Parallel Integer Sort: Theory and Practice

A Foray into Dovetail Sort

Authors

PPoPP '24: Proceedings of the 29th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming

Parallel Integer Sort: Theory and Practice

Xiaojun Dong, UC Riverside, xdong038@ucr.edu
Laxman Dhulipala, University of Maryland, laxman@umd.edu
Yan Gu, UC Riverside, ygu@cs.ucr.edu
Yihan Sun, UC Riverside, yihans@cs.ucr.edu

Abstract

Integer sorting is a fundamental problem in computer science. This paper studies parallel integer sort both in theory and in practice. In theory, we show tighter bounds for a class of existing practical integer sort algorithms, which provides a solid theoretical foundation for their widespread usage in practice and strong performance. In practice, we design a new integer sorting algorithm, DovetailSort, that is theoretically-efficient and has good practical performance.

In particular, DovetailSort overcomes a common challenge in existing parallel integer sorting algorithms, which is the difficulty of detecting and taking advantage of duplicate keys. The key insight in DovetailSort is to combine algorithmic ideas from both integer- and comparison-sorting algorithms. In our experiments, DovetailSort achieves competitive or better performance than existing state-of-the-art parallel integer and comparison sorting algorithms on various synthetic and real-world datasets.

CCS Concepts: • **Theory of computation → Parallel algorithms; Shared memory algorithms; Sorting and searching.**

Keywords: Integer Sort, Radix Sort, Parallel Algorithms

1 Introduction

Sorting is one of the most widely-used primitives in algorithm design, and has been extensively studied. For many if not most cases, the keys to be sorted are fixed-length integers. Sorting integer keys is referred to as the ***integer sort*** problem. An integer sorting algorithm takes as input n records with integer keys in the range 0 to $r-1$, and outputs the records with keys in non-decreasing order. Despite decades of effort in studying integer sorting algorithms, however, obtaining *parallel integer sorting (IS) algorithms that are efficient both in theory and in practice* has remained elusive.

Theoretical Challenges. *As a special type of sorting, integer sorting algorithms* ***can*** *outperform comparison sorting algorithms by using the integer encoding of keys.* This claim is verified in many existing studies [5, 43] as well as our experiments (see Fig. 1). As a result, in real-world applications (e.g., [19, 51, 54]), integer sort is usually preferred (instead of comparison sort) when the keys are integers. While it is not surprising that IS algorithms can outperform comparison sorts, we observe a *significant gap in connecting the high performance with theory*. Theoretical parallel IS algorithms with good bounds [2, 3, 7, 31, 41] are quite complicated, and we are unaware of any implementations of them. On the other hand, for the practical parallel IS solutions [5, 9, 43], we are unaware of "meaningful" analysis to explain their good performance for general r. The best-known bounds for them, as discussed in [43], are $O(n \log r)$ work (number of operations) and polylog(rn) span (longest dependence chain). However, note that the main use case for integer sort is when $r = \Omega(n)$, since otherwise the simpler counting sort [45, 53] can be used. In this case, the bounds for practical parallel IS algorithms are no better than comparison sorts ($O(n \log n)$ work and polylogarithmic span [9, 12, 13, 24]). This leads to the following open question in theory: do the practical parallel IS algorithms indeed have lower asymptotic costs than comparison sort (and if so, under what circumstances)?

Practical Challenges. *As a special type of sorting, integer sorting algorithms* ***should*** *outperform comparison sorting algorithms by using the integer encoding of keys.* Since integers are comparable, comparison sort can be considered as a baseline for sorting integers. Unfortunately, SOTA parallel IS algorithms do not consistently outperform comparison-based sorting algorithms. One key reason is the inherent difficulty of dealing with *duplicate keys* in integer sorts. In principle, duplicate keys are beneficial for sorting algorithms. For example, *samplesort* can skip a recursive subproblem between two equal pivots; similarly, quicksort can separate keys equal to the pivot to avoid further processing them. Interestingly, such a case does not apply to integer sort. Existing parallel IS implementations follow the *most-significant digit (MSD)* framework that partitions all keys into buckets based on the *integer encoding* (i.e., 8–12 highest bits), and recurses within each bucket. As such, equal keys cannot be detected until the last recursion. Although some techniques can be used to detect special distributions (e.g., all keys are the same), to the best of our knowledge, no existing parallel IS implementation can benefit from duplicate keys in a general (provable) and non-trivial manner[1]. Fig. 1 and Tab. 3

[1] Some techniques in existing IS implementations [5, 43] have a side-effect to benefit from duplicates in some cases, at a cost of making the algorithms *unstable*. We want to overcome this issue without sacrificing stability.

PPoPP '24, March 2–6, 2024, Edinburgh, United Kingdom

ACM ISBN 979-8-4007-0435-2/24/03.
https://doi.org/10.1145/3627535.3638483

Xiaojun Dong
UC Riverside

- Final year PhD
- Bachelor at Huazhong University of Science and Technology)

Laxman Dhulipala
University of Maryland

- Assistant professor at U Maryland
- **Advised by Guy Blelloch, postdoc with Julian**

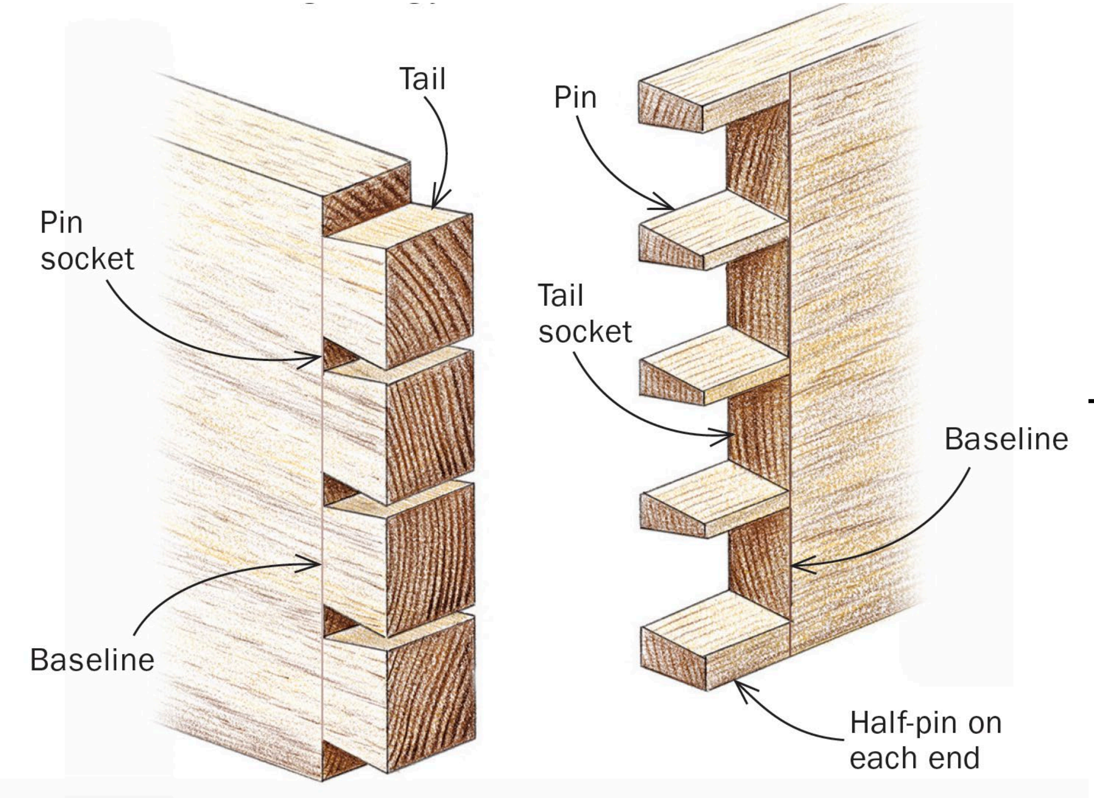
Yan Gu
UC Riverside

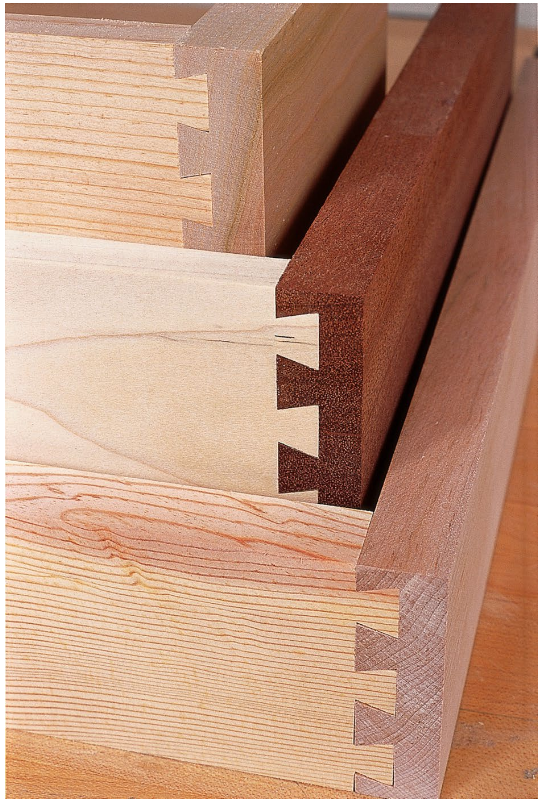
- Assistant professor at UC Riverside
- **Advised by Guy Blelloch, postdoc with Julian**
- Bachelor at Tsinghua

Yihan Sun
UC Riverside

- Associate professor at UC Riverside
- Bachelor at Tsinghua

Dovetail Sort: What is a dovetail anyway?

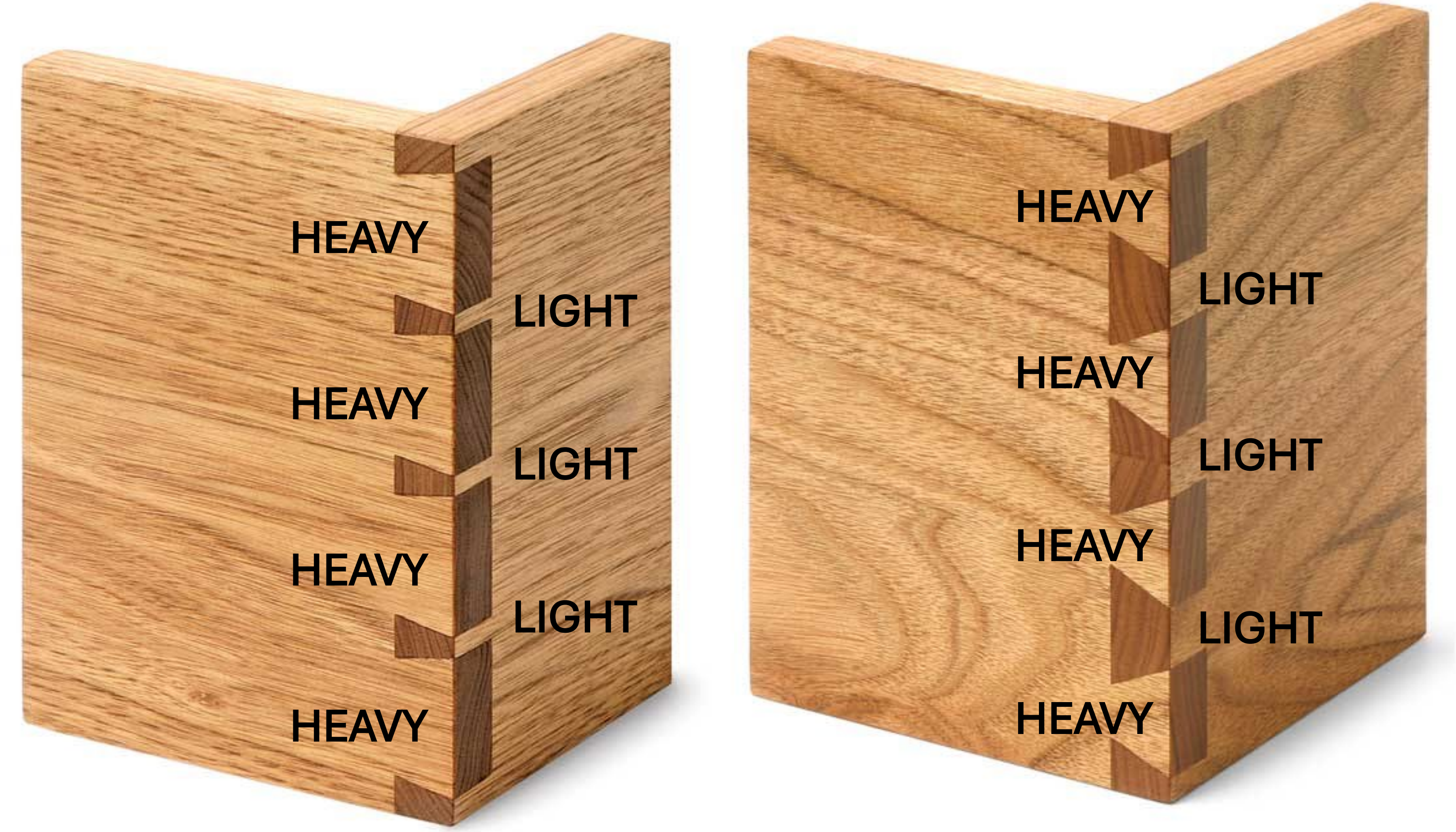




finewoodworking.com, popularwoodworking.com

Dovetail Sort: What is a dovetail anyway?

Dovetail Merge (DTMerge)



finewoodworking.com

Integer sorting

e.g. Counting sort. Faster than comparison sorts because integers can also be used as **indices**

7	8	4	3	3	5	0	7	7	9	1

Unsorted array. Range = 0-9

Integer sorting

e.g. Counting sort. Faster than comparison sorts because integers can also be used as **indices**

7	8	4	3	3	5	0	7	7	9	1

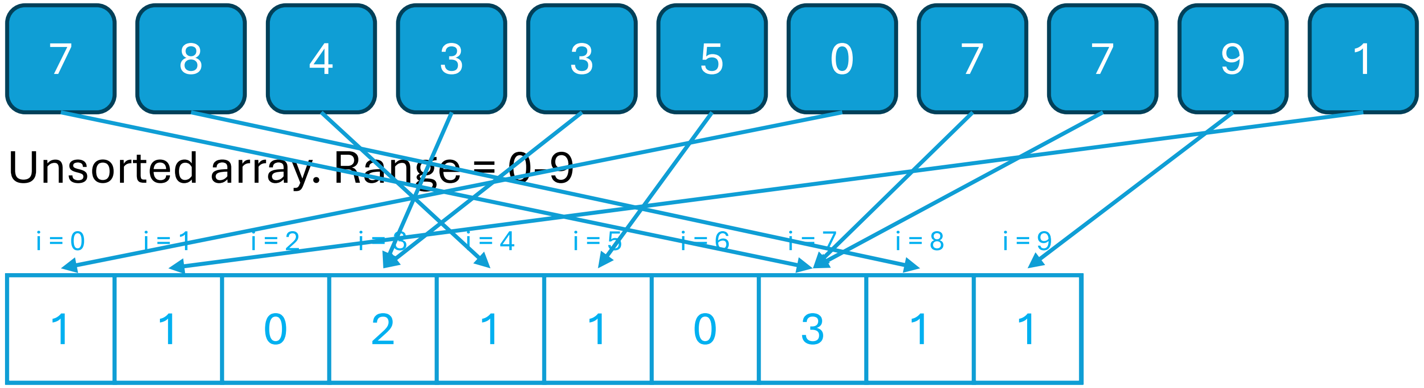
Unsorted array. Range = 0-9

i = 0	i = 1	i = 2	i = 3	i = 4	i = 5	i = 6	i = 7	i = 8	i = 9
0	0	0	0	0	0	0	0	0	0

Initialize counting array.

Integer sorting

e.g. Counting sort. Faster than comparison sorts because integers can also be used as **indices**



Unsorted array. Range = 0-9

Initialize counting array.

Integer sorting

e.g. Counting sort. Faster than comparison sorts because integers can also be used as **indices**

7	8	4	3	3	5	0	7	7	9	1

Unsorted array. Range = 0-9

i = 0	i = 1	i = 2	i = 3	i = 4	i = 5	i = 6	i = 7	i = 8	i = 9
1	1	0	2	1	1	0	3	1	1

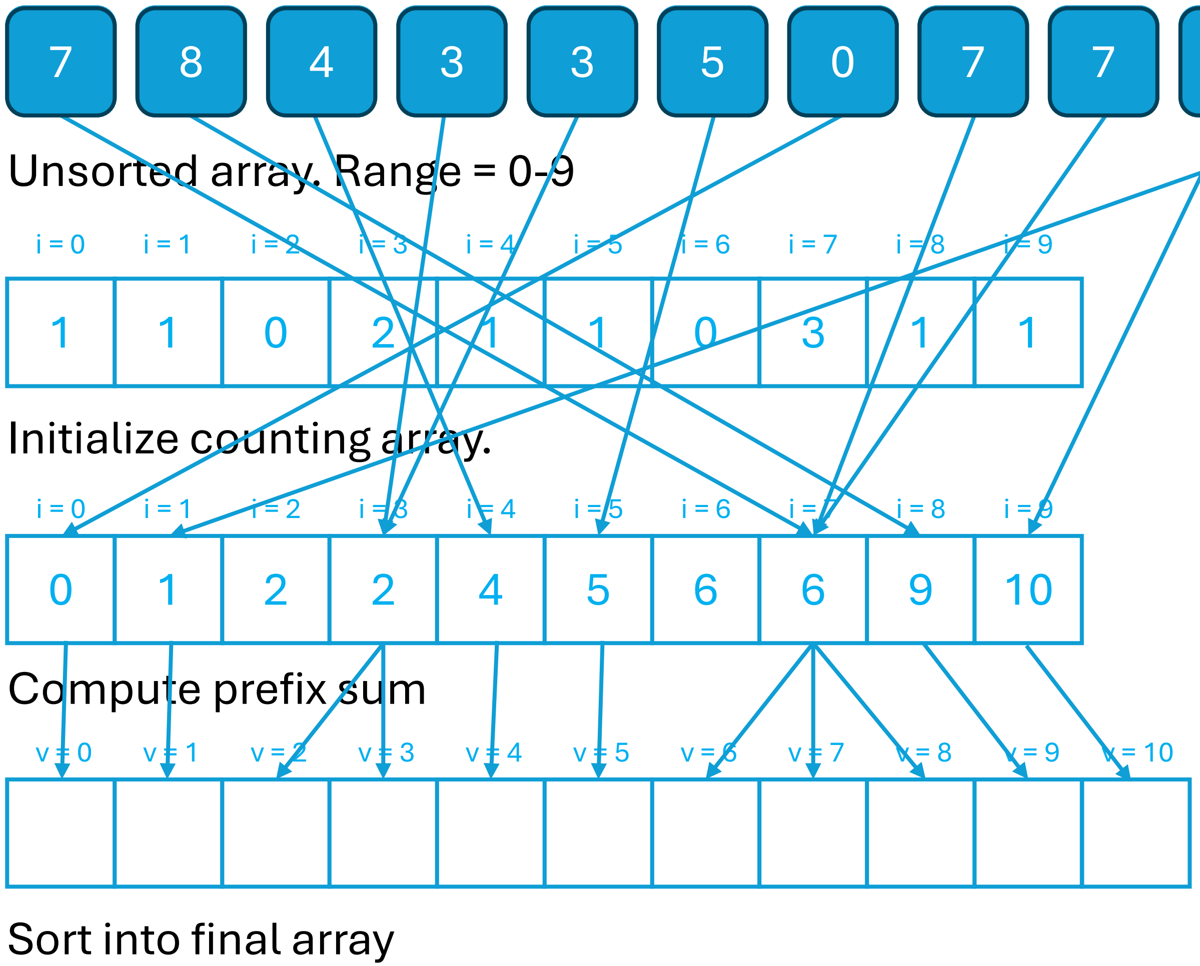
Initialize counting array.

i = 0	i = 1	i = 2	i = 3	i = 4	i = 5	i = 6	i = 7	i = 8	i = 9
0	1	2	2	4	5	6	6	9	10

Compute prefix sum

Integer sorting

e.g. Counting sort. Faster than comparison sorts because integers can also be used as **indices**



No comparisons needed! Just counting

Slow for integers where r >= Ω(n) (around 16 bits)

Radix sort

Chunking a big integer sort problem into smaller problems that can be handled quickly with counting sort.

Unsorted array

7921
0245
9224
0023
2331
2301
4531

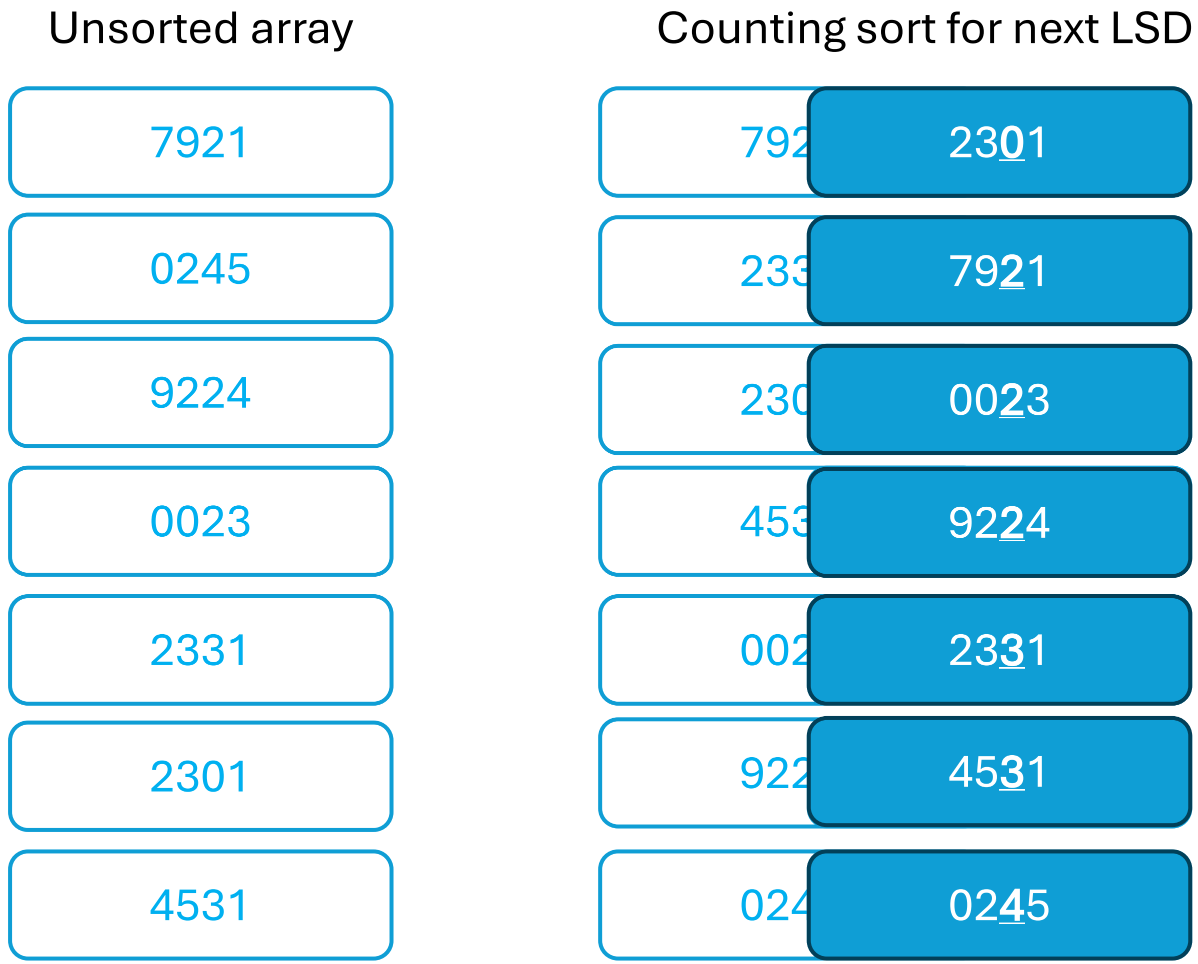
Radix sort

Chunking a big integer sort problem into smaller problems that can be handled quickly with counting sort.

Unsorted array	Counting sort for LSD
7921	7921
0245	2331
9224	2301
0023	4531
2331	0023
2301	9224
4531	0245

Radix sort

Chunking a big integer sort problem into smaller problems that can be handled quickly with counting sort.



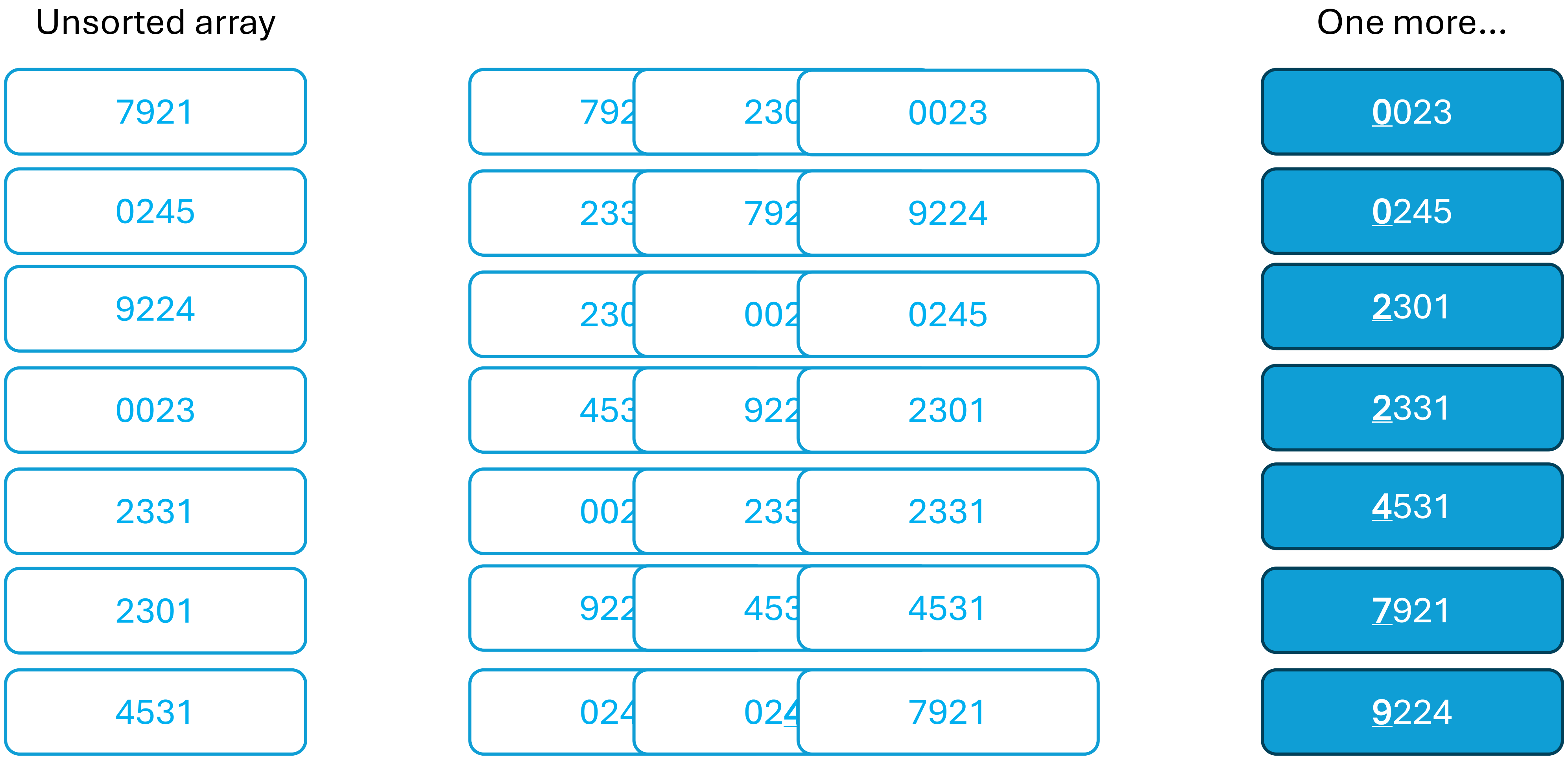
Radix sort

Chunking a big integer sort problem into smaller problems that can be handled quickly with counting sort.

Unsorted array			And the next LSD
7921	792	230	0023
0245	233	792	9224
9224	230	002	0245
0023	453	922	2301
2331	002	233	2331
2301	922	453	4531
4531	024	024	7921

Radix sort

Chunking a big integer sort problem into smaller problems that can be handled quickly with counting sort.



Radix sort

Chunking a big integer sort problem into smaller problems that can be handled quickly with counting sort.

Unsorted array				And done!
7921	792	230	0023	23
0245	233	792	9224	245
9224	230	002	0245	2301
0023	453	922	2301	2331
2331	002	233	2331	4531
2301	922	435	4531	7921
4531	024	024	7921	9224

Radix sort

Chunking a big integer sort problem into smaller problems that can be handled quickly with counting sort.

Unsorted array

01111010110001
00000011110101
10010001100000
00000000010111
00100100011011
00010010001101
10001101101011

Radix sort works in binary too.

γ = number of bits in a digit (e.g. 2)

b = 2^{γ} "radix" size (e.g. 4)

r = range (e.g. 16 for a nibble)

And done!

00000000010111
00000011110101
00010010001101
00100100011011
10001101101011
01111010110001
10010001100000

Radix sort

Chunking a big integer sort problem into smaller problems that can be handled quickly with counting sort.

Unsorted array		And done!
01111010110001		00000000010111
00000011110101		00000011110101
10010001100000		00010010001101
00000000010111		00100100011011
00100100011011		10001101101011
00010010001101		01111010110001
10001101101011		10010001100000

Redundant!!! duplicate values are always sorted individually.
In part because parallel IS algorithms use **MSD** instead of **LSD** so identical values are only identified at the end
(vs samplesort being able to identify buckets between identical low and high pivots and skip them, for instance)

$\gamma = 2$
$b = 2^{\gamma} = 4$
$r = 16$

Dovetail sort (DTSort)

Bucketing high frequency (heavy) values, recursing over low frequency (light) values, merging.

Random sampling

Input $\boxed{6}$ 4 0 $\boxed{4}$ 8 $\boxed{2}$ 6 4 $\boxed{7}$ $\boxed{9}$ 11 5 15 $\boxed{4}$ 13 10 9 $\boxed{4}$ 14 5 $\boxed{9}$ 11 $\boxed{6}$ 9

$\gamma = 2$
$b = 2^{\gamma} = 4$
$r = 16$

Dovetail sort (DTSort)

Bucketing high frequency (heavy) values, recursing over low frequency (light) values, merging.

Random sampling

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

Which values appear >1?

Based on Chernoff bounds, any key appearing more than once must have $\Omega(n/p)$ occurrences in the input! "Heavy" keys.

$\gamma = 2$
$b = 2^\gamma = 4$
$r = 16$

Dovetail sort (DTSort)

Bucketing high frequency (heavy) values, recursing over low frequency (light) values, merging.

Random sampling

Input $\boxed{6}$ 4 0 $\boxed{4}$ 8 2 6 4 7 $\boxed{9}$ 11 5 15 $\boxed{4}$ 13 10 9 $\boxed{4}$ 14 5 $\boxed{9}$ 11 $\boxed{6}$ 9

Binary representation

$\boxed{0110}$ 0100 0000 $\boxed{0100}$ 1000 0010 0110 0100 0111 $\boxed{1001}$ 1011 0101 1111 $\boxed{0100}$ 1101 1010 1001 $\boxed{0100}$ 1110 0101 $\boxed{1001}$ 1011 $\boxed{0110}$ 1001

$\gamma = 2$
$b = 2^{\gamma} = 4$
$r = 16$

Dovetail sort (DTSort)

Bucketing high frequency (heavy) values, recursing over low frequency (light) values, merging.

Random sampling

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

Most significant digit

[0110] 0100 0000 [0100] 1000 0010 0110 0100 0111 [1001] 1011 0101 1111 [0100] 1101 1010 1001 [0100] 1110 0101 [1001] 1011 [0110] 1001

$\gamma = 2$
$b = 2^{\gamma} = 4$
$r = 16$

Dovetail sort (DTSort)

Bucketing high frequency (heavy) values, recursing over low frequency (light) values, merging.

Random sampling

Input $\boxed{6}$ 4 0 $\boxed{4}$ 8 2 6 4 7 $\boxed{9}$ 11 5 15 $\boxed{4}$ 13 10 9 $\boxed{4}$ 14 5 $\boxed{9}$ 11 $\boxed{6}$ 9

Most significant digit

Sort by most significant digit ($\gamma = 2$)

00	01	10	11

0000 0010 $\boxed{0110}$ 0100 $\boxed{0100}$ 0110 0100 0111 0101 $\boxed{0100}$ $\boxed{0100}$ 0101 $\boxed{0110}$ 1000 $\boxed{1001}$ 1011 1010 1001 $\boxed{1001}$ 1011 1001 1111 1101 1110

$\gamma = 2$
$b = 2^{\gamma} = 4$
$r = 16$

Dovetail sort (DTSort)

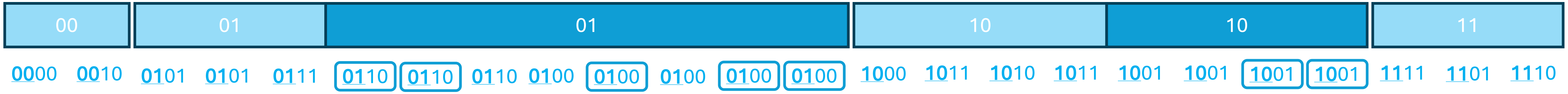
Bucketing high frequency (heavy) values, recursing over low frequency (light) values, merging.

Random sampling

Input $\boxed{6}$ 4 0 $\boxed{4}$ 8 2 6 4 7 $\boxed{9}$ 11 5 15 $\boxed{4}$ 13 10 9 $\boxed{4}$ 14 5 $\boxed{9}$ 11 $\boxed{6}$ 9

Most significant digit

Sort into light and heavy buckets



$\gamma = 2$
$b = 2^{\gamma} = 4$
$r = 16$

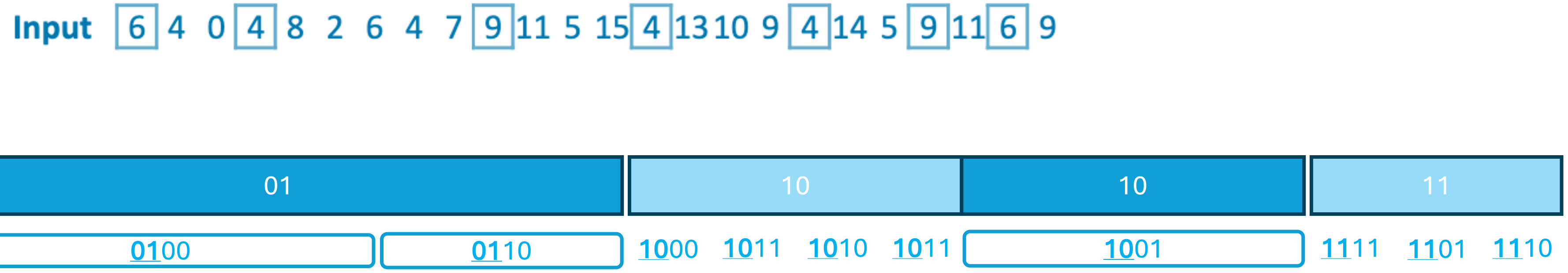
Dovetail sort (DTSort)

Bucketing high frequency (heavy) values, recursing over low frequency (light) values, merging.

Random sampling

Most significant digit

Sort heavy buckets



$\gamma = 2$
$b = 2^{\gamma} = 4$
$r = 16$

Dovetail sort (DTSort)

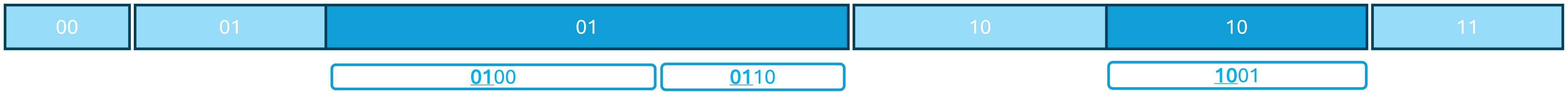
Bucketing high frequency (heavy) values, recursing over low frequency (light) values, merging.

Random sampling

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

Most significant digit

Sort heavy buckets



Recurse over light buckets (could resample but we reached base case)

$\gamma = 2$
$b = 2^\gamma = 4$
$r = 16$

Dovetail sort (DTSort)

Bucketing high frequency (heavy) values, recursing over low frequency (light) values, merging.

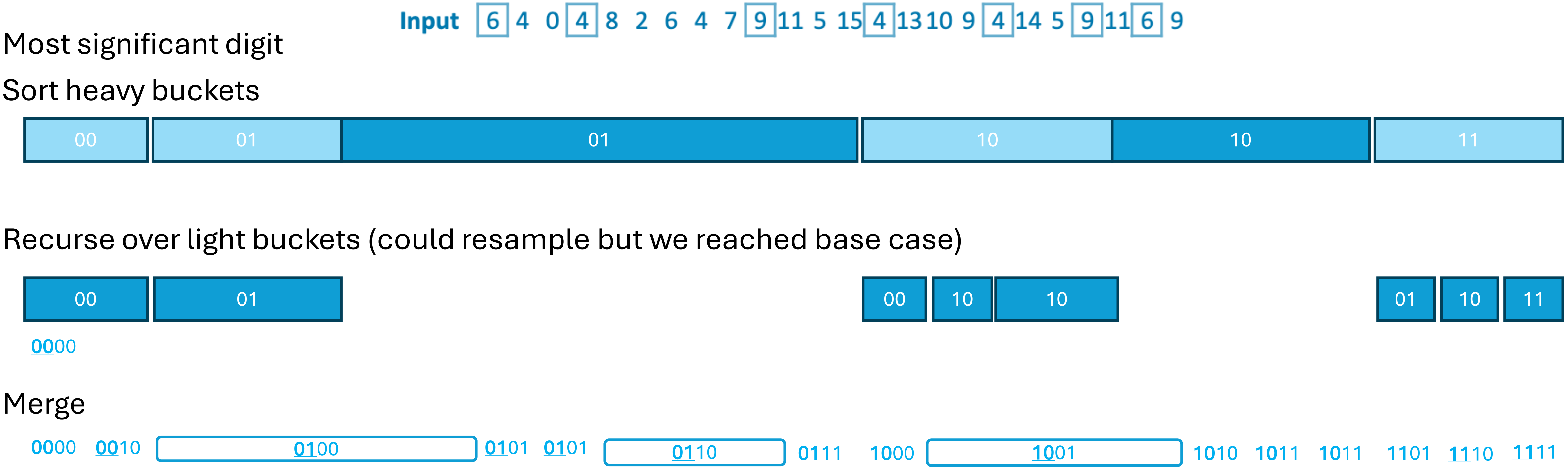
Random sampling

Most significant digit

Sort heavy buckets

Recurse over light buckets (could resample but we reached base case)

Merge



$\gamma = 2$
$b = 2^{\gamma} = 4$
$r = 16$

Dovetail sort (DTSort)

Bucketing high frequency (heavy) values, recursing over low frequency (light) values, merging.

Random sampling

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

Most significant digit

Sort heavy buckets

Recurse over light buckets (could resample but we reached base case)

0000

Merge

0000 0010 0100 0101 0101 0110 0111 1000 1001 1010 1011 1011 1101 1110 1111

Base 10 representation

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

Dovetail sort: Sampling

Sampling (find heavy keys & assign buckets):

$S \leftarrow \Theta(2^{\gamma} \log n)$ sampled keys from A

Sort S, subsample every $(\log n)$-th key, and store the keys that have more than one subsamples into S'

for $i \leftarrow 0$ *to* $|S'| - 1$ **do** *// each heavy keys' MSD and its key*

  $h[i] \leftarrow \langle$the d-th digit in $S'[i], S'[i]\rangle$

for $i \leftarrow 0$ *to* $2^{\gamma} - 1$ **do** *// each light bucket's MSD and a dummy key*

  $l[i] \leftarrow \langle i, -1\rangle$

Merge h and l into an array *buckets*

Initialize the hash table H and lookup table L

for $i \leftarrow 0$ *to* $|buckets|$ **do** *// assign bucket ids in order*

  $\langle x, y\rangle \leftarrow buckets[i]$

  if $y = -1$ **then** $L[x] \leftarrow i$ *// light bucket and its id*

  else Insert key y with value i to H *// heavy bucket and its id*

Inspired by Rajasekaran and Reif, 1989

< Previous Article   Next Article >

FULL ACCESS

Optimal and Sublogarithmic Time Randomized Parallel Sorting Algorithms

Authors: Sanguthevar Rajasekaran and John H. Reif | AUTHORS INFO & AFFILIATIONS

https://doi.org/10.1137/0218041

PDF BibTeX Tools

Abstract

This paper assumes a parallel RAM (random access machine) model which allows both concurrent reads and concurrent writes of a global memory.

The main result is an optimal randomized parallel algorithm for INTEGER_SORT (i.e., for sorting *n* integers in the range $[1,n]$). This algorithm costs only logarithmic time and is the first known that is optimal: the product of its time and processor bounds is upper bounded by a linear function of the input size. Also given is a deterministic sublogarithmic time algorithm for prefix sum. In addition this paper presents a sublogarithmic time algorithm for obtaining a random permutation of n elements in parallel. And finally, sublogarithmic time algorithms for GENERAL_SORT and INTEGER_SORT are presented. Our sub-logarithmic GENERAL_SORT algorithm is also optimal.

Dovetail sort: Sampling

Sampling (find heavy keys & assign buckets):

$S \leftarrow \Theta(2^\gamma \log n)$ sampled keys from A
Sort S, subsample every $(\log n)$-th key, and store the keys that have more than one subsamples into S'
for $i \leftarrow 0$ *to* $|S'| - 1$ **do** *// each heavy keys' MSD and its key*
  $h[i] \leftarrow \langle$the d-th digit in $S'[i], S'[i]\rangle$
for $i \leftarrow 0$ *to* $2^\gamma - 1$ **do** *// each light bucket's MSD and a dummy key*
  $l[i] \leftarrow \langle i, -1\rangle$
Merge h and l into an array *buckets*
Initialize the hash table H and lookup table L
for $i \leftarrow 0$ *to* $|buckets|$ **do** *// assign bucket ids in order*
  $\langle x, y\rangle \leftarrow buckets[i]$
  if $y = -1$ **then** $L[x] \leftarrow i$ *// light bucket and its id*
  else Insert key y with value i to H *// heavy bucket and its id*

1. Set parameter p, input size n
2. S ← uniformly select ($p \log n$) random samples ($2^\gamma \log n$)
3. S' ← subsample $\log n$-th key, add if multiple occurrences

If a key appears more than once in S' it is likely to have $\Omega(n/p)$ occurrences ($\Omega(n/2^\gamma)$)

Keep heavy and light buckets together in same MSD zone (light first, then sorted heavy)

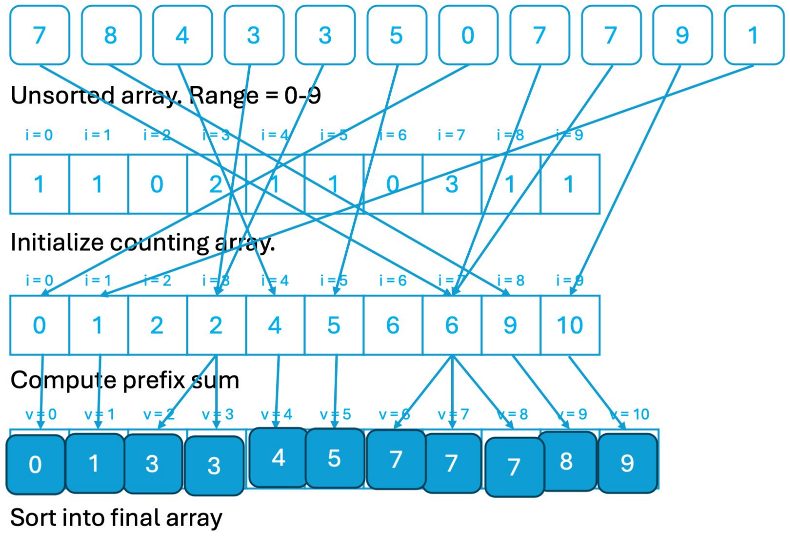
Hash map for heavy bucket ID (zone, key) and associated light bucket

Dovetail sort: Distributing

Distributing (Reorder A by bucket ids):

15 Use counting sort on A with key function GetBucketID (line 21)

Counting sort with bucket ID as key to order records into buckets



Dovetail sort: Recursing

Recursing (sort light buckets):

parallel_for_each *light bucket B* **do** DTSort($B, d-1$)

Recursively sort each light bucket

Keep identifying heavy keys! Will make distribution much more lightweight.

Heavy heavy → medium heavy → light heavy through levels of recursion

Status check

1. All keys sorted into MSD zones
2. All heavy keys in a zone sorted
3. All light records sorted within the light bucket

Dovetail Merging (interleave light and heavy buckets):

parallel_for $i \in [2^\gamma]$ **do** *// merge buckets for all MSD zones*
  Let $B_0, B_1, \ldots B_m$ be all buckets in MSD zone i
  DTMerge$(B_0, B_1, \ldots B_m)$???!!?
return A

Function GetBucketId(k)
  if k *is found in* H **then return** $H[k]$ **else return** $L[k]$

Sort heavy buckets

Recurse over light buckets

Merge

Dovetail merge

Goal: minimize data movement, beat parallel merge polylogarithmic span and linear work

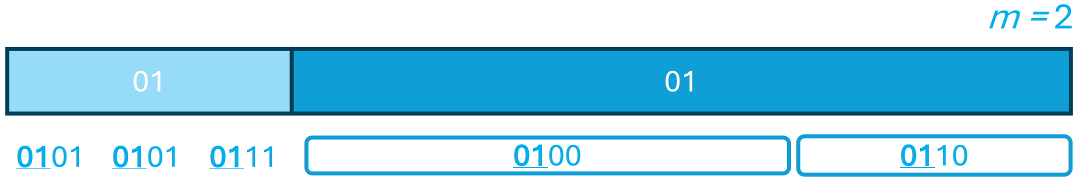
Algorithm 3: DTMerge($A, B_0, B_1, \ldots B_m$)

Input: $m+1$ buckets consecutive in array A. The first one B_0 is a sorted light bucket. The other m buckets are heavy buckets sorted by their keys.

```
Binary search all heavy keys in B_0, accordingly get p_{1..m}, where
  p_m is the starting index for heavy bucket B_i when the array is
  fully sorted.
if |B_0| ≤ Σ_{i=1}^m |B_i| then                 // more heavy keys than light
    copy B_0 to array T
    for i ← 1 to m do                    // starting from the first heavy bucket
        // Move B_i to the final positions starting from A[p_i]
        if The final positions for B_i overlap with the current
          positions of B_i then
            Let s be the current start point of B_i
            flip(A, s, s + |B_i|)                  // Flip the entire region
            flip(A, p_i, s + |B_i|)           // Flip the destination region
        else                                  // No overlap. Directly copy
            ParallelForEach j ← 0 to |B_i| − 1 do
                Copy the j-th record in B_i to A[p_i + j]
    Move light records from T to their final positions in A
else Symmetric case: copy heavy keys out and merge
```

$m+1$ buckets (maximum $2^\gamma + 1$)

Sort serially within each bucket!

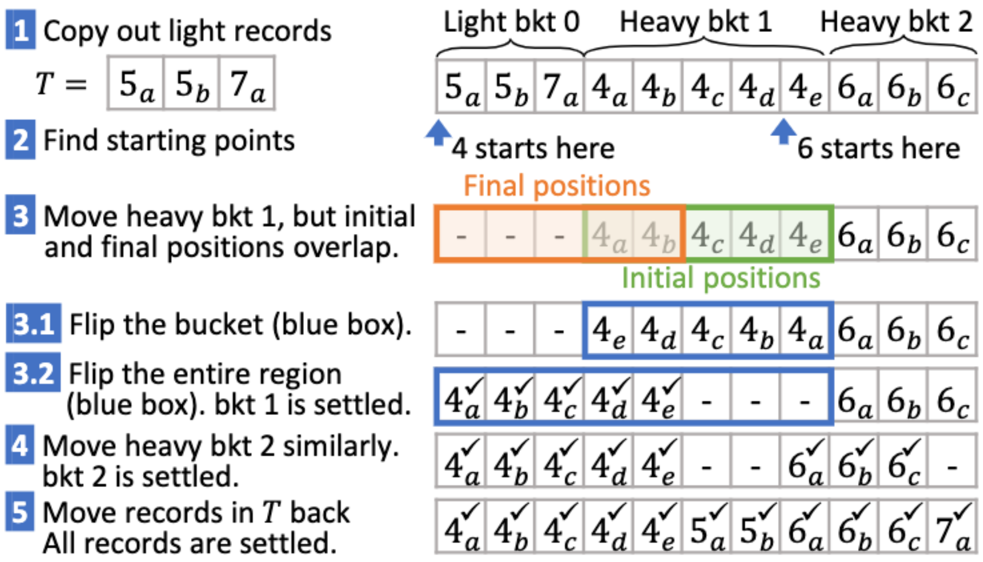


1. Copy smaller of light/heavy into T
2. Safely/in-place move remaining keys into final positions

3. Copy T keys back in

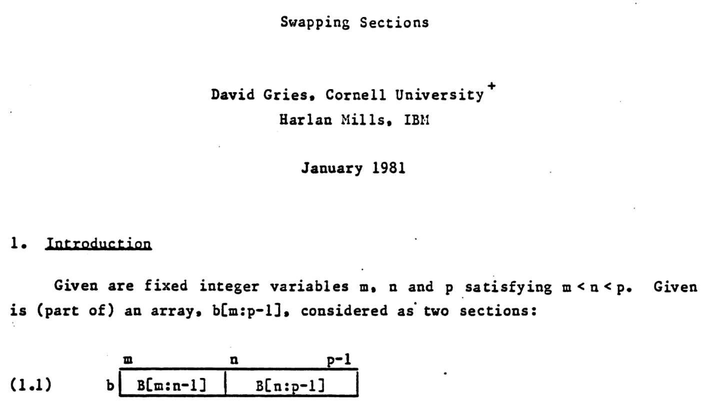
Dovetail merge: reshuffle safe and in-place?

How to get bucket B_i from one place to the next while minimizing data movement?



Scenarios:

1. B_i contains light records ☑ backed up in T!
2. New position of B_i overlaps with earlier bucket $B_{i'}$ ☑ $B_{i'}$ has already been moved!
3. New position of B_i overlaps with its own original position Uh oh! Circular shift algorithm!



Swapping Sections

David Gries, Cornell University+
Harlan Mills, IBM

January 1981

1. Introduction

Given are fixed integer variables m, n and p satisfying m < n < p. Given is (part of) an array, b[m:p-1], considered as two sections:

	m	n	p-1
(1.1) b	B[m:n-1]	B[n:p-1]	

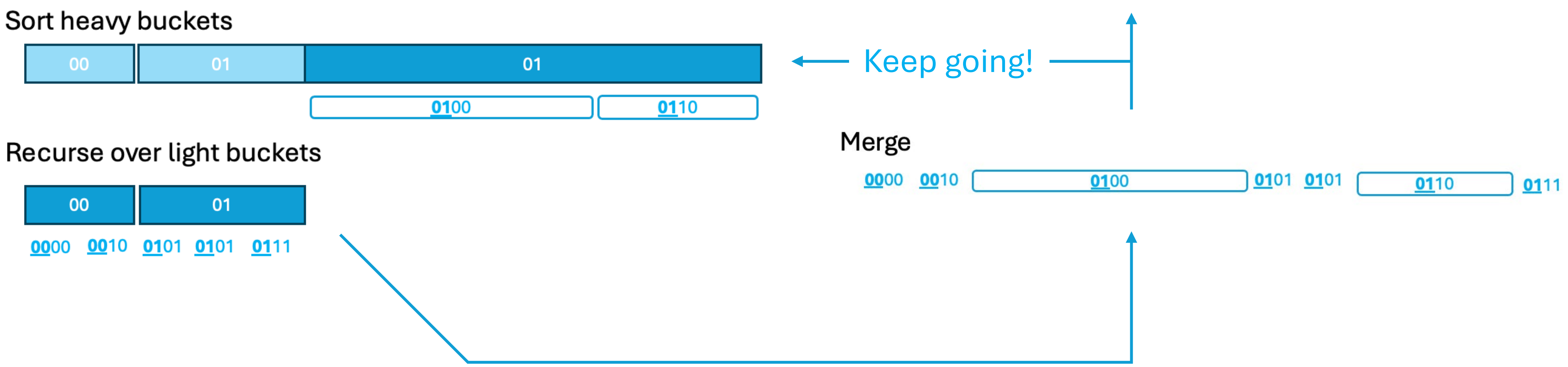
Status check

1. All keys sorted into MSD zones
2. All heavy keys in a zone sorted
3. All light records sorted within the light bucket

Done!

Base case?

- All digits are sorted
- Problem size $n < 2^{2\gamma}$ (use comparison sort)



Theory

Theorem 4.1. *There exists an unstable parallel MSD sorting algorithm with $O(n\sqrt{\log r})$ work and $O(\log r + \sqrt{\log r}\log n)$ span* whp.

Lemma 4.2. *With the chosen parameters, the total work for base cases (line 2) in Alg. 1* is $O(n\sqrt{\log r})$.

Lemma 4.3. *With the chosen parameters, the distribution step (line 3) Alg. 1 in all recursive calls has work $O(n\sqrt{\log r})$* whp.

Theorem 4.4. *There exists a stable parallel MSD sorting algorithm with $O(n\sqrt{\log r})$ work and $O(2^{\sqrt{\log r}}\sqrt{\log r})$ span.*

Theorem 4.5. *The DTSort algorithm (Alg. 2) is a stable integer sort with $O(n\sqrt{\log r})$ work and $\tilde{O}(2^{\sqrt{\log r}})$ span.*

Theorem 4.6. *DTSort has $O(n)$ work* whp *if the input key frequency exhibits an exponential distribution with $\lambda e^{\lambda} \geq \bar{c}/2^{\gamma}$ for some $\bar{c} > 1$. Here $\lambda > 0$ is the parameter of the exponential distribution, which gives probability density function $f(x;\lambda) = \lambda e^{-\lambda x}$ for $x > 0$.*

Theorem 4.7. *DTSort has $O(n)$ work* whp *if there are no more than $c'2^{\gamma}$ distinct keys, for some constant $c' < 1$.*

Prove that parallel integer sort >>> comparison sort

Prove that the same applies to Dovetail sort

Special input distributions!

Theory-Guided Practice. Thm. 4.4 shows $O(n\sqrt{\log r})$ work for practical parallel MSD sort. This explains why the integer sort algorithms can outperform comparison sorts with $O(n\log n)$ work for realistic range of $r = n^{O(1)}$. Our parameter $\gamma = \sqrt{\log r}$ used in the analysis is also the choice for most practical MSD algorithms. When $r = 2^{64}$, we have $\gamma = \sqrt{\log r} = 8$ and $\theta = 2^{c\gamma} = 2^{8c}$, which roughly matches the existing algorithms that empirically sort 8–12 bits in each recursive level, and use base case sizes from 2^8 to 2^{16} [5, 9, 43].

Theory

Why the **theory** and practice of parallel integer sort?

Recall! main use case for integer sort
Else counting sort is better

Best work bound so far:

Comparison sort where r = Ω(n):

$$O(n \log r) \longrightarrow O(n \log r)$$

Theoretically-Efficient and Practical Parallel In-Place Radix Sorting

hi julian

Omar Obeya
MIT CSAIL
obeya@mit.edu

Endrias Kahssay
MIT CSAIL
endrias@mit.edu

Edward Fan
MIT CSAIL
edwardf@mit.edu

Julian Shun
MIT CSAIL
jshun@mit.edu

… really shouldn't be the same given practical performance of parallel IS

Start by closing theoretical gap and get it down to $O(n\sqrt{\log r})$

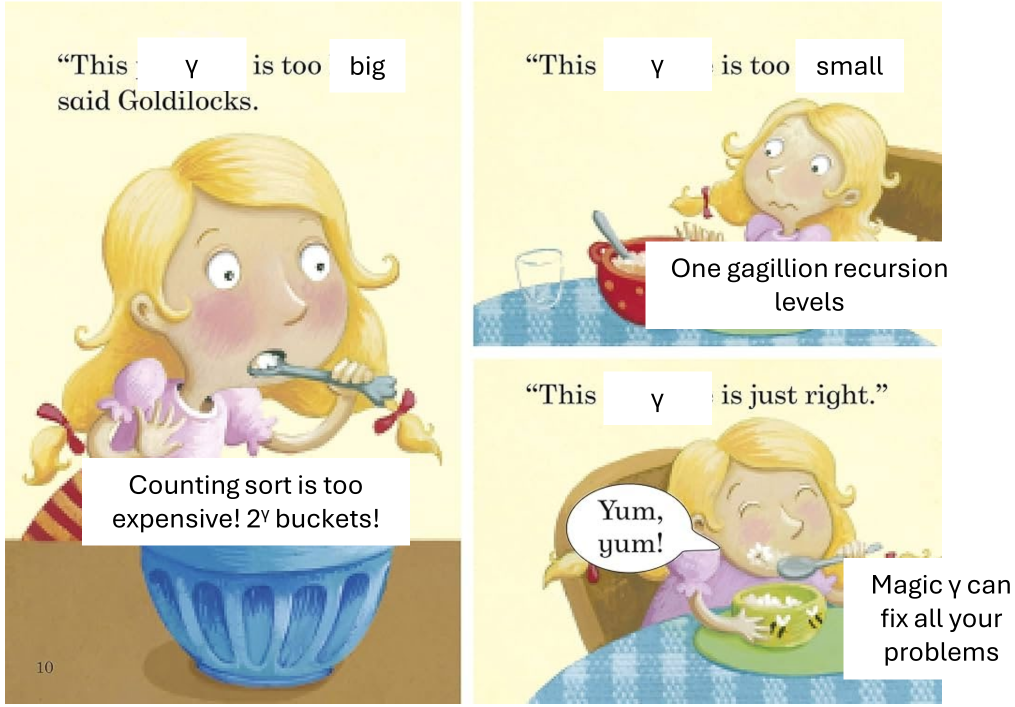
Theory

SETUP
DIGIT: γ bits
RADIX: 2^γ bits

Theorem 4.1. *There exists an unstable parallel MSD sorting algorithm with $O(n\sqrt{\log r})$ work and $O(\log r + \sqrt{\log r}\log n)$ span* whp.

Theorem 4.4. *There exists a stable parallel MSD sorting algorithm with $O(n\sqrt{\log r})$ work and $O(2^{\sqrt{\log r}}\sqrt{\log r})$ span.*

Problem: if key has log r bits and a digit is γ bits then n levels = log r / γ. How to pick γ?



Read It Yourself, 2013

Theory

SETUP
DIGIT: γ bits
RADIX: 2^γ bits

Theorem 4.1. *There exists an unstable parallel MSD sorting algorithm with $O(n\sqrt{\log r})$ work and $O(\log r + \sqrt{\log r}\log n)$ span* whp.

Theorem 4.4. *There exists a stable parallel MSD sorting algorithm with $O(n\sqrt{\log r})$ work and $O(2^{\sqrt{\log r}}\sqrt{\log r})$ span.*

Key insight: $\gamma = \sqrt{\log r}$ allows us to bound the number of recursive levels to

$$(\log r)/\gamma = O(\sqrt{\log r}) = O(\gamma)$$

So each element only participates in $\sqrt{\log r}$ levels, v.s. $\log r$!!!

Non-recursive work per level (distribution, comparison sort) has at most O(n)

Now total work (work per level * number of levels) = $O(n\sqrt{\log r})$

Clever First Fairytales

Theory

Does this general theory apply to Dovetail sort as well?

Need to show that sampling and Dovetail merge don't add higher-order costs.

Theorem 4.5. *The DTSort algorithm (Alg. 2) is a stable integer sort with $O(n\sqrt{\log r})$ work and $\tilde{O}(2^{\sqrt{\log r}})$ span.* ✅

Without sampling and merging, n' size problem has $O(n')$ work

Cost of sorting samples: $O(|S| \log |S|) = O(2^{\gamma} \log n \cdot \gamma) = o(2^{2\gamma}) = o(n')$ = overall work

Cost of Dovetail merge: $O(2^{\gamma})$ binary searches of n' records = $O(2^{\gamma} \log n') \approx O(n')$

Moving at most $2t$ records has $O(t)$ work + copying out and back $n'/2$ records = $O(n')$

= asymptotically still $O(n\sqrt{\log r})$!!!!

Theory

Special distributions: if there are a lot of duplicates we get O(n) work! (Recursion tree collapses!)

Theorem 4.6. *DTSort has $O(n)$ work* whp *if the input key frequency exhibits an exponential distribution with $\lambda e^{\lambda} \geq \bar{c}/2^{\gamma}$ for some $\bar{c} > 1$. Here $\lambda > 0$ is the parameter of the exponential distribution, which gives probability density function $f(x;\lambda) = \lambda e^{-\lambda x}$ for $x > 0$.*

Theorem 4.7. *DTSort has $O(n)$ work* whp *if there are no more than $c'2^{\gamma}$ distinct keys, for some constant $c' < 1$.*

Experiments

Tested algorithms

Name	Stable	In-place	Type	Notes
DTSort	Yes	No	Integer	Our integer sort algorithm
PLIS	Yes	No	Integer	ParlayLib integer sort [9]
IPS^{2}Ra	No	Yes	Integer	IPS^{2}Ra integer sort [5]
RS	No	Yes	Integer	RegionsSort [43]
RD	No	No	Integer	RADULS [36]
PLSS	Y/N	Y/N	Comparison	ParlayLib sample sort [9]
IPS^{4}o	No	Yes	Comparison	IPS^{4}o sample sort [5]

Table 2. Algorithms tested in our experiments. There are two versions of PLSS. Here we use the unstable but faster one. In-place means fully in-place ($o(n)$ extra memory).

Machine

2.1 GHz Intel Xeon Gold 6252 CPUs
96 cores total
1.5TB main memory

Data distributions

Unif-μ : uniform with μ distinct keys

Exp-λ : exponential with param $10^{-5}\lambda$

Zipf-s : Zipfian with param *s*

BExp-t : custom kryptonite distribution Bit-Exponential (problem sizes uneven, maximum recursion)

Experiments: numerical distributions

Instances		32-bit						64-bit						
		Integer				Comparison		Integer					Comparison	
		Ours	PLIS	IPS2Ra	RS	PLSS	IPS4o	Ours	PLIS	IPS2Ra	RS	RD	PLSS	IPS4o
Uniform	1.00E+09	**0.5**	0.537	0.671	0.718	1.27	0.69	**0.994**	1.14	1.09	1.43	1.86	1.65	1.11
	1.00E+07	**0.501**	0.549	0.6	0.705	1.14	0.604	**1.03**	1.15	1.06	1.71	1.86	1.47	1.05
	1.00E+05	**0.478**	0.542	0.595	0.696	1.08	0.653	**0.859**	1.22	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
	1.00E+03	0.506	0.505	0.538	0.613	0.805	**0.432**	0.795	1.41	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
	1.00E+01	**0.308**	0.707	1.13	0.438	0.959	0.456	**0.581**	1.93	3.78	1.26	8.25	1.12	0.85
Exponential	1.00E+00	**0.526**	0.536	0.574	0.711	1.11	0.671	**0.976**	1.16	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
	2.00E+00	**0.502**	0.546	0.577	0.711	1.12	0.661	**0.919**	1.22	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
	5.00E+00	**0.435**	0.567	0.583	0.705	1.11	0.612	**0.819**	1.52	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
	7.00E+00	**0.419**	0.582	0.554	0.708	1.08	0.609	**0.782**	1.69	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
	1.00E+01	**0.402**	0.603	0.56	0.682	1.09	0.561	**0.763**	1.87	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
Zipfian	6.00E-01	**0.493**	0.543	0.63	0.72	1.23	0.691	**1**	1.14	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
	8.00E-01	**0.524**	0.542	0.619	0.71	1.2	0.67	**1**	1.18	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
	1.00E+00	0.601	0.631	0.648	0.735	1.08	**0.59**	**1.04**	1.44	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
	1.20E+00	**0.516**	0.832	1.07	0.709	1.1	0.743	**0.918**	1.95	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
	1.50E+00	**0.446**	0.946	1.9	0.695	1.48	0.939	**0.883**	2.56	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
	Avg	**0.472**	0.601	0.698	0.679	1.11	0.629	**0.882**	1.46	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
Bit Exponential	1.00E+01	1.11	0.833	1.38	0.841	0.857	**0.61**	3.3	2.7	3.31	1.89	8.03	1.57	**1.08**
	3.00E+01	**0.643**	1.08	3.18	0.775	1.27	0.908	2.75	4.04	7.9	2.3	12.8	1.36	**1.26**
	5.00E+01	**0.55**	1.2	4.29	1.12	1.51	0.77	1.85	4.57	11.9	2.27	9.47	1.74	**1.45**
	1.00E+02	**0.512**	1.31	5.89	0.664	1.99	1.48	**1.42**	4.92	17.8	2.2	4.96	2.44	2.03
	3.00E+02	**0.616**	1.4	8.22	0.606	2.32	2.02	**1.32**	5.25	27.9	2.12	4.15	3.26	3.31
	Avg	**0.659**	1.15	3.91	0.716	1.52	1.12	1.99	4.19	10.9	2.15	7.25	1.97	**1.68**

* Other integer sort algorithms are competitive with DTSort on light distributions but much slower on heavy

* Integer sorting algorithms affected more dramatically by increase from 32-64 bits than comparison due to work bound dependence on r

* Worst performance happens on Unif-10^3 when γ is close to threshold for finding heavy keys at root level (light keys are pretty heavy)

* Bit Exponential causes huge load imbalance as none of the MSD zones are of similar size. Heavily affects IS, especially stable IS. Merging step in DTSort takes ~50% of running time.

Table 3. Running time (in seconds) on synthetic data with $n = 10^9$. The instances of one distribution is ordered by increasing number of heavy records from top to bottom. The fastest running time on each input instance is underlined. "Avg." = geometric mean. The keys of RD need to be padded to multiples of 64 bits. We remove it from the 32-bit experiments because it is too slow after padding to 64 bits.

Experiments: real world use-cases

			Integer				Comparison	
	Instances	n	Ours	PLIS	IPS2Ra	RS		
Graph transpose	LiveJournal	69.0M	**0.043**	0.043	s.g.	[illegible]	[illegible]	[illegible]
	Twitter	1.47B	**0.888**	0.942	3.24	[illegible]	[illegible]	[illegible]
	Cosmo50	1.61B	**0.782**	0.945	1.41	[illegible]	[illegible]	[illegible]
	sd_arc	2.04B	**1.1**	1.29	2.87	[illegible]	[illegible]	[illegible]
	Clueweb	42.6B	28.5	37	32.5	s.g.	[illegible]	[illegible]
	Avg.	-	**0.985**	1.13	-	-	1.96	1.37
Morton order	GeoLife	24.9M	0.026	0.028	0.259	**0.024**	0.028	0.171
	Cosmo50	321M	0.184	**0.178**	0.343	0.209	0.327	0.338
	OpenStreetMap	2.77B	2.32	2.39	3.65	s.g.	2.73	**1.53**
	Avg.	-	**0.223**	0.227	0.687	-	0.293	0.445
MO synth (Varden)	SS2d	1B	**0.498**	0.557	0.662	0.634	1.27	0.775
	SS3d	1B	**0.512**	0.568	0.778	0.611	1.12	0.754
	SS2d'	2B	**0.973**	1.16	1.27	1.17	2.44	1.39
	SS2d"	2B	**0.99**	1.6	2.86	1.17	2.3	1.96
	Avg.	-	**0.704**	0.875	1.17	0.854	1.68	1.12

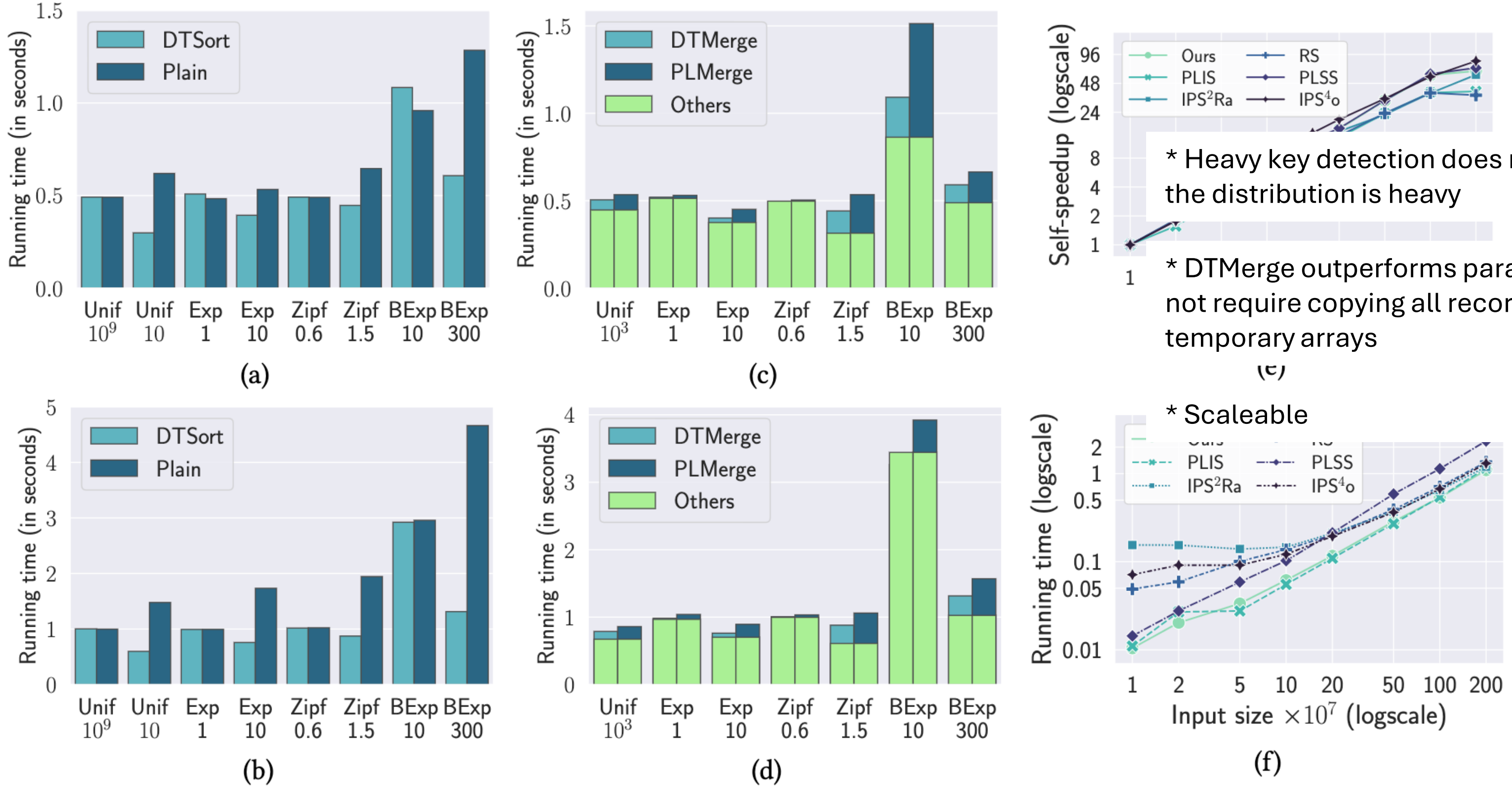
* DTSort performs best on graph transpose except for ClueWeb (largest), where SampleSort beats out

* Morton Order on real-world graphs done best by existing integer sort algorithms/comparison sort, but best average because no "worst case" performance.

Table 4. Running time (in seconds) on multiple applications. We use 32-bit keys and 32-bit values. See Sec. 6.2 for the detailed explanation on the keys and values. The fastest running time on each instance is underlined. "*n*" = input sizes. "s.g." = segmentation fault. "Avg." = the geometric mean on instances of the same application. "-" = not applicable.

Cosmo50 is k-NN graph with relatively evenly distributed degrees

Experiments: in-depth breakdown



* Heavy key detection does make a difference when the distribution is heavy

* DTMerge outperforms parallel merge because it does not require copying all records between input and temporary arrays

* Scaleable

Figure 4. (a) and (b): Analysis for the performance of heavy-key detection. Numbers are running time (lower is better) with or without heavy-key detection. (a) is for 32-bit keys and (b) is for 64-bit keys. (c) and (d): Analysis for the performance of dovetail merging. Numbers are running time (lower is better) using our dovetail merging algorithm or a baseline merging algorithm. (c) is for 32-bit keys and (d) is for 64-bit keys. (e) and (f): Scalability (higher is better) with varying number of threads and running time (lower is better) with varying input sizes on 32-bit key and 32-bit value pairs on one instance: *Zipf-0.8*. Full analysis is given in the full paper [21]. Discussions are in Sec. 6.3.

Conclusion + discussion questions

Strengths

- Extremely thorough and takes on a significant challenge (make a better algorithm and provide first theoretical bounds for whole family of parallel IS algorithms to explain why they empirically outperform comparison sort)
- Really elegant approach (magic γ!)
- Successful algorithm – removes worst-case behavior and can improve average-case speed too
- Comprehensive and fair experiments

Weaknesses

- Analysis ignores NUMA, cache line alignment, etc. (integer sort is memory bound – how does memory subsystem impact the algorithm performance?)
- Dovetail merge is complex to implement (alas)

Questions

- Why stick to MSD? Could LSD/MSD local/global hybrid improve work/span bounds?
- Is counting sort really always better for small r?
- Could we adaptively pick γ?
- How might dovetail merge be applied to other contexts? Parallel graph algorithms, etc.