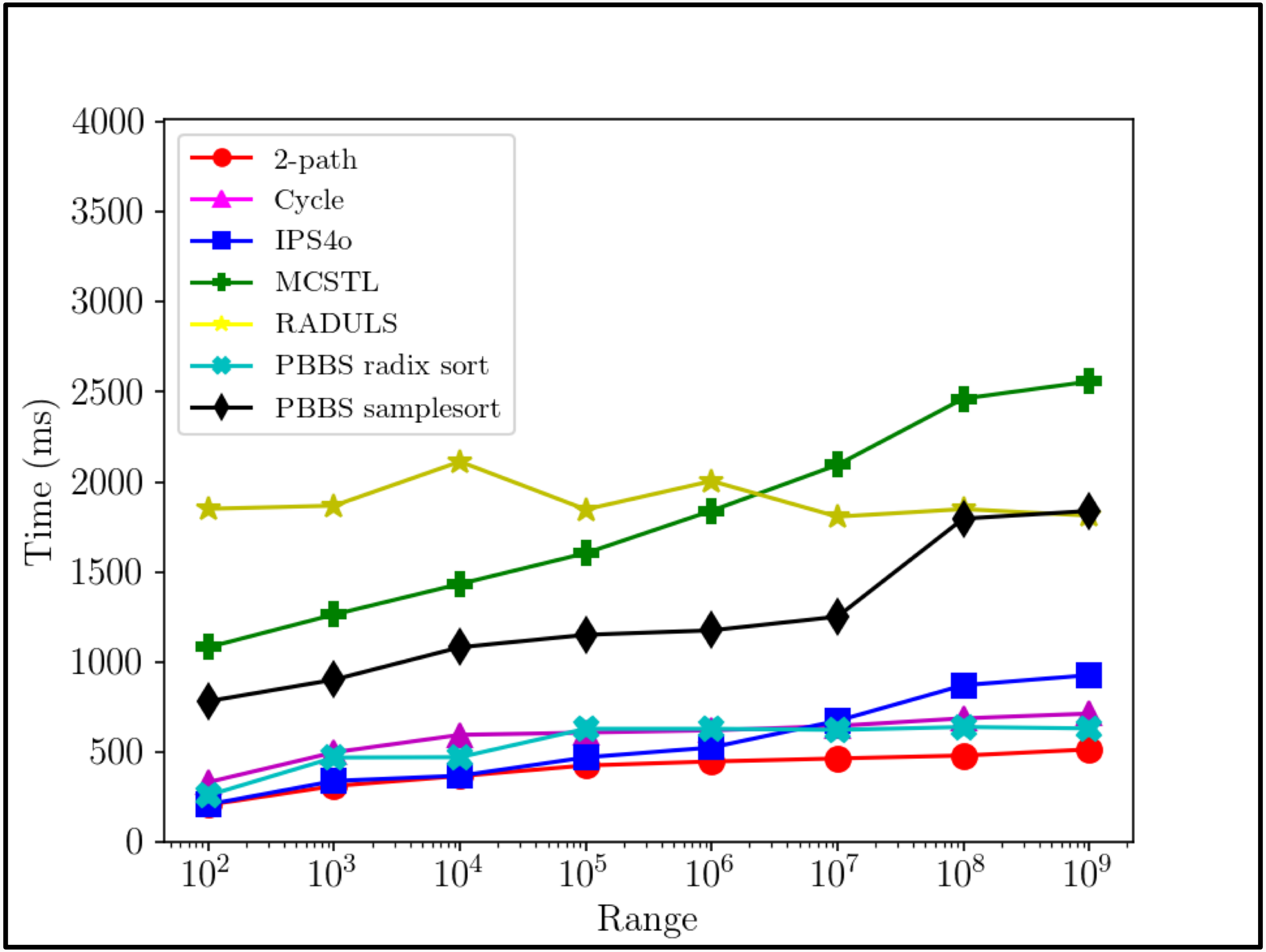
Distribution independence: 1 billion integers from Zipf



Regions Sort: fastest across all input sizes (Random)



Input Range - Uniform Sequence (1 billion integers)



Comments

Regions Sort is efficient:

- Since it needs at most 4n writes for local and global sort (per recursion level).
- The size of graph is asymptotically smaller than the input size.
- Has good temporal locality because we split the inputs to blocks that can more easily fit in

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 outperforms state of the art parallel sorting algorithms in our extensive experiments.

Regions Sort Code

https://github.com/omarobeya/parallel-inplace-radixsort

Questions?

Thank you!

References

- Cho, Minsik, Daniel Brand, Rajesh Bordawekar, Ulrich Finkler, Vincent Kulandaisamy, and Ruchir Puri. "PARADIS: an efficient parallel algorithm for in-place radix sort." Proceedings of the VLDB Endowment 8, no. 12 (2015): 1518-1529.

- Axtmann, Michael, Sascha Witt, Daniel Ferizovic, and Peter Sanders. "In-place Parallel Super Scalar Samplesort (IPS4o)." arXiv preprint arXiv:1705.02257 (2017).

- Kokot, Marek, Sebastian Deorowicz, and Agnieszka Debudaj-Grabysz. "Sorting data on ultra-large scale with RADULS." In International Conference: Beyond Databases, Architectures and Structures, pp. 235-245. Springer, Cham, 2017.

References

- Shun, Julian, Guy E. Blelloch, Jeremy T. Fineman, Phillip B. Gibbons, Aapo Kyrola, Harsha Vardhan Simhadri, and Kanat Tangwongsan. "Brief announcement: the problem based benchmark suite." In Proceedings of the twenty-fourth annual ACM symposium on Parallelism in algorithms and architectures, pp. 68-70. ACM, 2012.

- Singler, Johannes, Peter Sanders, and Felix Putze. "MCSTL: The multi-core standard template library." In European Conference on Parallel Processing, pp. 682-694. Springer, Berlin, Heidelberg, 2007.

- Malte Skarupke. 2016. I Wrote a Faster Sorting Algorithm. https://probablydance.com/2016/12/27/i-wrote-a-faster-sorting-algorithm/.