Parallel Integer Sort: Theory and Practice

Xiaojun Dong[1], Laxman Dhulipala[2], Yan Gu[1], Yihan Sun[1]

Full version: https://arxiv.org/abs/2401.00710

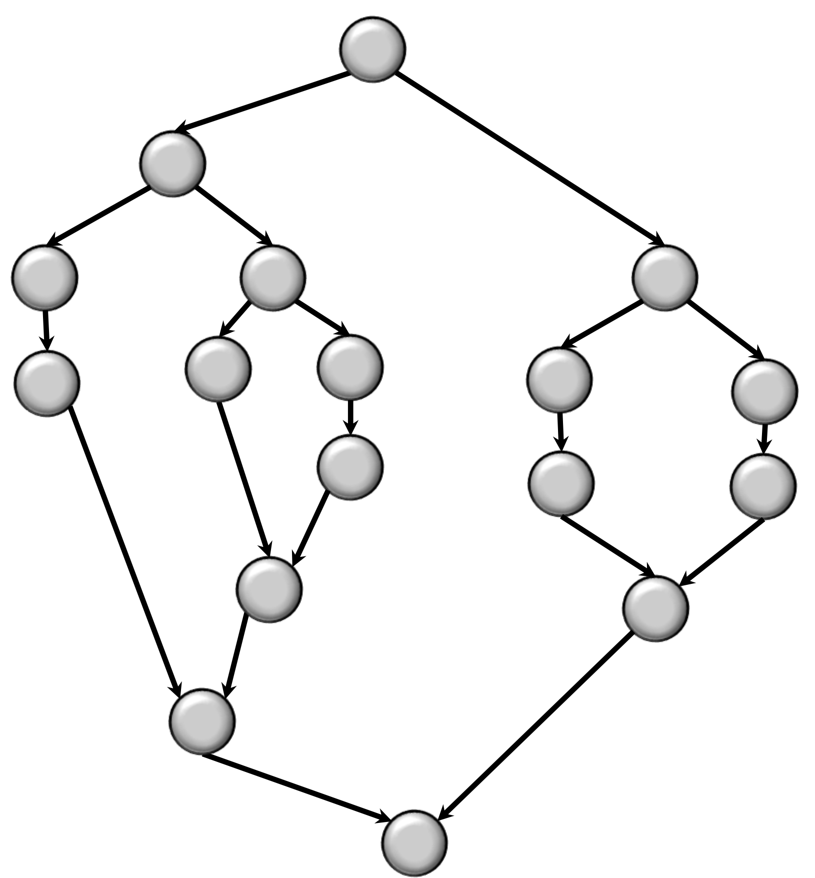
Code: https://github.com/ucrparlay/DovetailSort

[1]

[2]

Computational Model

- Shared-memory multi-core setting
- Work-span model
- Work: the total number of operations
- Span (depth): the length of the longest dependency chain



Sorting implementations are integrated in most standard libraries

C++

```cpp
#include <algorithm>
#include <execution>
#include <vector>

int main() {
  int arr[] = {3, 1, 4, 2};
  std::sort(arr, arr+4);
  return 0;
}
```

Java

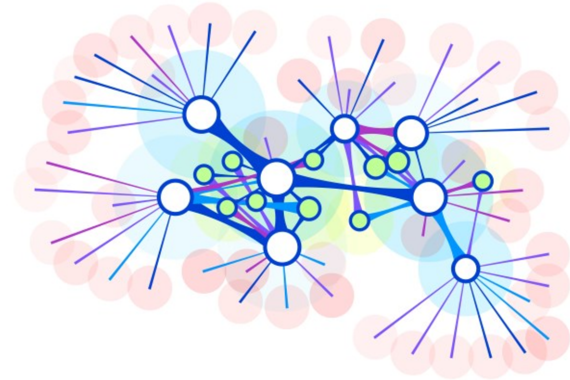
```java
import java.util.Arrays;

public class Main {
  public static void main(String[] args) {
    int numbers[] = {3, 1, 4, 2};
    Arrays.sort(numbers);
  }
}
```

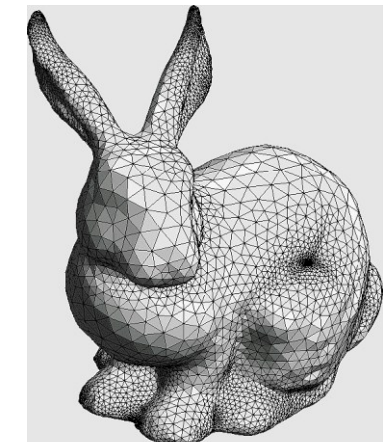
- Possible data types to sort: integers, floating-point numbers, strings, …

Keys are integers in many use cases

- Graph processing
  - Edges and vertices are represented as integers



- Geometry processing
- Pointer sorting
- Timestamps sorting



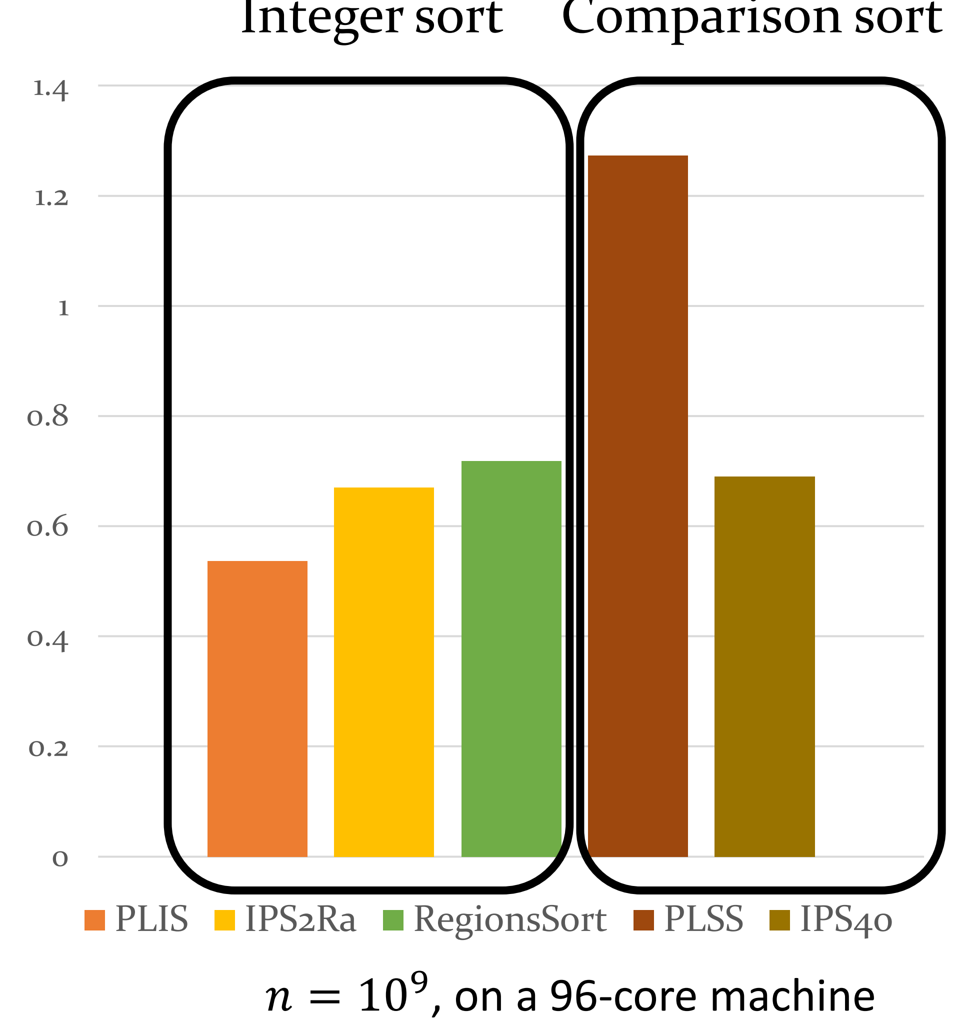
[1]: https://www.datanami.com/2017/10/02/enterprise-knowledge-graphs-need-semantics/
[2]: https://community.cadence.com/cadence_blogs_8/b/cfd/posts/fidelity-pointwise-andle-both-analytic-and-discrete-geometry-types
[3]: https://blog.teamnexus.in/blog/2022/06/23/database-of-databases/

Integer sort can be faster than comparison sort

- In theory
  - Comparison sort has a $\Omega(n \log n)$ lower bound
  - Integer sort can achieve $O(n+r)$ or $O\left(n\sqrt{\log\log r}\right)$ for key range $[0, r]$ in sequential
- In practice
  - Integer sort is faster than comparison sort

Integer sort can be practically faster than comparison sort

- Bars are the running times on a *uniform distribution without duplicates*
- Lower is better



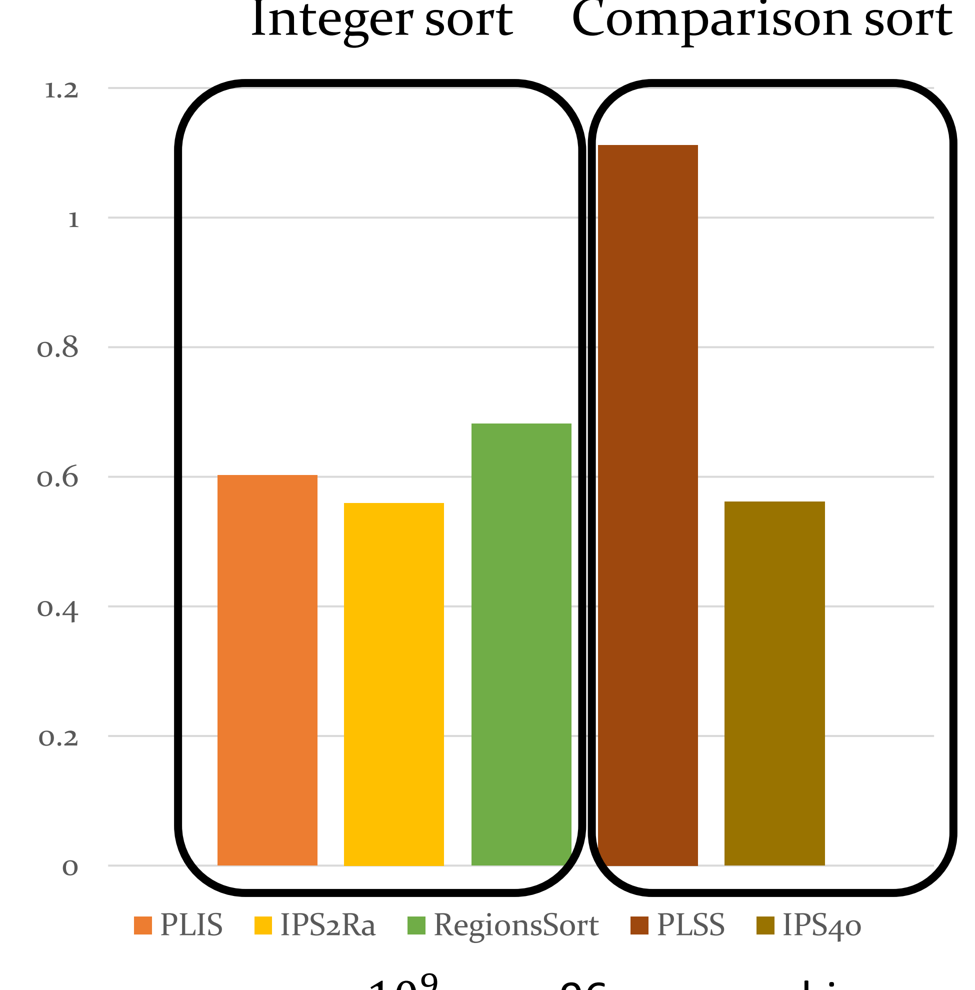
Is integer sort faster than comparison sort in parallel?

- Challenge in theory
  - Despite some theoretical studies [AH97,AHNR98]
  - The best work bound for practical implementations for the range of $[0, r]$: $O(n \log r)$ [OKFS19]
  - No better than comparison sort when $r = \Omega(n)$

- Can we theoretically explain, why integer sort is practically faster than comparison sort?

Existing integer sort algorithms can be slow on heavy duplicates

- Bars are the running times on a *uniform distribution with heavy duplicates* (10 different keys)
- Lower is better



$n = 10^9$, on a 96-core machine

Our contributions

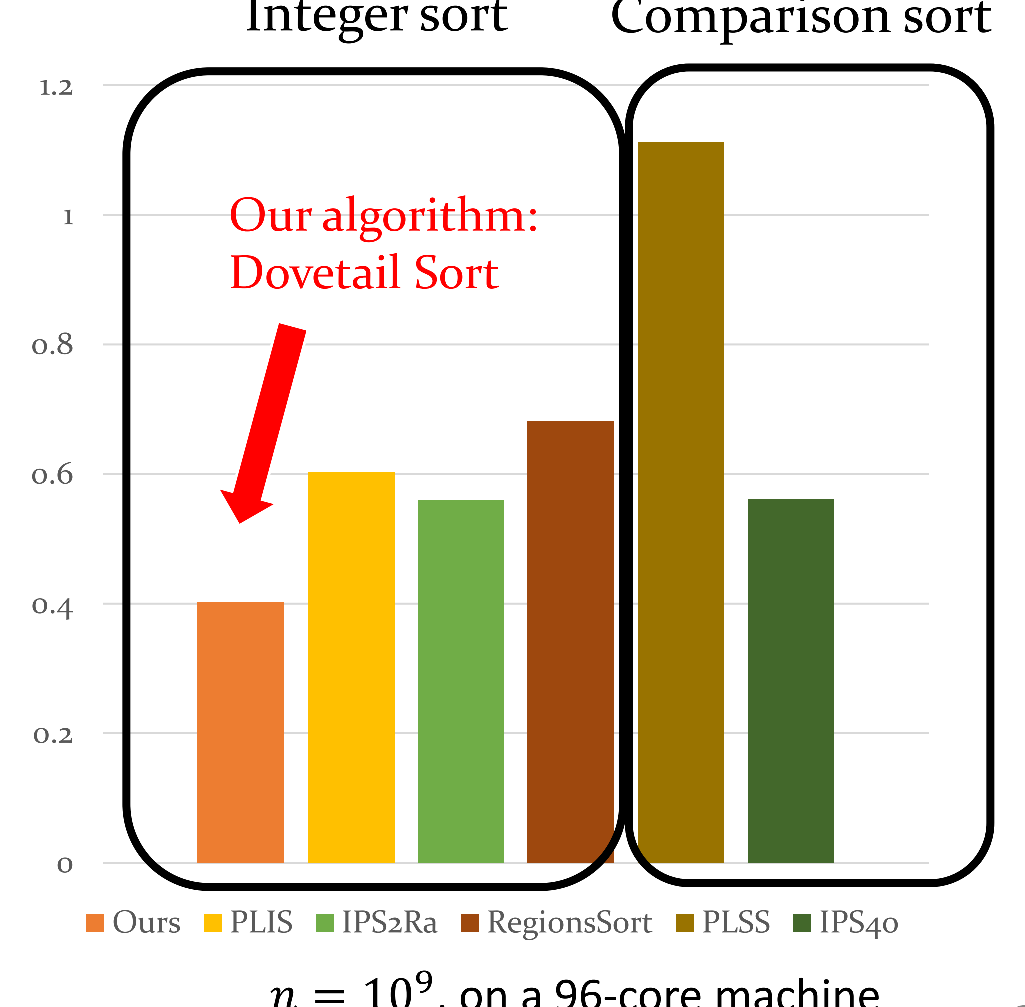
- Theoretical challenge: Can we theoretically explain, why integer sort is practically faster than comparison sort?
- We proved that a class of practical parallel integer sort implementations, including our new one, have $O\left(n\sqrt{\log r}\right)$ work

- Practically challenge: Can integer sort consistently outperform comparison sort (particularly with heavy duplicates)?
- We proposed DovetailSort that combines the advantages of integer and sample sort and is consistently faster than all existing algorithms in most tested cases

Our algorithm is impervious to heavy duplicates!

- Bars are the running times on a *uniform distribution with heavy duplicates*

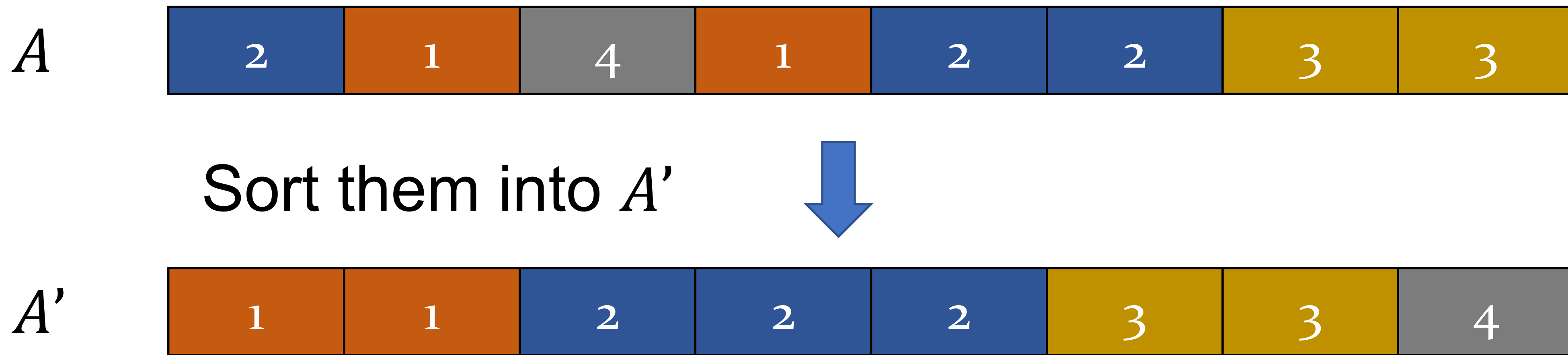
- Our algorithm achieves the best performance, which is 39% faster than the best baseline



$n = 10^9$, on a 96-core machine

Integer sort definition

- Input: a sequence of integers A of size n in the range $[0, r]$
- Output: records that are reordered by the keys



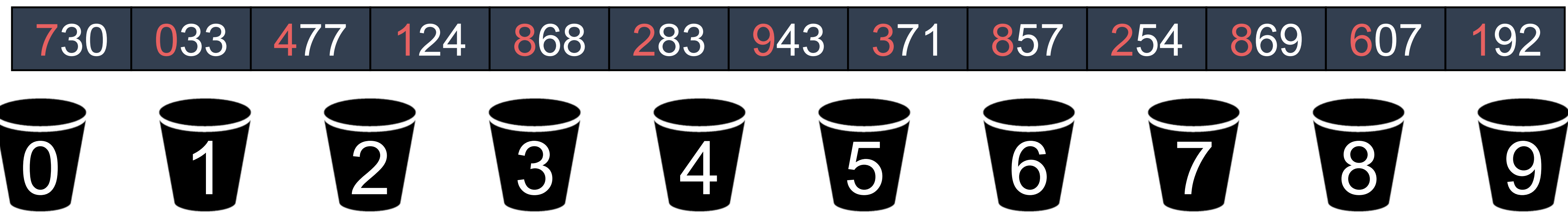
Types of integer sort algorithms

- Counting sort / Pigeonhole sort
  - Has $\tilde{O}(n + r)$ work and $O(\log n)$ span theoretically [RR89]
  - Impractical for large key range
- Radix Sort
  - Favorable for large key range
  - Recursive structure, and use counting sort as a subroutine
  - Least significant digit (LSD)
  - Most significant digit (MSD)
    - Divide-and-conquer and amenable to good parallelism

Most Significant Digit (MSD) Sort

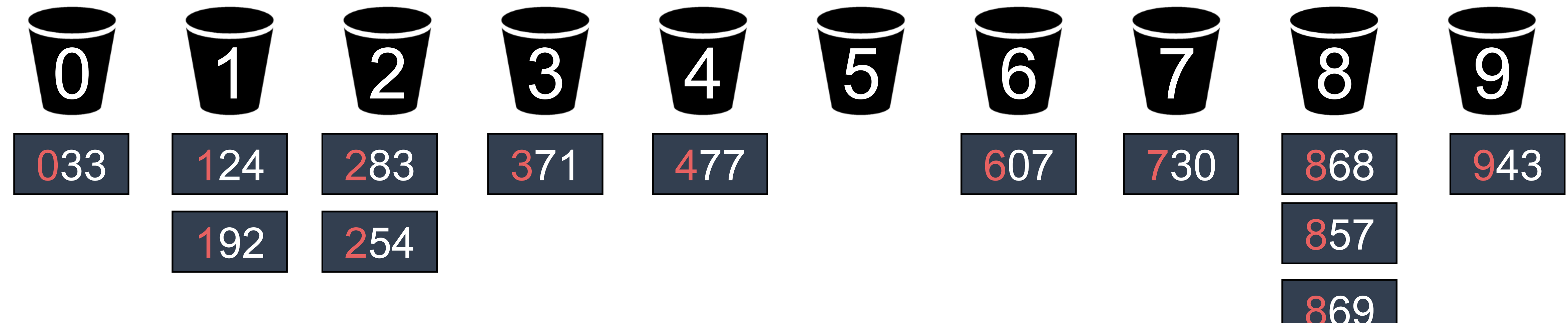
Most Significant Digit (MSD) Sort

- Counting sort all numbers based on the most significant digit



Most Significant Digit (MSD) Sort

- Counting sort all numbers based on the most significant digit
- Within each bucket, counting sort the next significant digit

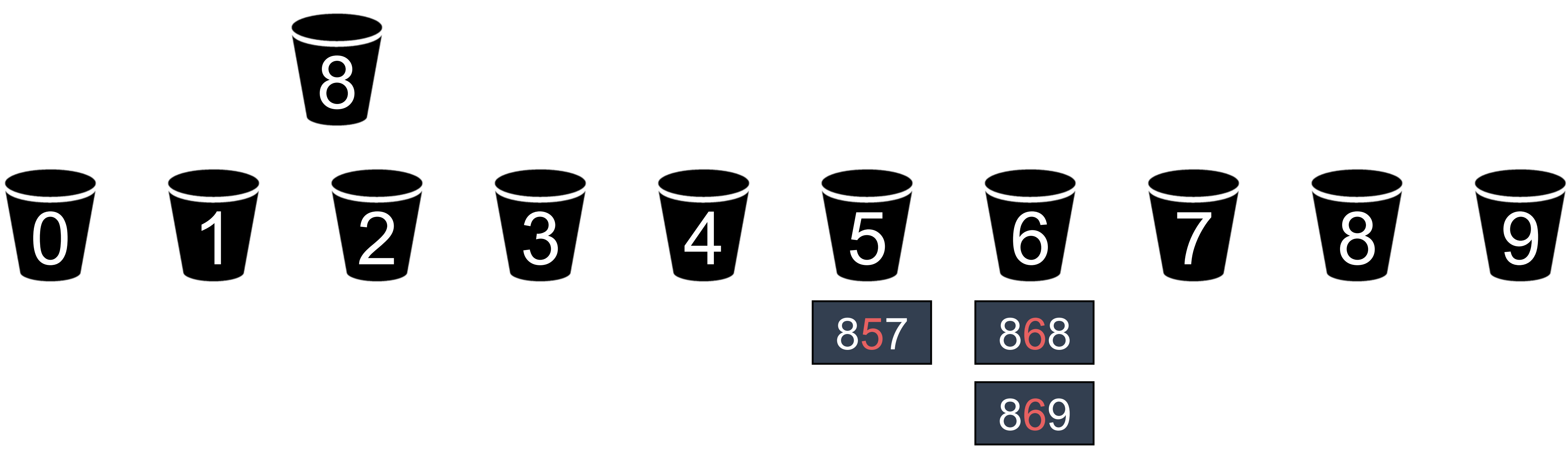


Most Significant Digit (MSD) Sort

- Counting sort all numbers based on the most significant digit
- Within each bucket, counting sort the next significant digit

Most Significant Digit (MSD) Sort

- Counting sort all numbers based on the most significant digit
- Within each bucket, counting sort the next significant digit



Most Significant Digit (MSD) Sort

- Counting sort all numbers based on the most significant digit
- Within each bucket, counting sort the next significant digit

Most Significant Digit (MSD) Sort

- Counting sort all numbers based on the most significant digit
- Within each bucket, counting sort the next significant digit
- Recurse until every digit is sorted. Pack the elements back

Most Significant Digit (MSD) Sort

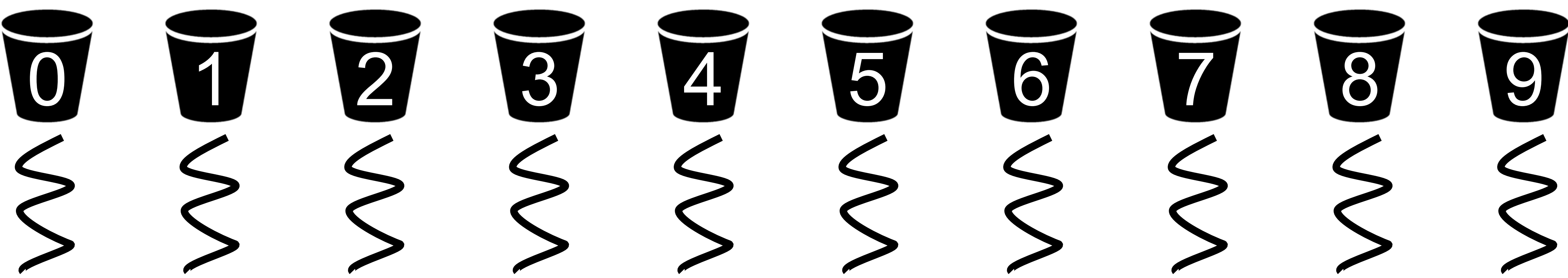
- Counting sort all numbers based on the most significant digit
- Within each bucket, counting sort the next significant digit
- Recurse until every digit is sorted. Pack the elements back

033	124	192	254	283	371	477	607	730	857	868	869	943

0 1 2 3 4 5 6 7 8 9

MSD sort is amenable to good parallelism

- Once the keys are distributed into the corresponding buckets, they can be process recursively in parallel



Every subproblem can be solved individually in parallel

“Digits” are in binary practically

- Rather than using radix 10, each digit is represented using γ bits
- For example, if $\gamma = 4$, we have $2^4 = 16$ buckets
- Practical algorithms pick $\gamma = 8$ with 256 buckets

Parallel MSD sort pseudocode

Base Cases

Distributing

Recursing

$PMSDSort(A[0..n-1], d)$

if $d = 0$ **then return** A

if $|A| < \theta$ **then return** $ComparisonSort(A)$

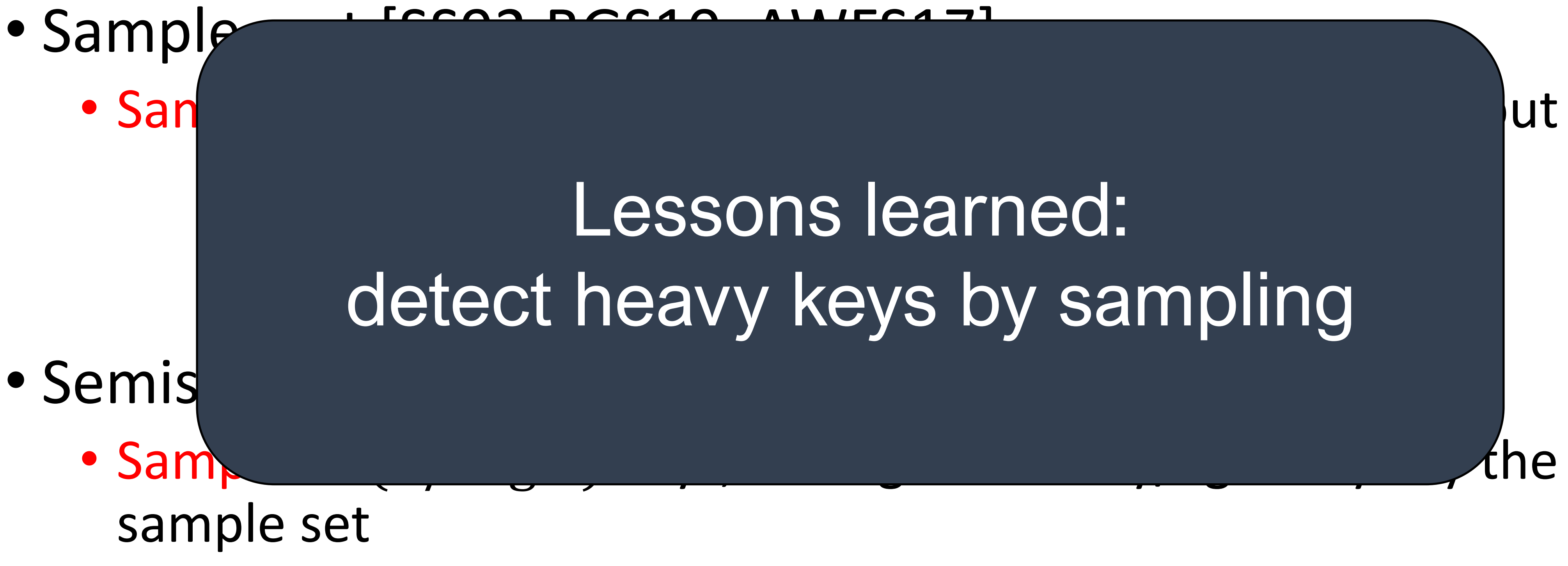
Distribute each integer to buckets using the d-th digit as the bucket id

Parallel_for_each bucket $\boldsymbol{B}$ $PMSDSort(B, d-1)$

MSD sort is susceptible to heavy duplicates

- Most numbers are the same with some outliers
  - Most numbers will be distributed to bucket 0 for the first two digits
  - Can we save some distribution costs by identifying and taking advantage of heavy duplicates?

How do other sorting algorithms process duplicates?



sample set

How to integrate heavy keys detection into integer sort?

- Challenge: detecting heavy keys is complicated for integer sort
  - Integer sort: input range is divided based on digits, and all records need to be fully sorted by the keys
- Our technique: interweave the light and heavy keys by **dovetailing**

Our DovetailSort Algorithm

DovetailSort: MSD framework

- Our algorithm also follows the MSD radix sort framework

- It recursively sorts by the most significant digit, until the base cases that
    1) All bits all sorted: all keys have been fully sorted
    2) The input size is small enough: use comparison sort

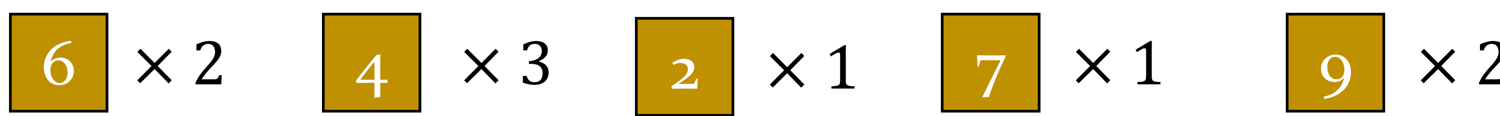
- Two extra steps:
    - Sampling: detect heavy keys
    - Dovetailing: merge heavy keys with light keys

DovetailSort: sampling

- In each recursion, we first use sampling to detect heavy keys as in sample sort and semisort

DovetailSort: sampling

- We take $O(2^\gamma \log n)$ samples and count the frequencies



- Keys with more than $\log n$ frequencies are heavy keys

- Each heavy key owns a separate heavy bucket
- Multiple light keys share will share one light bucket based on its digit

6	4	0	4	8	2	6	4	7	9	11	5	15	4	13	10	9	4	14	5	9	11	6	9

DovetailSort: distributing

- Every key belongs to an MSD zone based on its digit
  - MSD zone 00: [0,4)
  - MSD zone 01: [4,8)
  - MSD zone 10: [8,12)
  - MSD zone 11: [12, 16)

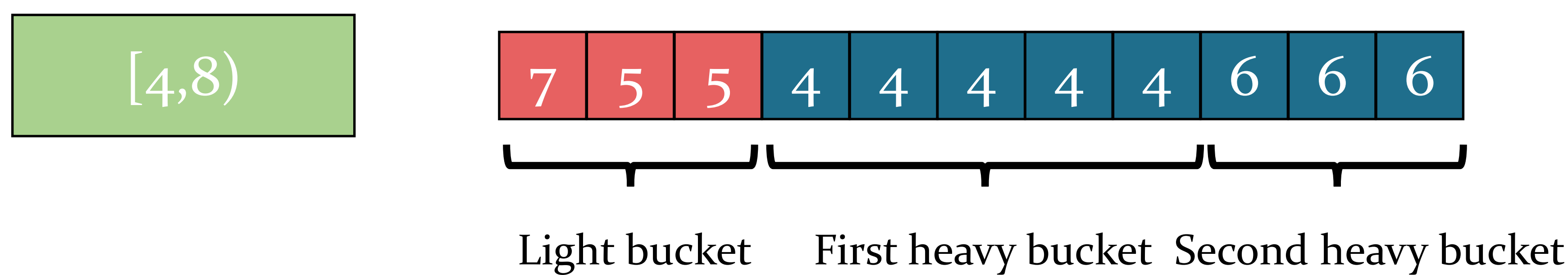
[0,4) [4,8) [8,12) [12,16)

DovetailSort: distributing

- Every key belongs to an MSD zone based on its digit
  - MSD zone 00: [0,4)
  - MSD zone 01: [4,8)
  - MSD zone 10: [8,12)
  - MSD zone 11: [12, 16)
- Keys across different MSD zones are already in order

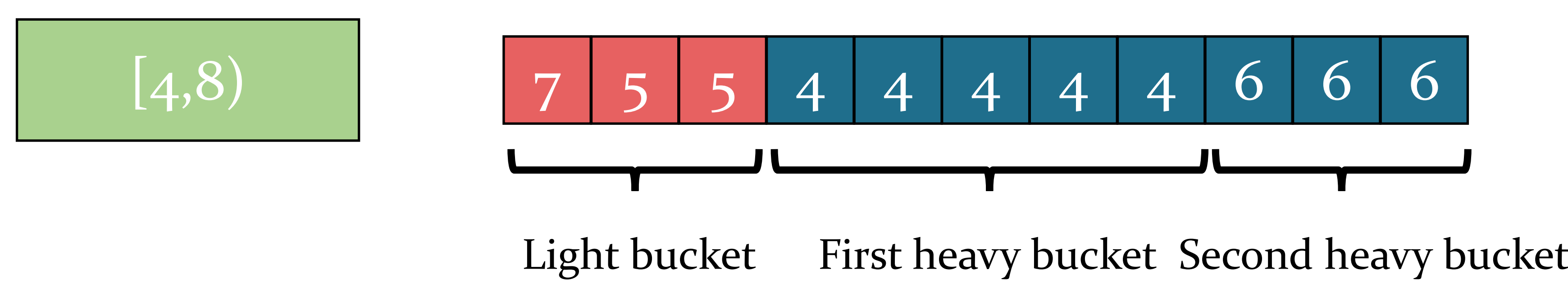
DovetailSort: distributing

- Each MSD zone consists of several buckets:
  - One light bucket: hold all light keys in this MSD zone
  - Multiple (or none) heavy buckets: hold each heavy key
- Keys are distributed into the buckets using counting sort



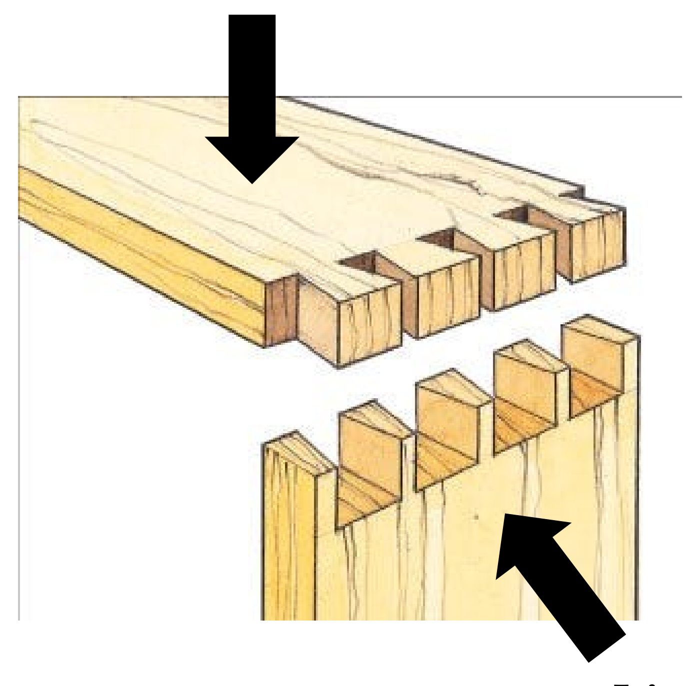
DovetailSort: recursing

- Sort the light buckets recursively



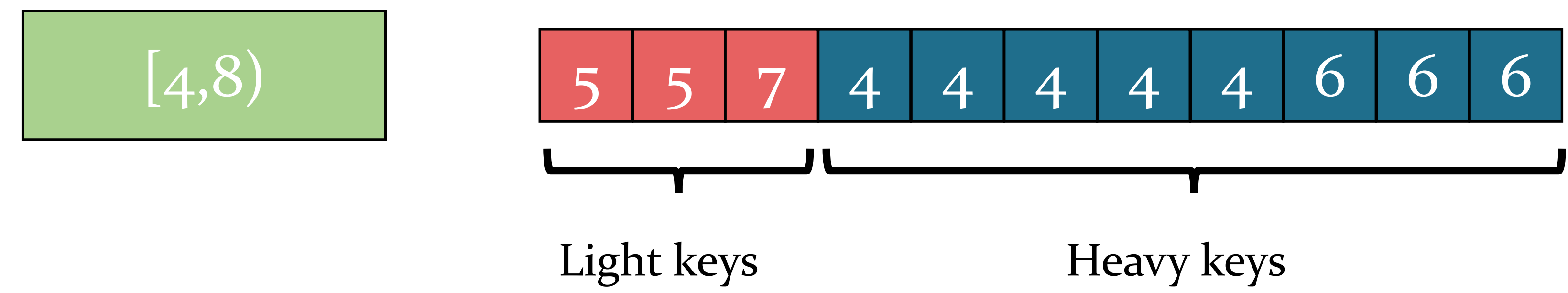
DovetailSort: recursing

- Sort the light buckets recursively
- Our goal: merge the two sorted lists



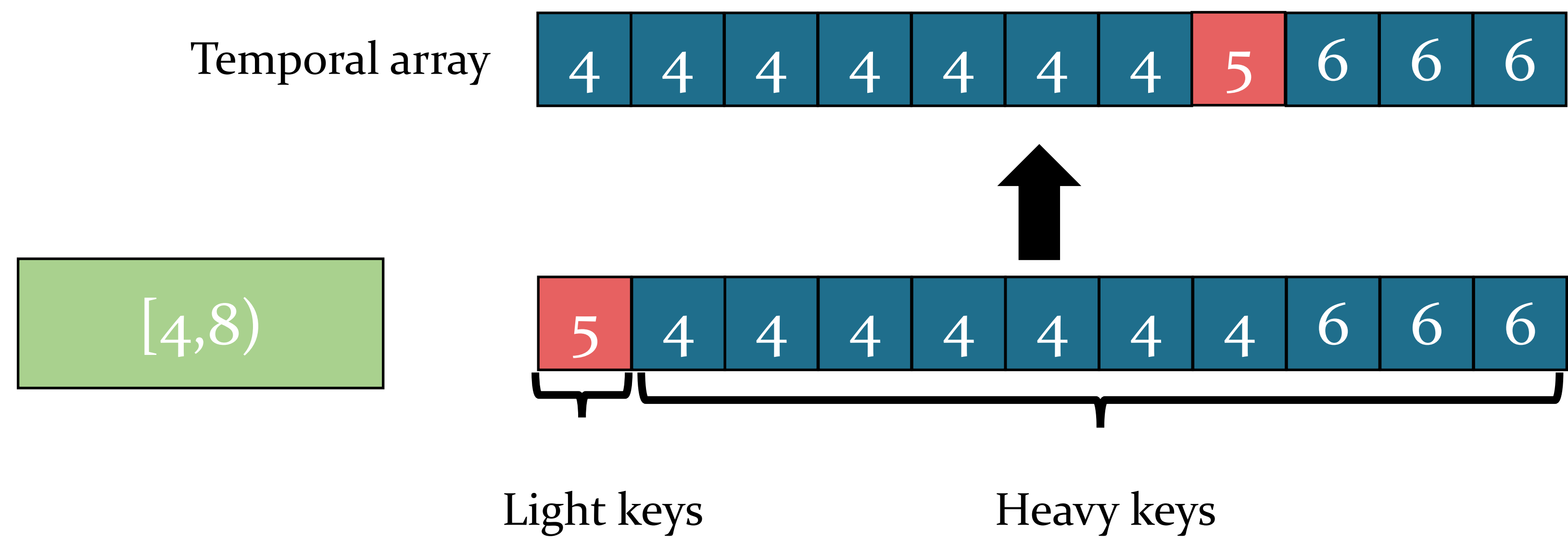
DovetailSort: dovetail merging

- Our goal: merge the two sorted lists
- Baseline solution: directly use a standard parallel merge algorithm



DovetailSort: dovetail merging

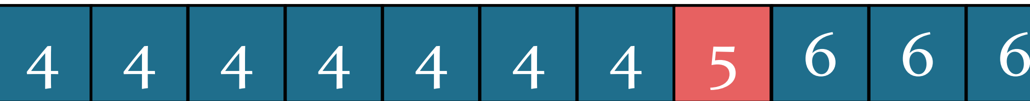
- However, practical parallel merge implementations are out-of-place
- We'll have to merge all keys into a temporary array, then copy it back



DovetailSort: dovetail merging

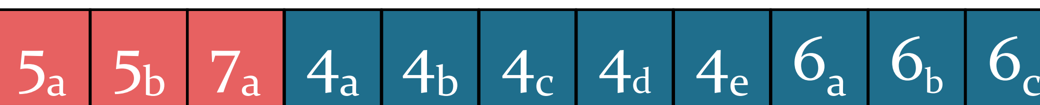
- However, practical parallel merge implementations are out-of-place
- We'll have to merge all keys into a temporary array, then copy it back

- Can we make this step more space-efficient?



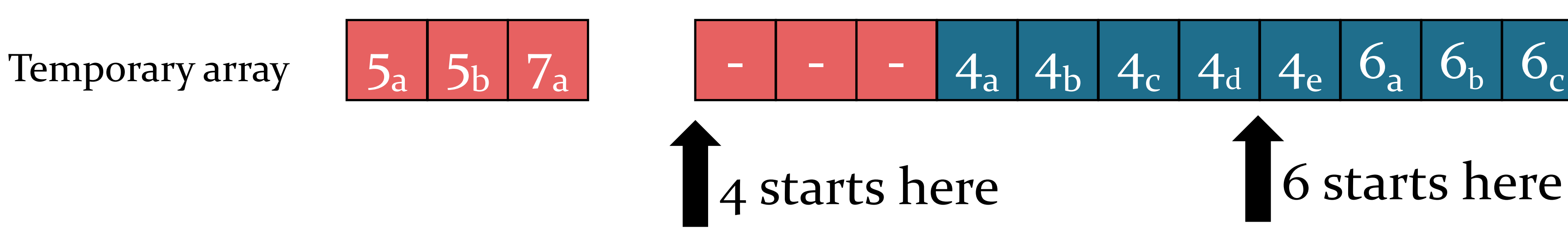
DovetailSort: dovetail merging

- Goal: merge the light keys and heavy keys into sorted order
  - Minimize data movements
  - Keep the sorted order stable
- Observation: the number of different heavy keys are small
- Our solution: to process the buckets in each MSD zone in sequential order



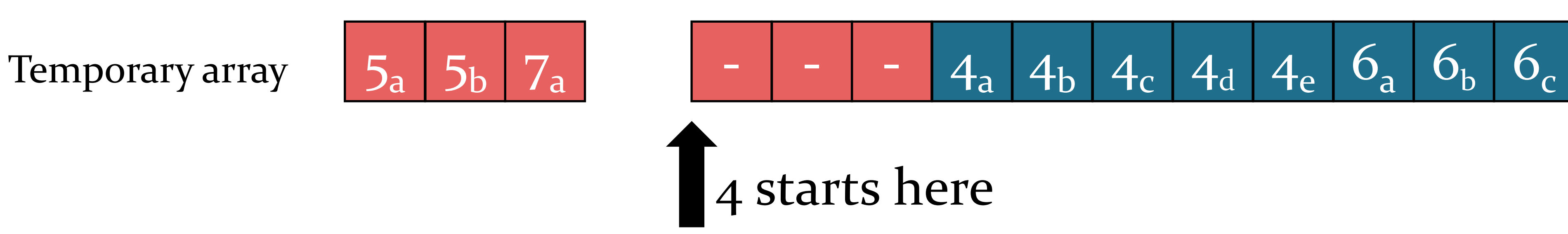
DovetailSort: dovetail merging

- Our solution: to process the buckets in each MSD zone in sequential order
- Step 1: copy the light (heavy) keys into a temporary array
- Step 2: for each heavy key, find the start position of each heavy key



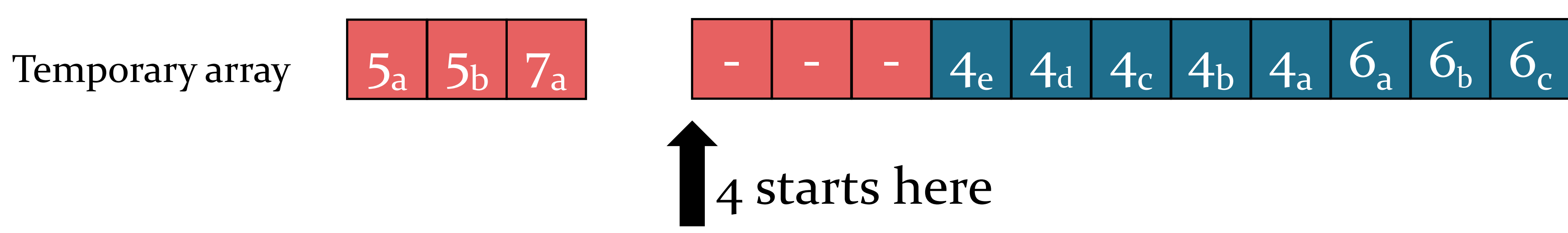
DovetailSort: dovetail merging

- Our solution: to process the buckets in each MSD zone in sequential order
- Step 1: copy the light (heavy) keys into a temporary array
- Step 2: for each heavy key, find the start position of each heavy key
    - Flip the bucket (in-place shifting [GM81])



DovetailSort: dovetail merging

- Our solution: to process the buckets in each MSD zone in sequential order
- Step 1: copy the light (heavy) keys into a temporary array
- Step 2: for each heavy key, find the start position of each heavy key
  - Flip the bucket (in-place shifting [GM81])
  - Flip the region from the start position to the end of the key



DovetailSort: dovetail merging

- Our solution: to process the buckets in each MSD zone in sequential order
- Step 1: copy the light (heavy) keys into a temporary array
- Step 2: for each heavy key, find the start position of each heavy key
  - Flip the bucket (in-place shifting [GM81])
  - Flip the region from the start position to the end of the key

Temporary array

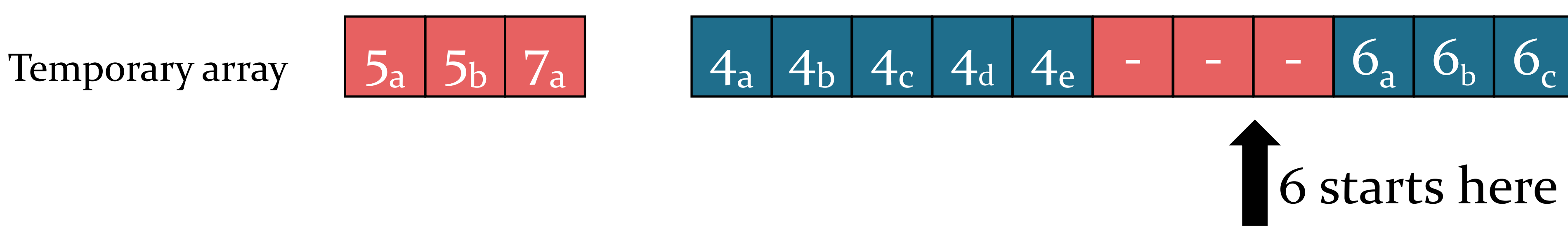
5_a	5_b	7_a

4_a	4_b	4_c	4_d	4_e	-	-	-	6_a	6_b	6_c

4 starts here

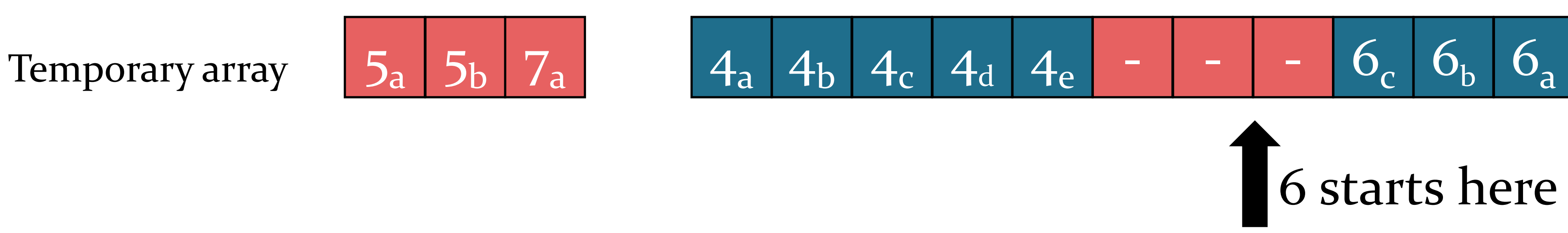
DovetailSort: dovetail merging

- Our solution: to process the buckets in each MSD zone in sequential order
- Step 1: copy the light (heavy) keys into a temporary array
- Step 2: for each heavy key, find the start position of each heavy key
  - Flip the bucket (in-place shifting [GM81])
  - Flip the region from the start position to the end of the key



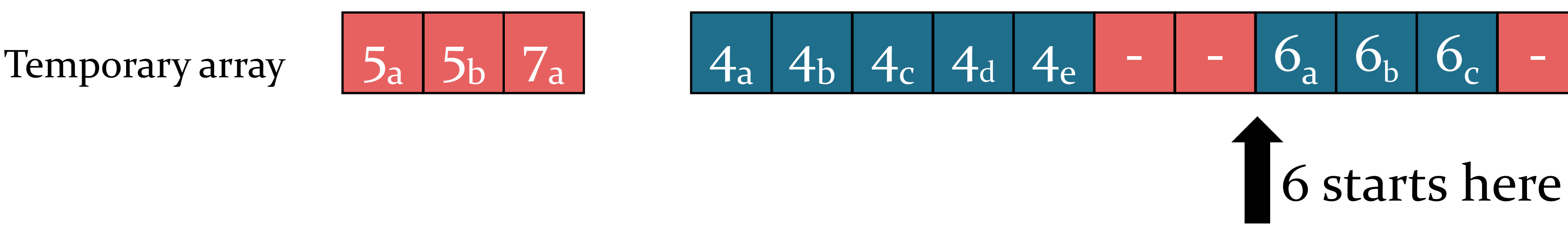
DovetailSort: dovetail merging

- Our solution: to process the buckets in each MSD zone in sequential order
- Step 1: copy the light (heavy) keys into a temporary array
- Step 2: for each heavy key, find the start position of each heavy key
  - Flip the bucket (in-place shifting [GM81])
  - Flip the region from the start position to the end of the key



DovetailSort: dovetail merging

- Our solution: to process the buckets in each MSD zone in sequential order
- Step 1: copy the light (heavy) keys into a temporary array
- Step 2: for each heavy key, find the start position of each heavy key
  - Flip the bucket (in-place shifting [GM81])
  - Flip the region from the start position to the end of the key

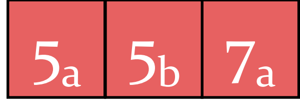


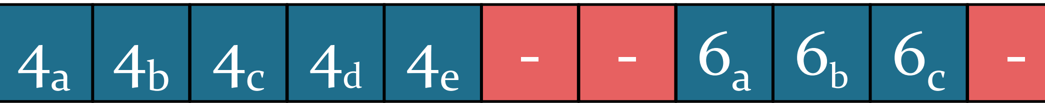
DovetailSort: dovetail merging

- Our solution: to process the buckets in each MSD zone in sequential order
- Step 1: copy the light (heavy) keys into a temporary array
- Step 2: for each heavy key, find the start position of each heavy key

- Step 3: copy the light keys back from the temporary array

Temporary array





DovetailSort: dovetail merging

- Our solution: to process the buckets in each MSD zone in sequential order
- Step 1: copy the light (heavy) keys into a temporary array
- Step 2: for each heavy key, find the start position of each heavy key
- Step 3: copy the light keys back from the temporary array

DovetailSort pseudocode

$DTSort(A[0..n-1], d)$

Base Cases	**If** $d = 0$ **then return** A
	if $\|A\| < \theta$ **then return** $ComparisonSort(A)$
Sampling	Detect heavy keys by sampling
Distributing	Distribute each integer to the corresponding heavy/light buckets
Recursing	**Parallel_for_each bucket** $\boldsymbol{B}$ $DTSort(B, d-1)$
Merging	Merge the heavy keys with the light keys

Theoretical analysis

Theoretical analysis

- Despite many theoretical studies [AH97,AHNR98], for practical implementations, the best-known work bound is $O(n \log r)$ [OKFS19]
  - No better than comparison sort when $r = \Omega(n)$

- We prove that existing parallel MSD sort implementations [OKFS19, BAD20, AWFS22], including ours, have $\mathrm{O}(n\sqrt{\log r})$ work

Analysis on existing parallel integer sort implementations

- Input parameters:
  - r: the maximum key
- Algorithm parameters:
  - θ: base case size to comparison sort
  - γ: number of bits to sort in one recursion

PMSDSort$(A[0..n-1], d)$

If $d = 0$ **then**

    return A

if $|A| < \theta$ **then**

    return *ComparisonSort*(A)

CountSort(A)

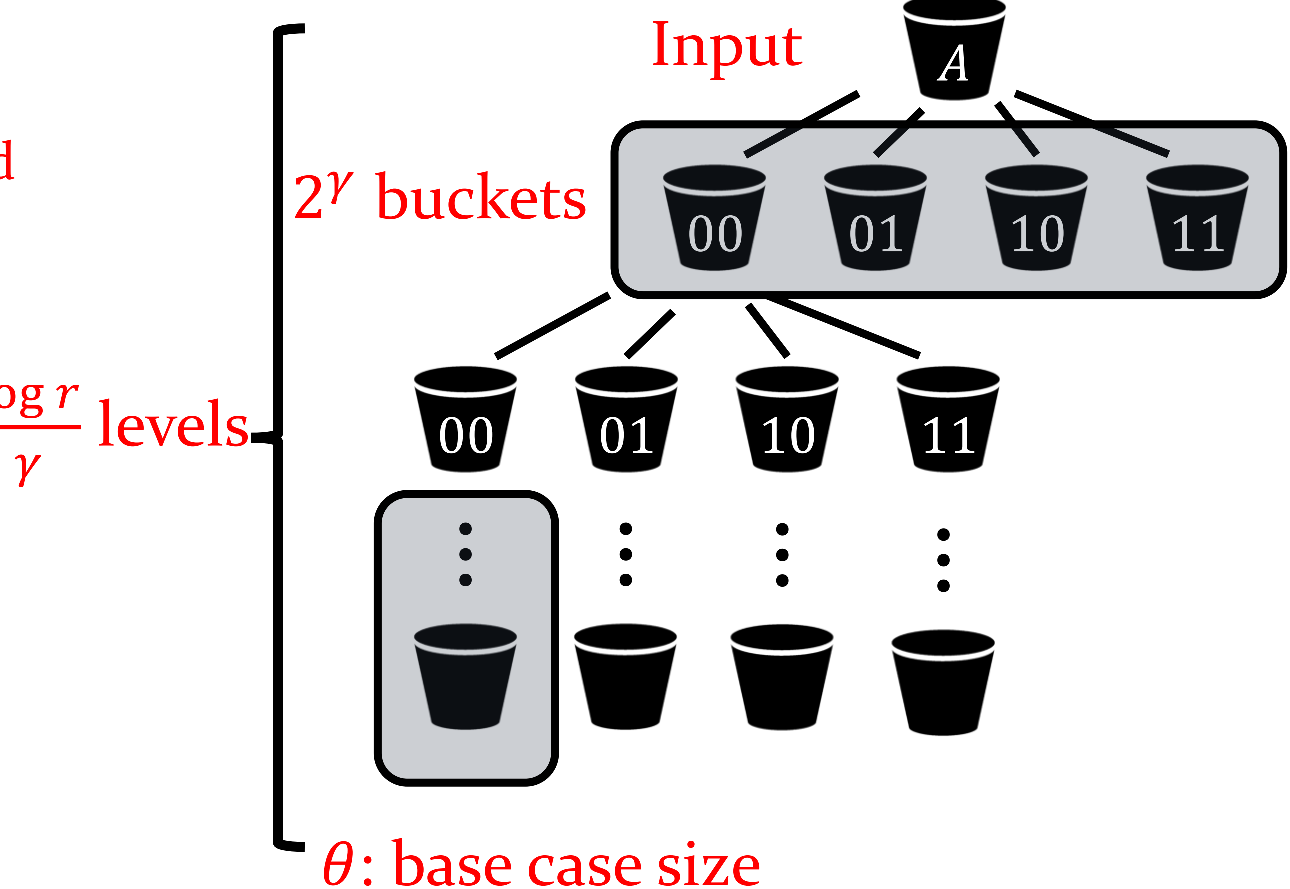
Parallel_for $i \leftarrow 0$ to 2^{γ} **do**

    PMSDSort$(A_i, d-1)$

Recursion tree of MSD sort

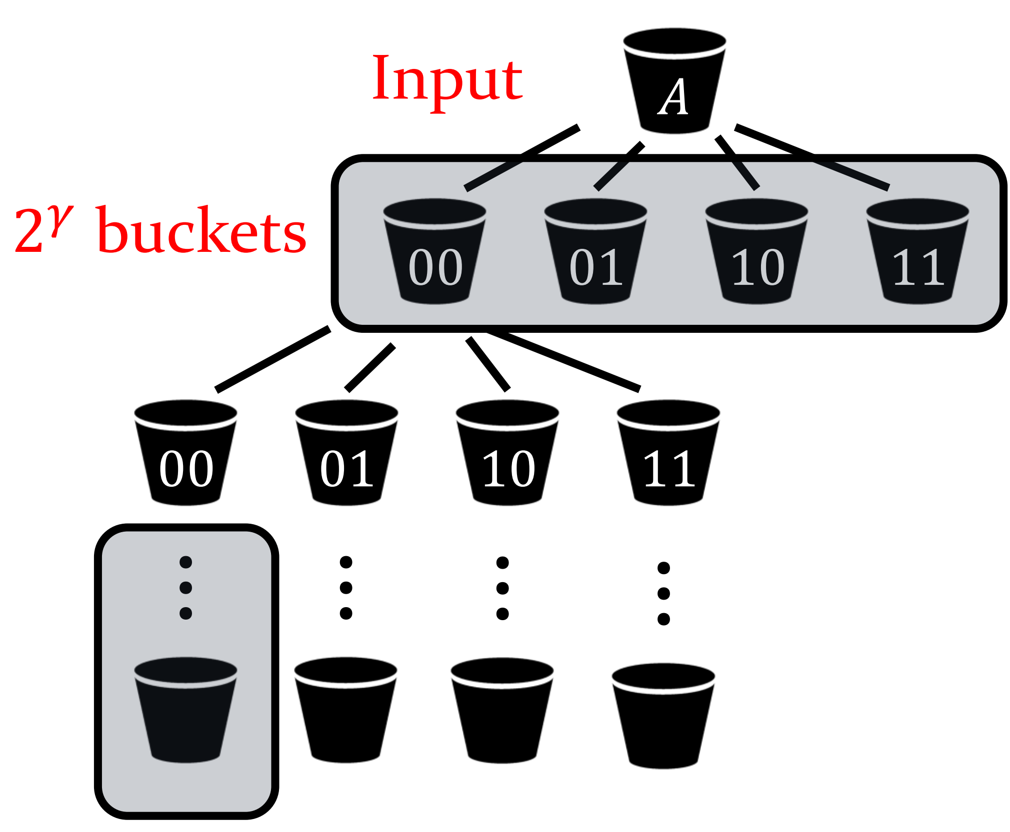
- We need to sort $\log r$ bits in total, and γ bits in each recursion
- Costs from internal nodes
  - Count sort has $O(n_i + 2^\gamma)$ work
  - If $n_i \le 2^\gamma$, go to the base case
  - $\theta = 2^\gamma$
  - Cost per level: $\sum O(n_i) = O(n)$
  - Total costs: $\frac{\log r}{\gamma} \cdot O(n)$

Should be bounded



Recursion tree of MSD sort

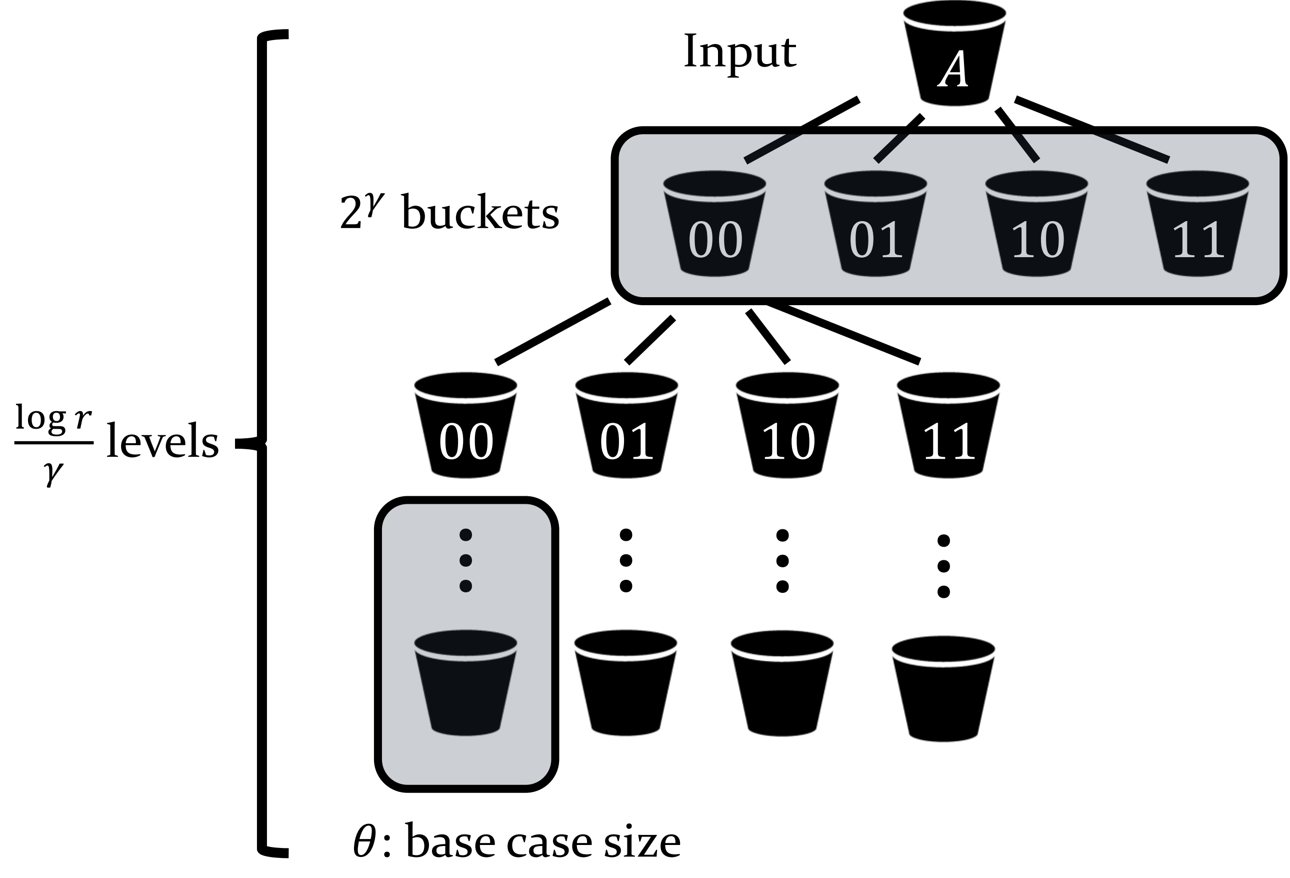
- $\theta = 2^{\gamma}$
- Costs from leaf nodes
  - $\sum O(n_i \log n_i)$ for $n = \sum n_i$
  - In the base case: $n_i \leq \theta = 2^{\gamma}$
  - $\sum O(n_i \log n_i) \leq \sum O(n_i \gamma) = O(n\gamma)$



Recursion tree of MSD sort

- Total cost: $O\left(\frac{\log r}{\gamma} \cdot n + n\gamma\right)$
- $\gamma = O(\sqrt{\log r})$
- Work bound: $O(n\sqrt{\log r})$

- In practice:
  - r is usually 2^{64}
  - $\gamma = \sqrt{\log r} = \sqrt{64} = 8$
  - $\theta = 2^{\gamma} = 2^{8}$



More theoretical analyses in our paper

- Span analysis on general parallel MSD sort algorithms
- The same work and span bound for DovetailSort
- Dovetail Sort can achieve linear work for exponential distributions and heavily duplicated uniform distributions

Experiments

Setup

- A 96-core with four Intel Xeon Gold 6252 CPUs (192 hyper-threads)
  - 1.5TB main memory and 36MB*4 L3 cache
  - C++ codes compiled with clang 14.0.6 using ParlayLib with -O3 flag
- 4 distributions (uniform, exponential, Zipfian, BExp) are tested
  - Each with 5 different parameters
  - BExp is an adversarial distribution we designed for integer sort

Competitors

- Sample sort
  - IPS^{4}o: IPS^{4}o sample sort (Axtmann et al., ESA ’17)
  - PLSS: ParlayLib sample sort (Blelloch et al., SPAA ’20)
- Integer sort
  - PLIS: ParlayLib integer sort (Blelloch et al., SPAA ’20)
  - IPS^{2}Ra: IPS^{2}Ra integer sort (Axtmann et al., TOPC ’22)
  - RS: Region sort (Obeya et al., SPAA ’19)

Performance Heatmap

- Three distributions (uniform, exponential, Zipfian), each with five parameters
  - Heaviness is ordered from high to low within each distribution.
- Each row is a test instance, each column is an algorithm
- The numbers are the slowdown relative to the fastest implementation on each instance
- Smaller/Green is better.

Fewer duplicates ↓ More duplicates

		Integer				Comparison	
		Ours	PLIS	IPS²Ra	RS	PLSS	IPS⁴o
Uniform	10^9	1.00	1.07	1.34	1.44	2.55	1.38
	10^7	1.00	1.09	1.20	1.41	2.27	1.21
	10^5	1.00	1.13	1.24	1.46	2.27	1.36
	10^3	1.17	1.17	1.24	1.42	1.86	1.00
	10	1.00	2.29	3.68	1.42	3.11	1.48
Exponential	1	1.00	1.02	1.09	1.35	2.07	1.28
	2	1.00	1.09	1.15	1.42	2.16	1.32
	5	1.00	1.31	1.34	1.62	2.55	1.41
	7	1.00	1.39	1.32	1.69	2.66	1.45
	10	1.00	1.50	1.39	1.70	2.77	1.40
Zipfian	0.6	1.00	1.10	1.28	1.46	3.00	1.40
	0.8	1.00	1.03	1.18	1.35	2.09	1.28
	1	1.02	1.07	1.10	1.25	1.83	1.00
	1.2	1.00	1.61	2.08	1.37	2.33	1.44
	1.5	1.00	2.12	4.27	1.56	2.75	2.10
	Avg.	1.01	1.29	1.49	1.46	2.39	1.35

$n = 10^9$, on a 96-core machine

Performance Heatmap

- Our algorithm performs the best on 13/15 test instances.
  - Close on the remaining two
- On geometric mean average, our algorithm achieves a 28% speedup over the next best implementation

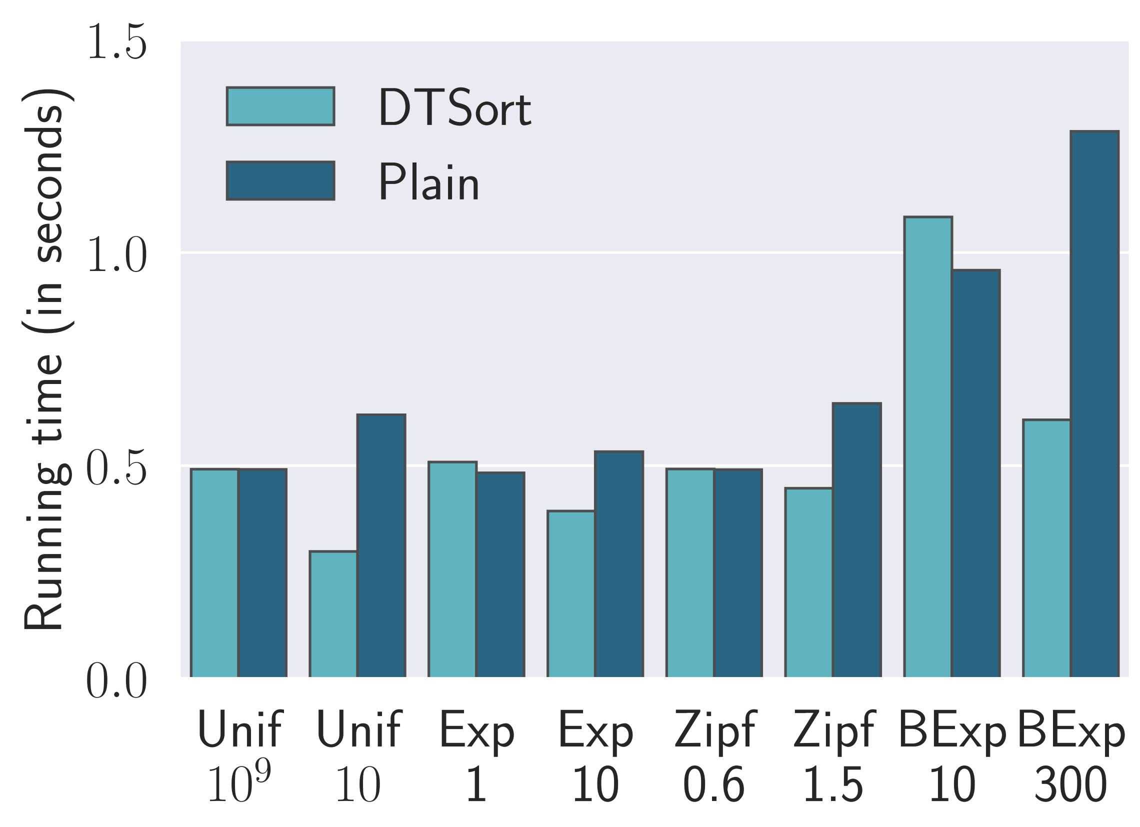
Fewer duplicates → More duplicates

		Integer				Comparison	
		Ours	PLIS	IPS²Ra	RS	PLSS	IPS⁴o
Uniform	10^9	1.00	1.07	1.34	1.44	2.55	1.38
	10^7	1.00	1.09	1.20	1.41	2.27	1.21
	10^5	1.00	1.13	1.24	1.46	2.27	1.36
	10^3	1.17	1.17	1.24	1.42	1.86	1.00
	10	1.00	2.29	3.68	1.42	3.11	1.48
Exponential	1	1.00	1.02	1.09	1.35	2.07	1.28
	2	1.00	1.09	1.15	1.42	2.16	1.32
	5	1.00	1.31	1.34	1.62	2.55	1.41
	7	1.00	1.39	1.32	1.69	2.66	1.45
	10	1.00	1.50	1.39	1.70	2.77	1.40
Zipfian	0.6	1.00	1.10	1.28	1.46	3.00	1.40
	0.8	1.00	1.03	1.18	1.35	2.09	1.28
	1	1.02	1.07	1.10	1.25	1.83	1.00
	1.2	1.00	1.61	2.08	1.37	2.33	1.44
	1.5	1.00	2.12	4.27	1.56	2.75	2.10
Avg.		1.01	1.29	1.49	1.46	2.39	1.35

$n = 10^9$, on a 96-core machine

Performance improvement by heavy key detections

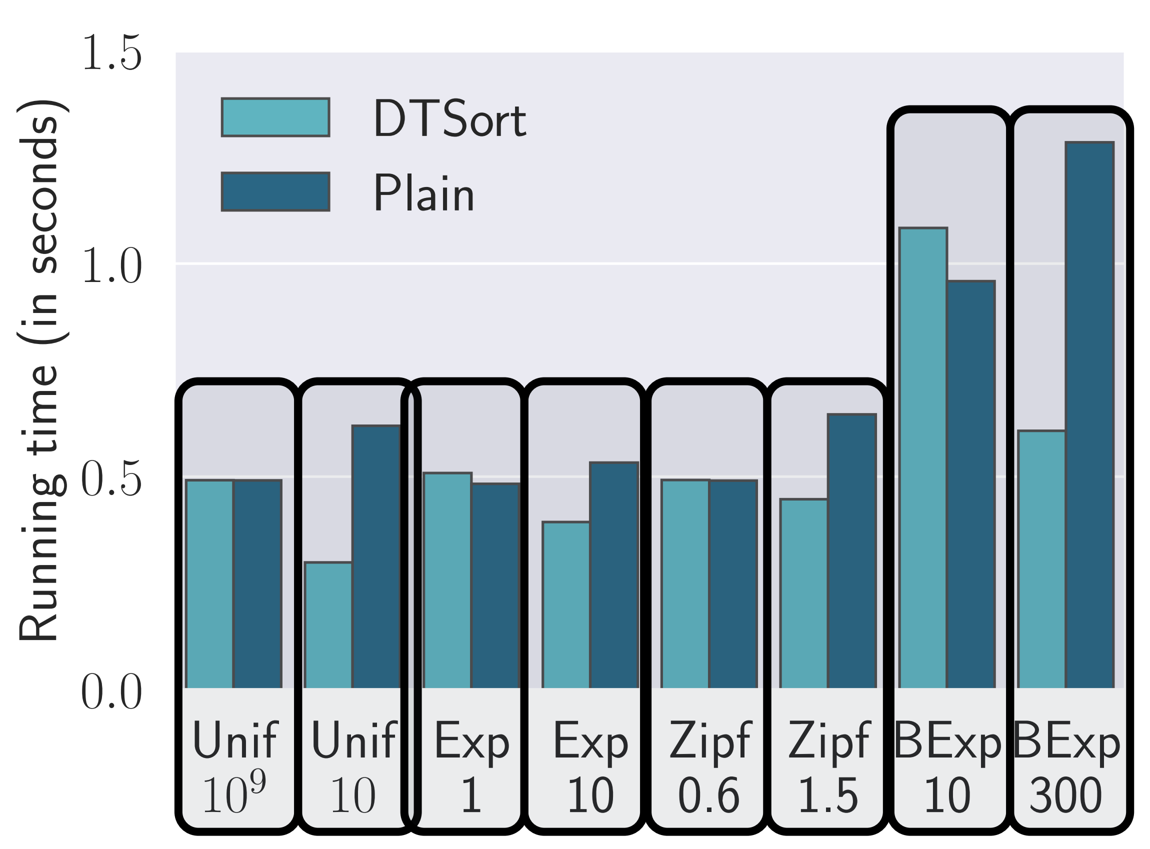
- Each bar is the actual running time
- Lower is better
- DTSort: our implementation with heavy key detections
- Plain: our implementation without heavy key detections



$n = 10^9$, on a 96-core machine

Performance improvement by heavy key detections

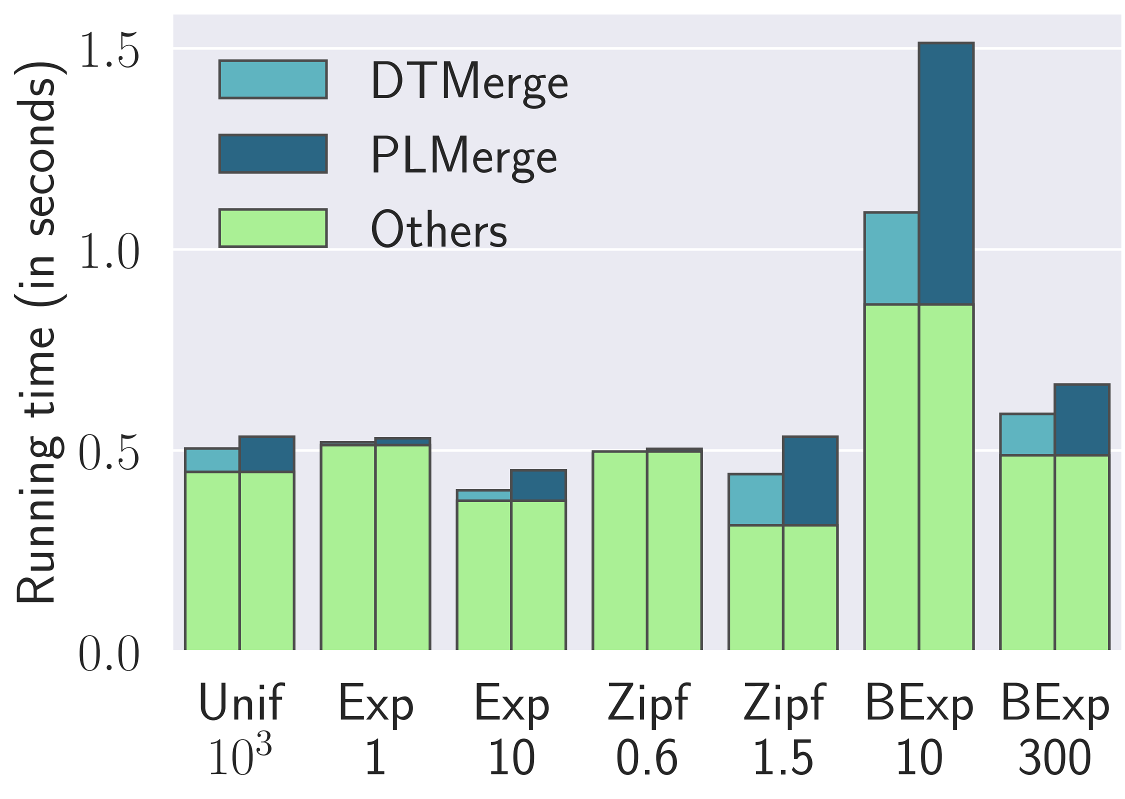
- On uniform distributions without duplicates, DTSort and plain has similar performance
  - The overhead of doing sampling and merging is very small
- On distributions with duplicates, the performance speedup is visible
  - Up to 2x speedup compared to Plain



$n = 10^9$, on a 96-core machine

DTMerge vs standard merge

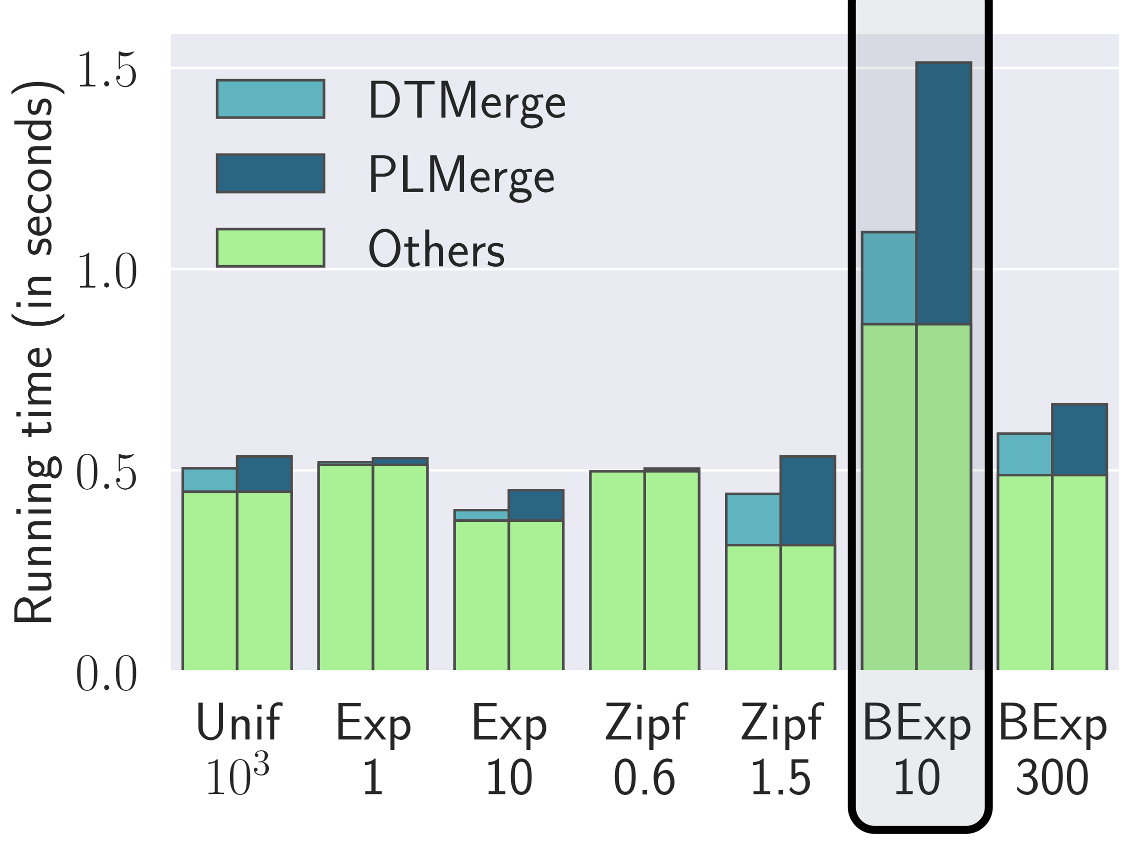
- Each bar is the actual running time
- Lower is better
- Blue: merge running time
- Green: other running time
- DTMerge: our merge algorithm
- PLMerge: a standard merge algorithm using the primitive from ParlayLib



$n = 10^9$, on a 96-core machine

DTMerge vs standard merge

- Our DTMerge always achieves better performance
- On distributions that interleaves light and heavy keys, our algorithm can improve the total running time by up to 39%



$n = 10^9$, on a 96-core machine

More in-depth experimental analysis

- Two applications
  - Graph transposing
  - Morton sort (z-order)

- Scalability with different number of threads

- Scalability with different input sizes

Summary

Our contributions

- Theoretical challenge: Can we theoretically explain, why integer sort is practically faster than comparison sort?
- We proved that a class of practical parallel integer sort implementations, including our new one, have $O(n\sqrt{\log r})$ work

- Practically challenge: Can integer sort consistently outperform comparison sort (particularly with heavy duplicates)?
- We proposed DovetailSort that combines the advantages of integer and sample sort and is consistently faster than all existing algorithms in most tested cases

Summary

- Theoretical contribution: we proved that a class of practical parallel integer sort implementations, including our new one, have $O(n\sqrt{\log r})$ work
- Practical contribution: DovetailSort is consistently faster than existing sorting implementations by detecting and processing heavy keys separately

- Full version: https://arxiv.org/abs/2401.00710
- Code: https://github.com/ucrparlay/DovetailSort
- Contact: Xiaojun Dong from UC Riverside (xdong038@ucr.edu)