Parallel Integer Sort: Theory and Practice

By: Xiaojun Dong, Laxman Dhulipala, Yan Gu, and Yihan Sun

Reviewed By: Jonathan Li

Previous Integer Sorting

Previous integer sorting algorithms have a large disparity between their theoretical and practical performance.

> Generally, integer sorting has better theoretical bounds since algorithms can use integer encoding of keys.
>
> However, these algorithms are fairly complicated and there exist few implementations of them.

Motivation for DovetailSort

Current parallel integer sorting algorithms use a method called MSD where elements are distributed to buckets based off their most significant digit and then sorted recursively.

Samplesort and semisort algorithms use sampling to determine heavy keys and light keys, allowing duplicates to be placed in the same bucket which requires no sorting.

DTSort combines these two ideas so that it takes advantage of duplicates while still sorting in order.

Notation

$A[1..n]$	Original input array with size n
$[r]$	The range of the integer keys $0, \dots, r-1$
γ	Number of bits in a "digit" (sorted by each level)
$b = 2^{\gamma}$	The "radix" size
θ	Base case size threshold
n'	Current (recursive) problem size
d	Number of remaining "digits" to be sorted

General Overview of DTSort

There are 4 steps to DTSort

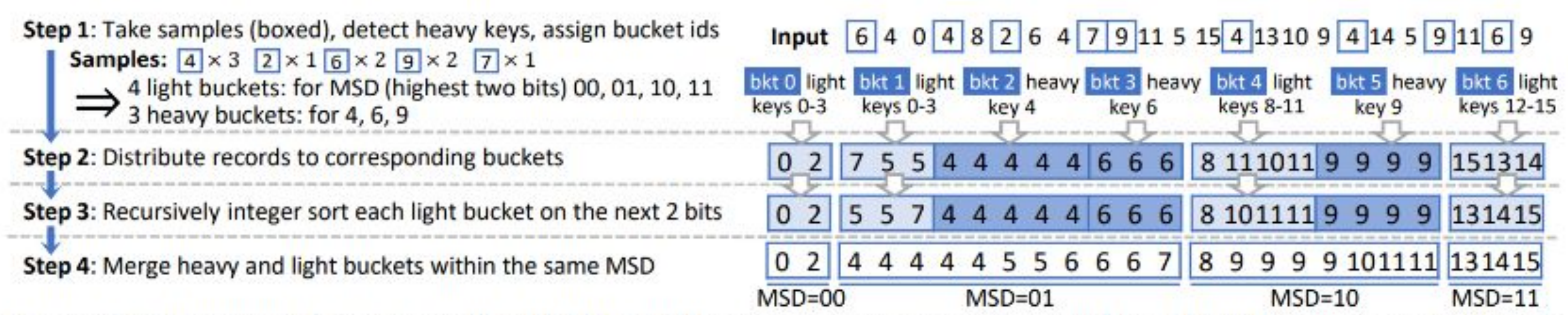
Step 1: Sampling

Step 2: Distributing

Step 3: Recursing

Step 4: Dovetail Merging

DTSort Example



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

Samples: 4 × 3 2 × 1 6 × 2 9 × 2 7 × 1

⇒ 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

Step 1: Sampling

We first select $\Theta(2^\gamma \log n)$ samples from the original array into an array S, where γ is the number of bits in each digit from before and n is the size of the array.

S is sorted and then we subsample every (log n)-th key in S, and any output keys which are duplicates are placed into another array S'.

Chernoff bounds state that any element in S' has $\Omega(n/2^\gamma)$ occurrences in the input array so these elements are now heavy keys. All other keys are light keys.

Light keys are given their corresponding MSD zone bucket id whereas each heavy key gets its own bucket id.

Step 2: Distributing

Now that each key has a bucket id, we use stable counting sort to distribute each key to its proper bucket.

Step 3: Recursing

Every MSD zone is sorted recursively using DTSort again, until all records in a MSD zone are fully sorted.

This is only needed for the light buckets since heavy buckets all contain the same key.

Due to recursion, there are multiple levels of heavy keys, with the most frequent keys detected in the first few levels and less frequent heavy keys detected in later levels.

Step 4: Dovetail Merging

There are 4 substeps to dovetail merging:

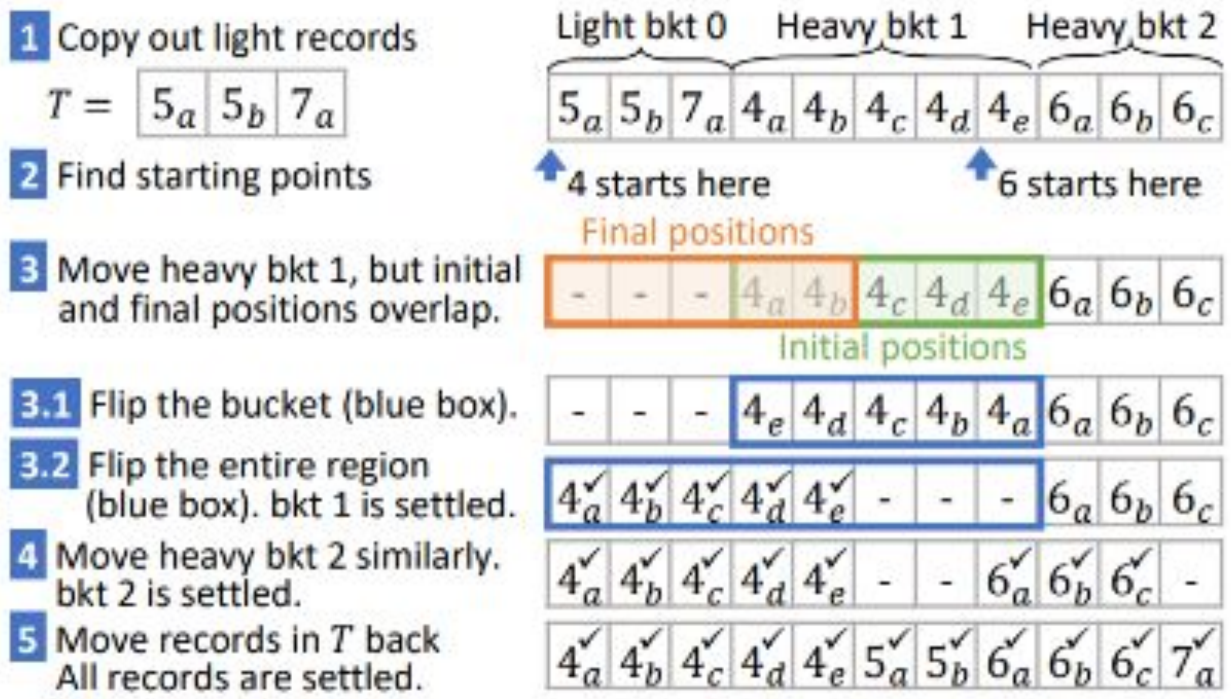
Step 1: Copy out light records within MSD.

Step 2: Find starting points of heavy keys.

Step 3: Move the heavy buckets to the corresponding location.

Step 4: Move the light records back into the correct locations.

Dovetail Merging Example



1 Copy out light records

$T = $ | 5_a | 5_b | 7_a |

2 Find starting points

3 Move heavy bkt 1, but initial and final positions overlap.

3.1 Flip the bucket (blue box).

3.2 Flip the entire region (blue box). bkt 1 is settled.

4 Move heavy bkt 2 similarly. bkt 2 is settled.

5 Move records in T back All records are settled.

Light bkt 0
Heavy bkt 1
Heavy bkt 2
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
6 starts here
Final positions
- - - 4_a 4_b 4_c 4_d 4_e 6_a 6_b 6_c
Initial positions
- - - 4_e 4_d 4_c 4_b 4_a 6_a 6_b 6_c
$4_a^{\checkmark}$ $4_b^{\checkmark}$ $4_c^{\checkmark}$ $4_d^{\checkmark}$ $4_e^{\checkmark}$ - - - 6_a 6_b 6_c
$4_a^{\checkmark}$ $4_b^{\checkmark}$ $4_c^{\checkmark}$ $4_d^{\checkmark}$ $4_e^{\checkmark}$ - - $6_a^{\checkmark}$ $6_b^{\checkmark}$ $6_c^{\checkmark}$ -
$4_a^{\checkmark}$ $4_b^{\checkmark}$ $4_c^{\checkmark}$ $4_d^{\checkmark}$ $4_e^{\checkmark}$ $5_a^{\checkmark}$ $5_b^{\checkmark}$ $6_a^{\checkmark}$ $6_b^{\checkmark}$ $6_c^{\checkmark}$ $7_a^{\checkmark}$

DTSort Analysis

DTSort is a stable integer sort with $O(n \sqrt{(\log r)})$ work and $O(2^{\sqrt{(\log r)}} \text{polylog}(r))$ span.

Step 1: Sampling takes $o(n')$ work and $o(\text{polylog } n')$ span

Step 2: Distributing takes $O(n')$ work and $O(r' + \log n')$ span

Step 3: Recursing takes $O(n \sqrt{(\log r)})$ work and $O(2^{\sqrt{(\log r)}} \sqrt{(\log r)})$

Step 4: Merging takes $O(2^{\gamma} \log n')$ work and $O(2^{\sqrt{(\log r)}} \sqrt{(\log r)})$

Optimization: Overflow Bucket

Records aren’t necessarily going to be as large as their size, i.e. key might be 32-bit integers but the actual range is much smaller.

When sampling, set the upper key range to be the largest value sampled. This usually results in a lower recursion depth.

Any values which are above the new upper range are placed into an overflow bucket which is sorted by comparison sort.

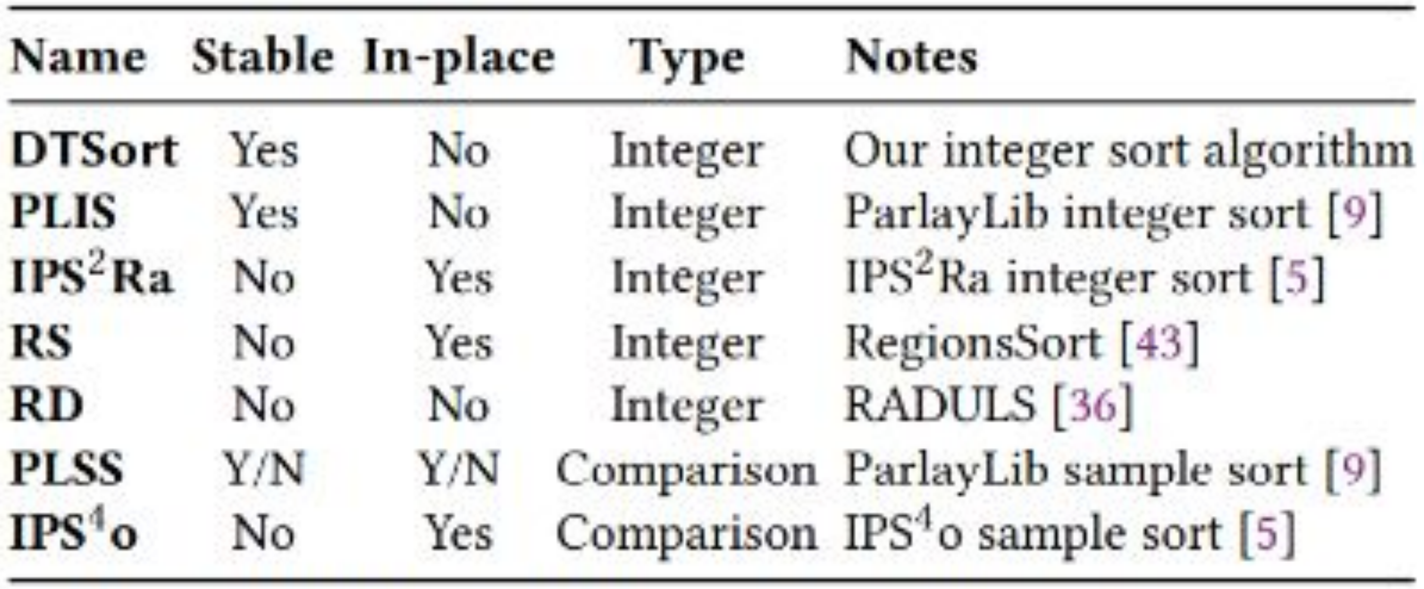
Optimization: Minimizing Data Movement

When executing the counting sort, it is not necessary to write back the bucket to the original array A, but instead use a temporary array T.

Execution is performed on T until another distribution step or merging step is reached in which case the results are transferred back to A.

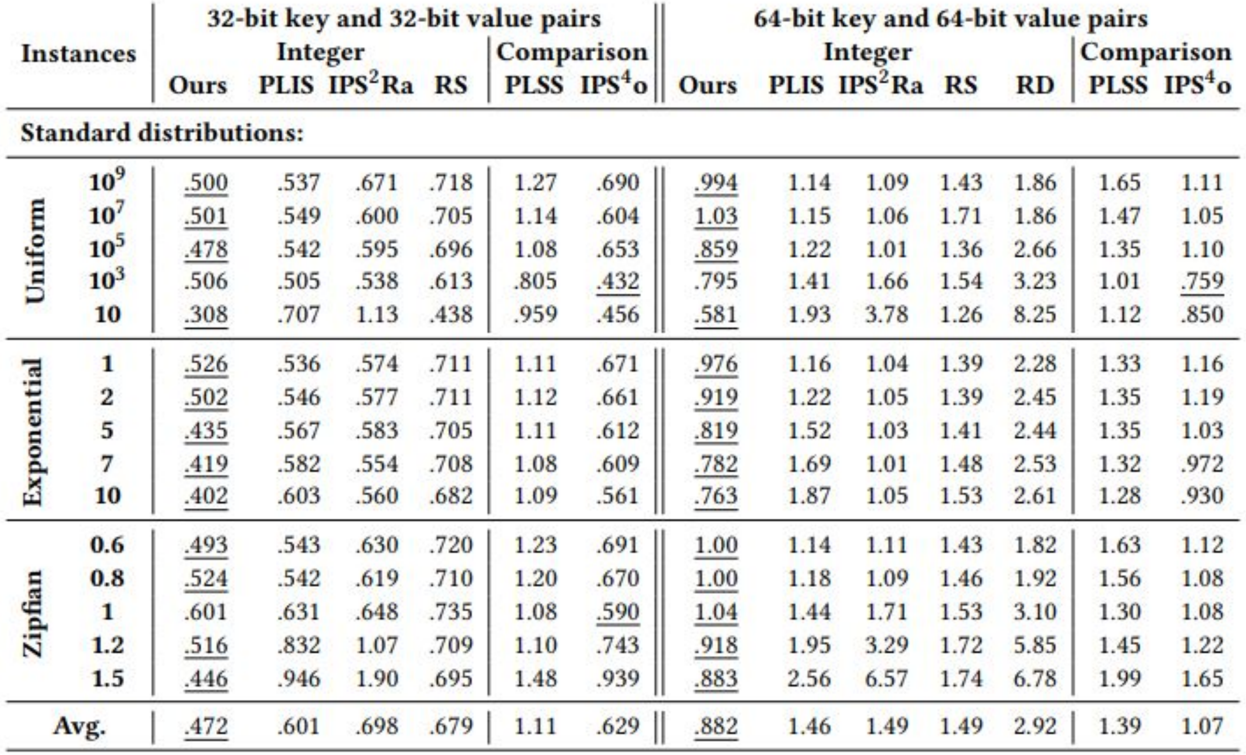
Data is moved from T to A at the end if necessary.

Experimental Comparisons



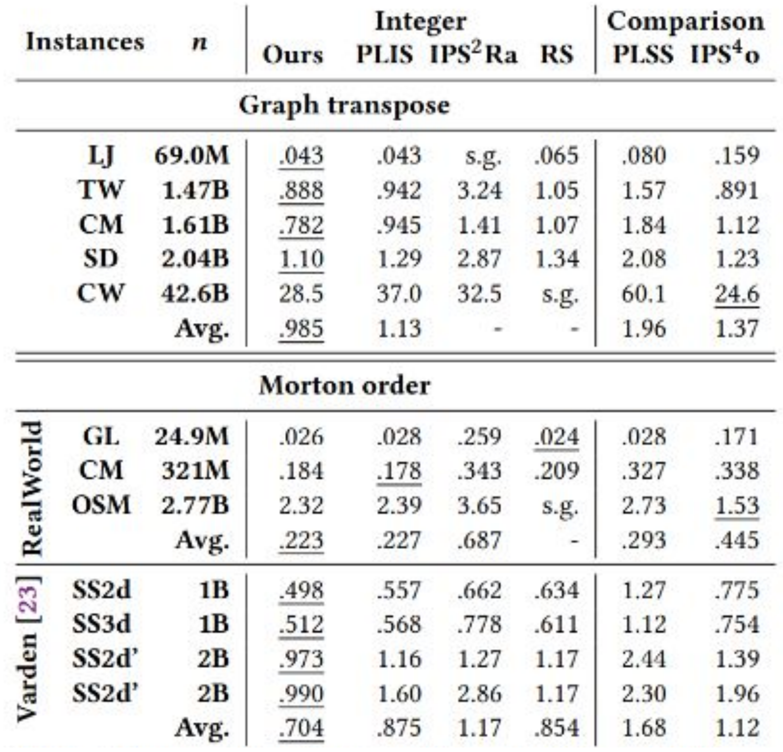
Name	Stable	In-place	Type	Notes
DTSort	Yes	No	Integer	Our integer sort algorithm
PLIS	Yes	No	Integer	ParlayLib integer sort [9]
$\mathbf{IPS^2Ra}$	No	Yes	Integer	$\mathrm{IPS^2Ra}$ integer sort [5]
RS	No	Yes	Integer	RegionsSort [43]
RD	No	No	Integer	RADULS [36]
PLSS	Y/N	Y/N	Comparison	ParlayLib sample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

Integer Sorting Experimental Results



Instances		32-bit key and 32-bit value pairs						64-bit key and 64-bit value pairs						
		Integer				Comparison		Integer					Comparison	
		Ours	PLIS	IPS^{2}Ra	RS	PLSS	IPS^{4}o	Ours	PLIS	IPS^{2}Ra	RS	RD	PLSS	IPS^{4}o
Standard distributions:														
Uniform	**10^9**	.500	.537	.671	.718	1.27	.690	.994	1.14	1.09	1.43	1.86	1.65	1.11
	10^7	.501	.549	.600	.705	1.14	.604	1.03	1.15	1.06	1.71	1.86	1.47	1.05
	10^5	.478	.542	.595	.696	1.08	.653	.859	1.22	1.01	1.36	2.66	1.35	1.10
	10^3	.506	.505	.538	.613	.805	.432	.795	1.41	1.66	1.54	3.23	1.01	.759
	10	.308	.707	1.13	.438	.959	.456	.581	1.93	3.78	1.26	8.25	1.12	.850
Exponential	**1**	.526	.536	.574	.711	1.11	.671	.976	1.16	1.04	1.39	2.28	1.33	1.16
	2	.502	.546	.577	.711	1.12	.661	.919	1.22	1.05	1.39	2.45	1.35	1.19
	5	.435	.567	.583	.705	1.11	.612	.819	1.52	1.03	1.41	2.44	1.35	1.03
	7	.419	.582	.554	.708	1.08	.609	.782	1.69	1.01	1.48	2.53	1.32	.972
	10	.402	.603	.560	.682	1.09	.561	.763	1.87	1.05	1.53	2.61	1.28	.930
Zipfian	**0.6**	.493	.543	.630	.720	1.23	.691	1.00	1.14	1.11	1.43	1.82	1.63	1.12
	0.8	.524	.542	.619	.710	1.20	.670	1.00	1.18	1.09	1.46	1.92	1.56	1.08
	1	.601	.631	.648	.735	1.08	.590	1.04	1.44	1.71	1.53	3.10	1.30	1.08
	1.2	.516	.832	1.07	.709	1.10	.743	.918	1.95	3.29	1.72	5.85	1.45	1.22
	1.5	.446	.946	1.90	.695	1.48	.939	.883	2.56	6.57	1.74	6.78	1.99	1.65
Avg.		.472	.601	.698	.679	1.11	.629	.882	1.46	1.49	1.49	2.92	1.39	1.07

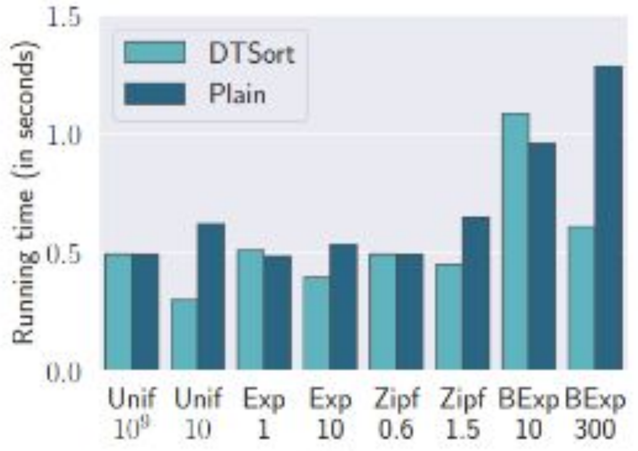
Application Experimental Results



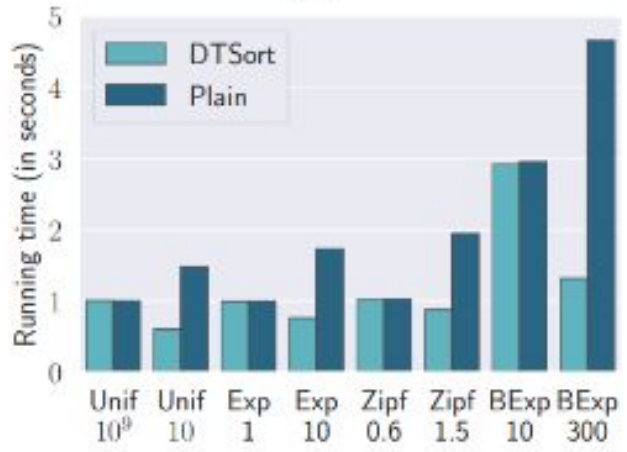
	Instances	n	Integer: Ours	Integer: PLIS	Integer: IPS^{2}Ra	Integer: RS	Comparison: PLSS	Comparison: IPS^{4}o
Graph transpose								
	LJ	69.0M	.043	.043	s.g.	.065	.080	.159
	TW	1.47B	.888	.942	3.24	1.05	1.57	.891
	CM	1.61B	.782	.945	1.41	1.07	1.84	1.12
	SD	2.04B	1.10	1.29	2.87	1.34	2.08	1.23
	CW	42.6B	28.5	37.0	32.5	s.g.	60.1	24.6
		Avg.	.985	1.13	-	-	1.96	1.37
Morton order								
RealWorld	GL	24.9M	.026	.028	.259	.024	.028	.171
	CM	321M	.184	.178	.343	.209	.327	.338
	OSM	2.77B	2.32	2.39	3.65	s.g.	2.73	1.53
		Avg.	.223	.227	.687	-	.293	.445
Varden [23]	SS2d	1B	.498	.557	.662	.634	1.27	.775
	SS3d	1B	.512	.568	.778	.611	1.12	.754
	SS2d'	2B	.973	1.16	1.27	1.17	2.44	1.39
	SS2d'	2B	.990	1.60	2.86	1.17	2.30	1.96
		Avg.	.704	.875	1.17	.854	1.68	1.12

Impact of Duplicates

Generally, as the number of duplicates, we see that DTSort performs better in comparison with other algorithms.



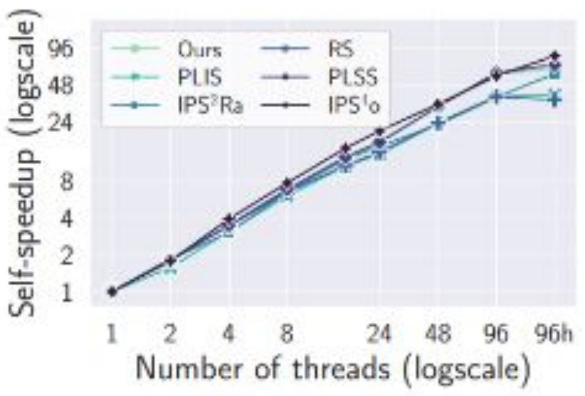
(a)



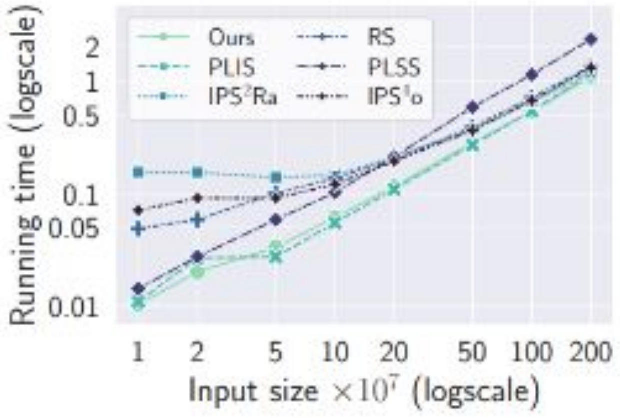
(b)

Speedup and Runtime Comparison

In general, DTSort is competitive when it comes to speedup while performing better in running time.



(e)



(f)

Final Thoughts

Overall, a notable improvement over current integer sort algorithms and is especially useful in cases where duplicates are expected.

Possible areas to improve:

> Potentially explore changing the base cases a bit? Might not be necessarily the best choice and there could be some tuning involved in determining it.
>
> Dovetail merging flips seem quite complicated for just moving data, maybe there’s a simpler alternative?