Parallel Integer Sort - Theory and Practice (appeared at PPoPP ’24)

Xiaojun Dong Laxman Dhulipala Yan Gu Yihan Sun

Full Version: https://arxiv.org/abs/2401.00710 Code: https://github.com/ucrparlay/DovetailSort

Problem Definitions and Notations

- Given a sequence of integers A of size n in the range $[0, r]$, order the elements by their integer keys.

$A[1..n]$	Original input array with size n
$[r]$	The range of the integer keys $0, \ldots, r-1$
γ	Number of bits in a “digit” (sorted by each level)
θ	Base case size threshold
d	Number of remaining “digits” to be sorted

Challenges and Our Contributions

- Theoretical challenge: the best work bound of practical implementations for the range of $[0, r]$ is $O(n \log r)$, which is no better than comparison sort when $r = \Omega(n)$. 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

DovetailSort

	DTSort$(A[0..n-1], d)$
Base Cases	**if** $d = 0$ **then return** A **if** $\lvert A \rvert < \theta$ **then return** *ComparisonSort*(A)
Sampling	Detect heavy keys by sampling
Distributing	Distribute each integer to buckets using the d-th digit as the bucket id
Recursing	**Parallel_for_each bucket** $\boldsymbol{B}$ *DTSort*$(B, d-1)$
Merging	Merge the heavy keys with the light keys

- Our algorithm follows the most-significant digit (MSD) framework (given by the black boxes) that partitions all keys into buckets based on the integer encoding (i.e., $\lambda \in [8,12]$ highest bits), and recurses within each bucket.
- Our algorithm can detect heavy duplicate keys by sampling and take advantage of them by avoiding sorting them in subsequent recursive levels. The heavy keys will be merged with the light keys after the recursion.
- We propose DTSort to accelerate the merge subroutine (more details given in the paper).

Experimental Results

- Experimental setup: 96 cores and 1.5TB of main memory.
- Baselines algorithms:
  - Integer sort: PLIS: ParlayLib integer sort [3] (**PLIS**), IPS2Ra integer sort [2] (**IPS^{2}Ra**), Region sort [7] (**RS**)
  - Comparison sort: ParlayLib sample sort [3,4] (**PLSS**), IPS4o sample sort [2] (**IPS^{4}o**)

		Integer				Comparison	
		Ours	**PLIS**	**IPS^{2}Ra**	**RS**	**PLSS**	**IPS^{4}o**
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

Underline: stable

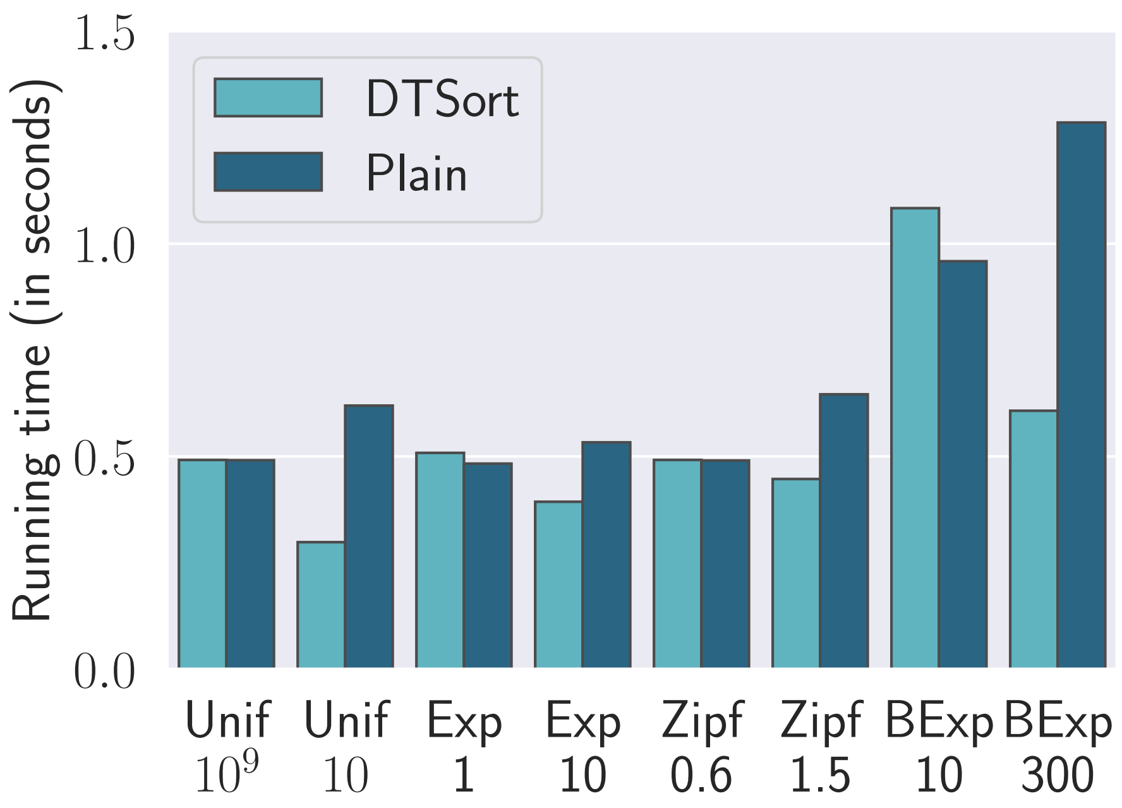
Avg. = GeoMean

Fewer duplicates

More duplicates

1
1.1
1.2
1.5
2
4
>4

- The heatmap shows relative speedups over the fastest implementation on each input distribution with $n = 10^9$.
- DTSort achieves the best performance on 13 out of 15 test cases and a 28% speedup over the next best implementation on geometric mean average.
- The bar graph shows the running time of our code with (**DTSort**) and without (**Plain**) heavy key detection with $n = 10^9$.



- The overhead of doing sampling and merging is very small (See Unif-10^9).
- It can achieve up to 2x speedup compared to Plain (See BExp-300).

References

[1] Susanne Albers and Torben Hagerup. 1997. Improved parallel integer sorting without concurrent writing. *Information and Computation* 136, 1 (1997), 25–51.
[2] Michael Axtmann, Sascha Witt, Daniel Ferizovic, and Peter Sanders. 2022. Engineering in-place (shared-memory) sorting algorithms. *ACM Transactions on Parallel Computing (TOPC)* 9, 1 (2022), 1–62.
[3] Guy E. Blelloch, Daniel Anderson, and Laxman Dhulipala. 2020. ParlayLib — a toolkit for parallel algorithms on shared-memory multicore machines. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*.
[4] Guy E. Blelloch, Phillip B. Gibbons, and Harsha Vardhan Simhadri. 2010. Low depth cache-oblivious algorithms. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*.
[5] Xiaojun Dong, Laxman Dhulipala, Yan Gu, and Yihan Sun. 2024. Parallel Integer Sort: Theory and Practice. In *ACM Symposium on Principles and Practice of Parallel Programming (PPOPP)*.
[6] Marek Kokot, Sebastian Deorowicz, and Maciej Dlugosz. [n. d.]. Even Faster Sorting of (Not Only) Integers. In *Man-Machine Interactions 5 - 5th International Conference on Man-Machine Interactions, ICMMI 2017, Kraków, Poland, October 3-6, 2017*.
[7] Omar Obeya, Endrias Kahssay, Edward Fan, and Julian Shun. 2019. Theoretically-Efficient and Practical Parallel In-Place Radix Sorting. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. 213–224.
[8] Uzi Vishkin. 2010. Thinking in parallel: Some basic data-parallel algorithms and techniques. (2010).

Algorithm Overview

Step 1: Take samples (boxed), detect heavy keys, assign bucket ids

Samples: $\boxed{4} \times 3$ $\boxed{2} \times 1$ $\boxed{6} \times 2$ $\boxed{9} \times 2$ $\boxed{7} \times 1$

$\Rightarrow$ 4 light buckets: for MSD (highest two bits) 00, 01, 10, 11
3 heavy buckets: for 4, 6, 9

Step 2: Distribute records to corresponding buckets

Step 3: Recursively integer sort each light bucket on the next 2 bits

Step 4: Merge heavy and light buckets within the same MSD