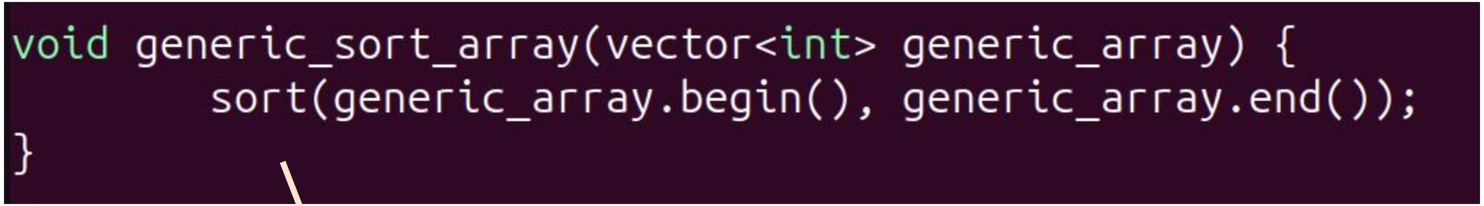
Engineering In-place (Shared-memory) Sorting Algorithms

Michael Axtmann, Sascha Witt, Daniel Ferizovic, and Peter Sanders
Karlsruhe Institute of Technology, Germany

Published Jan 2022

Motivation



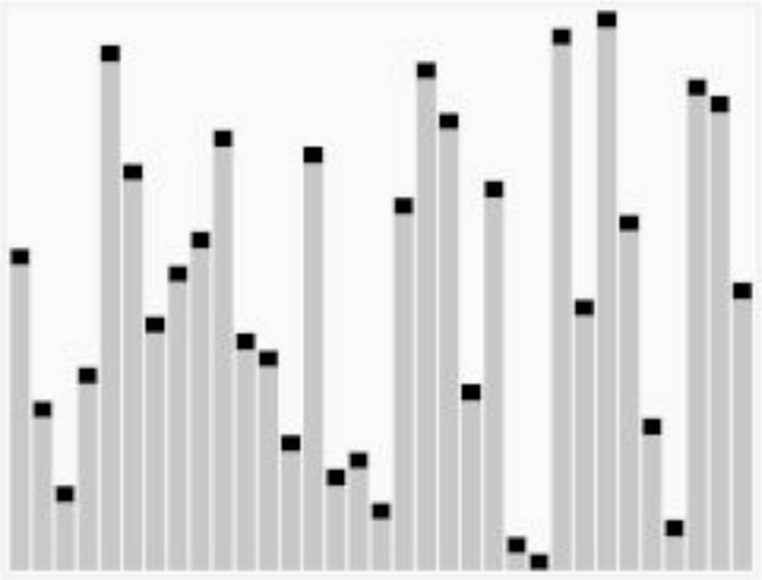
```
void generic_sort_array(vector<int> generic_array) {
        sort(generic_array.begin(), generic_array.end());
}
```

Quicksort ⇒ existed for 50 years! Can we make it faster?

Outline

- Quicksort
- Samplesort
- Super Scalar Samplesort
- In-Place Super Scalar Samplesort
- Algorithms Implemented
- Theoretical Results
- Empirical Results
- Strengths, Weaknesses, Directions for Future Work, Discussion Questions

Quicksort



Source: https://en.wikipedia.org/wiki/Quicksort

```
quicksort:
1.   pick a pivot p (somehow)
2.   swap elements such that all elements less than p are on the left and greater than p are on the right
3.   quicksort(below pivot)
4.   quicksort(above pivot)
```

Quicksort

```
quicksort:
1.   pick a pivot p (somehow)
2.   swap elements such that all elements less than p are on the left and greater than p are on the right
3.   quicksort(below pivot)
4.   quicksort(above pivot)
```

Can we do better?

Quicksort Humor

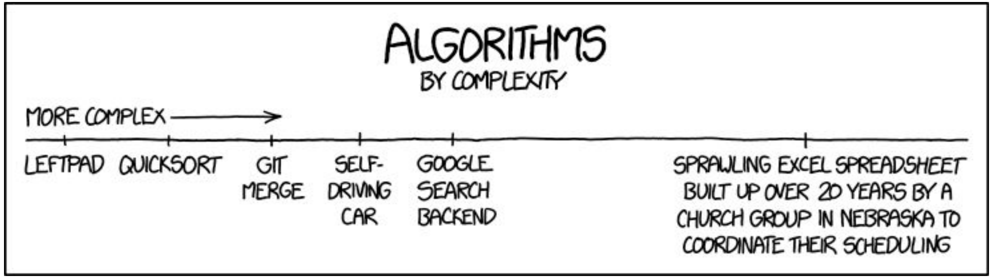
INEFFECTIVE SORTS

```
DEFINE HALFHEARTEDMERGESORT(LIST):
    IF LENGTH(LIST) < 2:
        RETURN LIST
    PIVOT = INT(LENGTH(LIST) / 2)
    A = HALFHEARTEDMERGESORT(LIST[:PIVOT])
    B = HALFHEARTEDMERGESORT(LIST[PIVOT:])
    // UMMMMM
    RETURN [A, B] // HERE. SORRY.
```

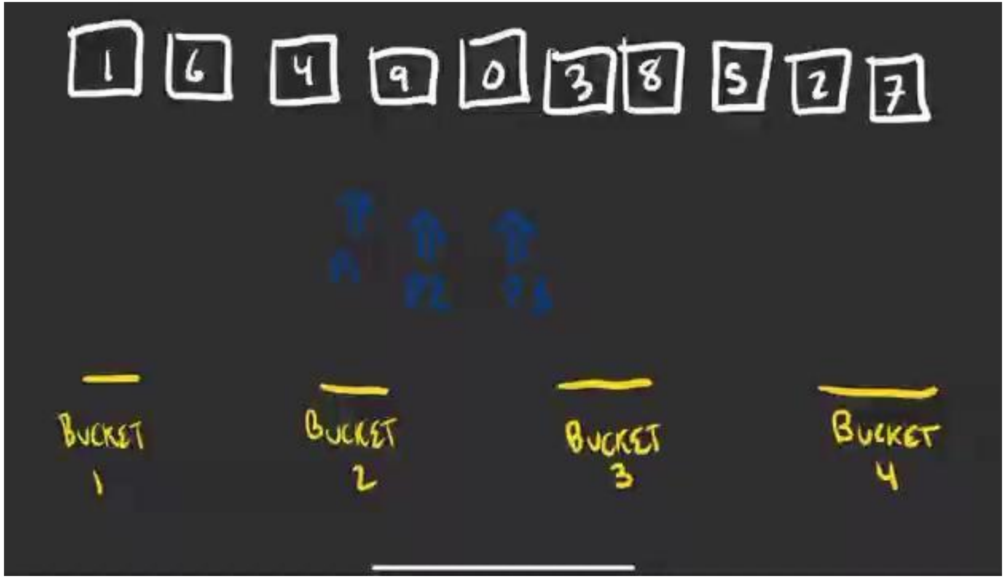
```
DEFINE FASTBOGOSORT(LIST):
    // AN OPTIMIZED BOGOSORT
    // RUNS IN O(N LOG N)
    FOR N FROM 1 TO LOG(LENGTH(LIST)):
        SHUFFLE(LIST):
        IF ISSORTED(LIST):
            RETURN LIST
    RETURN "KERNEL PAGE FAULT (ERROR CODE: 2)"
```

```
DEFINE JOBINTERVIEWQUICKSORT(LIST):
    OK SO YOU CHOOSE A PIVOT
    THEN DIVIDE THE LIST IN HALF
    FOR EACH HALF:
        CHECK TO SEE IF IT'S SORTED
            NO, WAIT, IT DOESN'T MATTER
        COMPARE EACH ELEMENT TO THE PIVOT
            THE BIGGER ONES GO IN A NEW LIST
            THE EQUAL ONES GO INTO, UH
            THE SECOND LIST FROM BEFORE
        HANG ON, LET ME NAME THE LISTS
            THIS IS LIST A
            THE NEW ONE IS LIST B
        PUT THE BIG ONES INTO LIST B
        NOW TAKE THE SECOND LIST
            CALL IT LIST, UH, A2
        WHICH ONE WAS THE PIVOT IN?
        SCRATCH ALL THAT
        IT JUST RECURSIVELY CALLS ITSELF
        UNTIL BOTH LISTS ARE EMPTY
            RIGHT?
        NOT EMPTY, BUT YOU KNOW WHAT I MEAN
    AM I ALLOWED TO USE THE STANDARD LIBRARIES?
```

```
DEFINE PANICSORT(LIST):
    IF ISSORTED(LIST):
        RETURN LIST
    FOR N FROM 1 TO 10000:
        PIVOT = RANDOM(0, LENGTH(LIST))
        LIST = LIST[PIVOT:] + LIST[:PIVOT]
        IF ISSORTED(LIST):
            RETURN LIST
    IF ISSORTED(LIST):
        RETURN LIST:
    IF ISSORTED(LIST): //THIS CAN'T BE HAPPENING
        RETURN LIST
    IF ISSORTED(LIST): // COME ON COME ON
        RETURN LIST
    // OH JEEZ
    // I'M GONNA BE IN SO MUCH TROUBLE
    LIST = [ ]
    SYSTEM("SHUTDOWN -H +5")
    SYSTEM("RM -RF ./")
    SYSTEM("RM -RF ~/*")
    SYSTEM("RM -RF /")
    SYSTEM("RD /S /Q C:\*") //PORTABILITY
    RETURN [1, 2, 3, 4, 5]
```



Samplesort *(Quicksort with Multiple Pivots)*



```
samplesort:
1.  pick p-1 pivots (somehow) and sort the pivots
2.  put each element in the correct bucket
3.  samplesort each bucket
```

Super Scalar Samplesort (S^3o)

(Samplesort with a Branchless Decision Tree)

```
s3o:
1.  randomly sample $\alpha k-1$ values and sort
    a.  pick k-1 of the values to be splitters equidistantly from the sorted sample
    b.  create a branchless decision tree with the splitters
2.  put each element in the correct bucket
3.  s3o each bucket
```

Disclaimer: Use this for the general idea; some of the smaller details may not be 1:1 to the implemented algorithm.

Super Scalar Samplesort (S^{3}o)

(Samplesort with a Branchless Decision Tree)

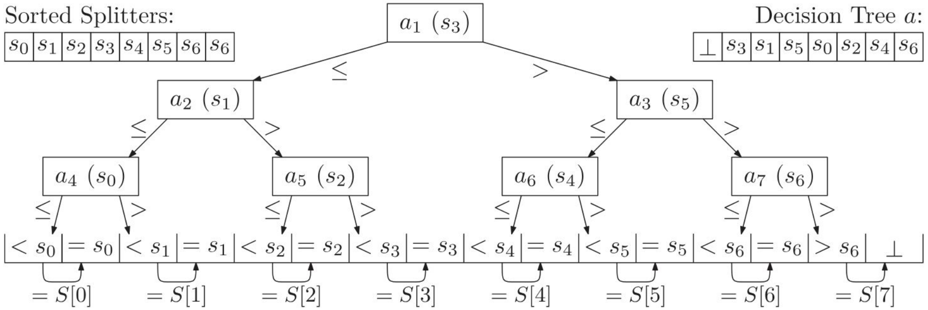


Fig. 1. Branchless decision tree with 7 splitters and 15 buckets, including 7 equality buckets. The first entry of the decision tree array stores a dummy to allow tree navigation. The last splitter in the sorted splitter array is duplicated to avoid a case distinction.

s3o:

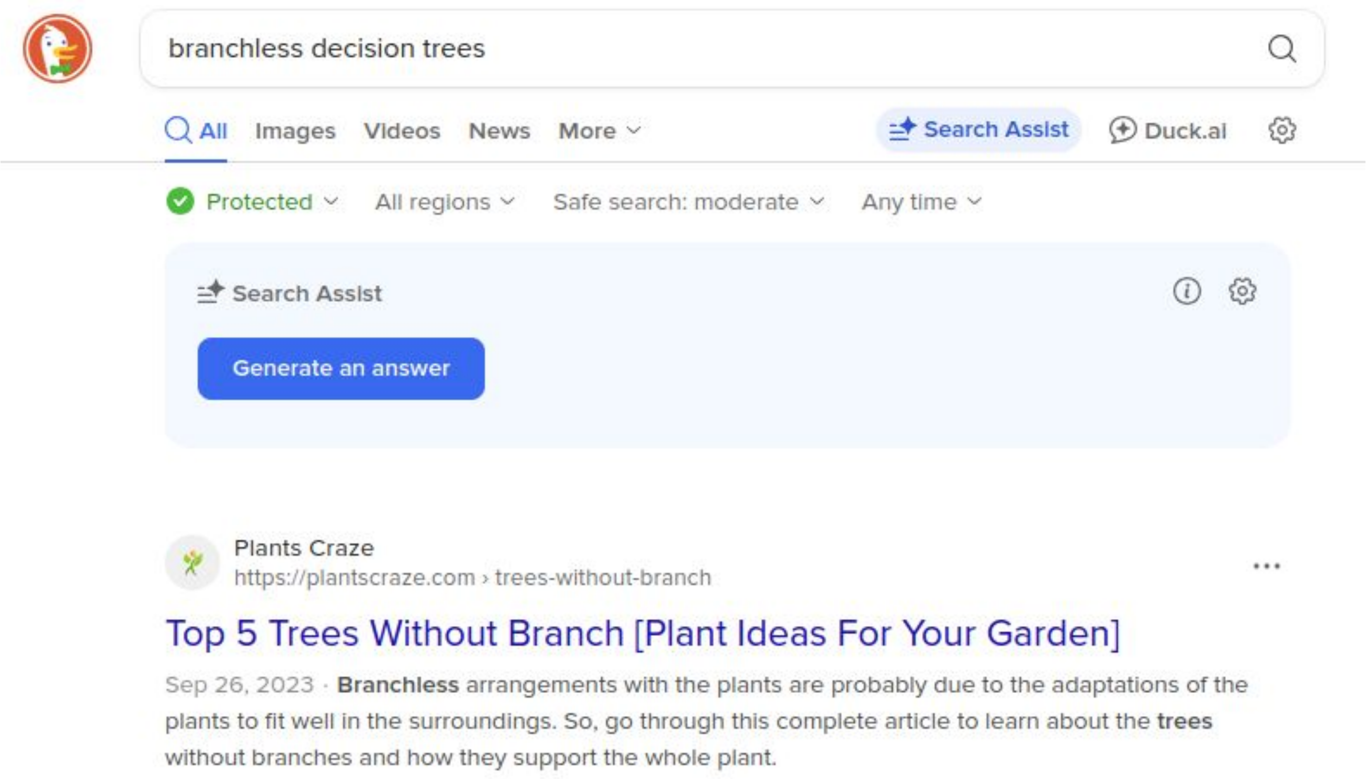
1. randomly sample αk-1 values and sort
    a. pick k-1 of the values to be splitters equidistantly from the sorted sample
    b. create a **branchless decision tree** with the splitters
2. put each element in the correct bucket
3. s3o each bucket

Eliminates branch mispredictions!

Notice: Not in-Place!

How can we modify this to be in-place?

I love gardening?



branchless decision trees

Plants Craze
https://plantscraze.com › trees-without-branch

Top 5 Trees Without Branch [Plant Ideas For Your Garden]

Sep 26, 2023 · **Branchless** arrangements with the plants are probably due to the adaptations of the plants to fit well in the surroundings. So, go through this complete article to learn about the **trees** without branches and how they support the whole plant.

In-Place Super Scalar Samplesort (IPS^{4}o)

```
ips4o:
1.   [similar selection step with αk-1 values and k-1 splitters]
2.   each thread gets part of the array to sort into k bins of size b
     a.   if b is used, replace it in the original array
3.   permutate the b-sized blocks to get bins in proper order
4.   cleanup excess values
5.   ips4o on the bins created
```

Implemented Algorithms

IPS^{4}o → in-place parallel superscalar samplesort

I1S^{4}o → in-place sequential superscalar samplesort

IPS^{2}Ra → in-place parallel superscalar radix sort (replace branchless decision tree with simple radix extractor function that accepts uint keys)

I1S^{2}Ra → in-place sequential superscalar radix sort

Theoretical Results – Summary

Memory Theorems

IPS^{4}o can be implemented with $O(kb)$ additional memory per thread

With a local stack, IPS^{4}o can be implemented with $O(k(b + t\log_k(n/n_0))$ additional memory per thread

I/O Complexity and Work

IPS^{4}o has an I/O complexity of $O(n/(tB)\log_k n/M)$ memory block transfers with ability of at least $1 - M/n$.

When using quicksort to sort the base cases and the samples, IPS^{4}o has local work of $O(n/t\log n)$ with probability at least $1–4/n$

Empirical Results – Setup

21 SORTING ALGORITHMS

Used implementations of:

```
[Parallel OR Sequential]
x
[In-Place OR Non-In-Place]
x
[Radix Sort OR
Comparison-Based Sorting]
```

x

10 DATA DISTRIBUTIONS *(Uniform, Exponential, AlmostSorted, RootDup, TwoDup, EightDup, Zipf, Sorted, ReverseSorter, Zero)*

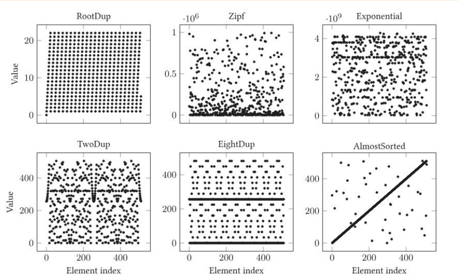


Fig. 10. Examples of nontrivial input distributions for 512 uint32 values.

x

4 MACHINES
x
6 DATA TYPES
(float, uint64, uint32, Pair, Quartet, 100B)

Empirical Results – Metrics

AVERAGE ALGORITHM SLOWDOWN

$$f_{\mathcal{A},I}(A) = \begin{cases} r(A,I)/\min(\{r(A',I) \mid A' \in \mathcal{A}\}) & I \in S_A(\mathcal{I}), \text{ i.e., } A \text{ successfully sorts } I \\ \infty & \text{otherwise.} \end{cases}$$

r(A, I) = runtime of algorithm A on input I

$$s_{\mathcal{A},\mathcal{I}}(A) = \sqrt[|S_A(\mathcal{I})|]{\prod_{I \in S_A(\mathcal{I})} f_{\mathcal{A},I}(A)}$$

"The average slowdown of algorithm A for input I is the geometric mean of the ratio of A's runtime on I to the best algorithm runtime on I (which is infinity if the algorithm fails)."

AVERAGE TYPE SLOWDOWN

$$f_{\mathcal{T},A,I}(T) = \begin{cases} r(A,I,T)/\min(\{r(A,I,T') \mid T' \in \mathcal{T}\}) & I \in S_A(\mathcal{I}), \text{ i.e., } A \text{ successfully sorts } I \\ \infty & \text{otherwise.} \end{cases}$$

r(A, I, T) = runtime of algorithm A on input I with type T

$$s_{A,\mathcal{T},\mathcal{I}}(T) = \sqrt[|S_A(\mathcal{I})|]{\prod_{I \in S_A(\mathcal{I})} f_{\mathcal{T},A,I}(T)}$$

"The average slowdown of type T for algorithm A and input I is the geometric mean of the ratio of A's runtime on I with type T to best type T' runtime with algorithm A and input I (which is infinity if the algorithm fails)."

(Also uses pairwise profilers for directly comparing two algorithms!)

Note: average of all runs but the first!

Empirical Results – Sequential

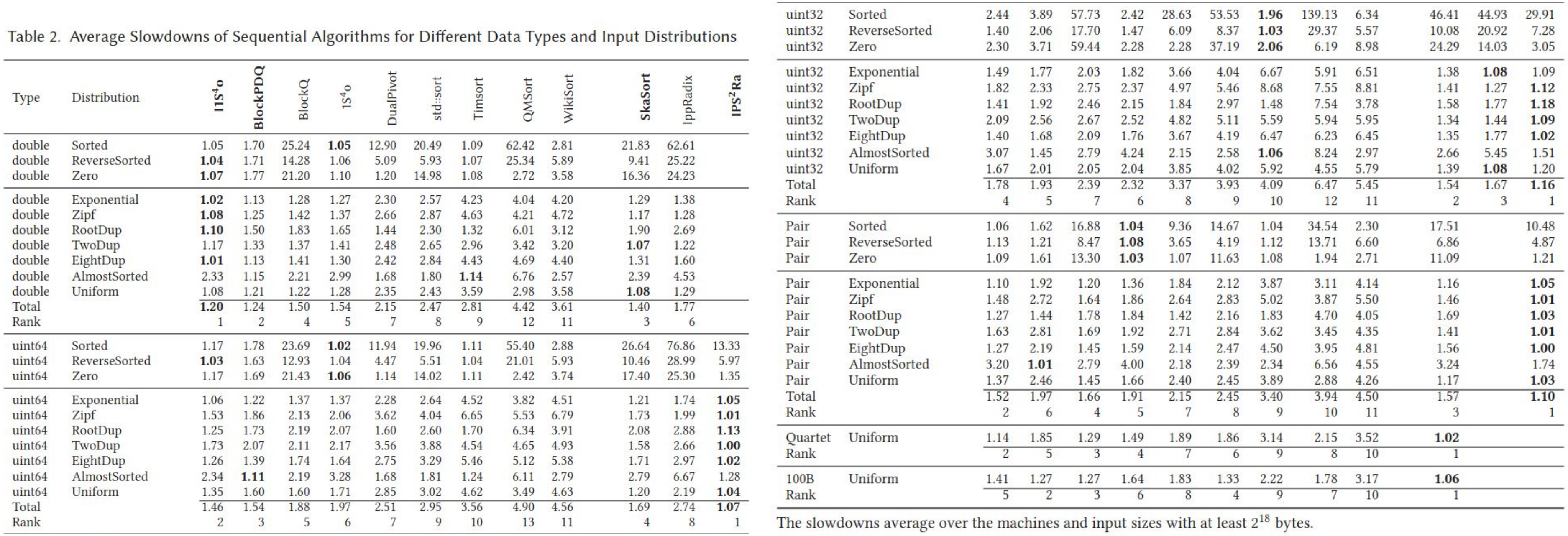


Table 2. Average Slowdowns of Sequential Algorithms for Different Data Types and Input Distributions

Type	Distribution	**I1S^{4}o**	**BlockPDQ**	BlockQ	1S^{4}o	DualPivot	std::sort	Timsort	QMSort	WikiSort	**SkaSort**	IppRadix	**IPS^{2}Ra**
double	Sorted	1.05	1.70	25.24	**1.05**	12.90	20.49	1.09	62.42	2.81	21.83	62.61	
double	ReverseSorted	**1.04**	1.71	14.28	1.06	5.09	5.93	1.07	25.34	5.89	9.41	25.22	
double	Zero	**1.07**	1.77	21.20	1.10	1.20	14.98	1.08	2.72	3.58	16.36	24.23	
double	Exponential	**1.02**	1.13	1.28	1.27	2.30	2.57	4.23	4.04	4.20	1.29	1.38	
double	Zipf	**1.08**	1.25	1.42	1.37	2.66	2.87	4.63	4.21	4.72	1.17	1.28	
double	RootDup	**1.10**	1.50	1.83	1.65	1.44	2.30	1.32	6.01	3.12	1.90	2.69	
double	TwoDup	1.17	1.33	1.37	1.41	2.48	2.65	2.96	3.42	3.20	**1.07**	1.22	
double	EightDup	**1.01**	1.13	1.41	1.30	2.42	2.84	4.43	4.69	4.40	1.31	1.60	
double	AlmostSorted	2.33	1.15	2.21	2.99	1.68	1.80	**1.14**	6.76	2.57	2.39	4.53	
double	Uniform	1.08	1.21	1.22	1.28	2.35	2.43	3.59	2.98	3.58	**1.08**	1.29	
Total		**1.20**	1.24	1.50	1.54	2.15	2.47	2.81	4.42	3.61	1.40	1.77	
Rank		1	2	4	5	7	8	9	12	11	3	6	
uint64	Sorted	1.17	1.78	23.69	**1.02**	11.94	19.96	1.11	55.40	2.88	26.64	76.86	13.33
uint64	ReverseSorted	**1.03**	1.63	12.93	1.04	4.47	5.51	1.04	21.01	5.93	10.46	28.99	5.97
uint64	Zero	1.17	1.69	21.43	**1.06**	1.14	14.02	1.11	2.42	3.74	17.40	25.30	1.35
uint64	Exponential	1.06	1.22	1.37	1.37	2.28	2.64	4.52	3.82	4.51	1.21	1.74	**1.05**
uint64	Zipf	1.53	1.86	2.13	2.06	3.62	4.04	6.65	5.53	6.79	1.73	1.99	**1.01**
uint64	RootDup	1.25	1.73	2.19	2.07	1.60	2.60	1.70	6.34	3.91	2.08	2.88	**1.13**
uint64	TwoDup	1.73	2.07	2.11	2.17	3.56	3.88	4.54	4.65	4.93	1.58	2.66	**1.00**
uint64	EightDup	1.26	1.39	1.74	1.64	2.75	3.29	5.46	5.12	5.38	1.71	2.97	**1.02**
uint64	AlmostSorted	2.34	**1.11**	2.19	3.28	1.68	1.81	1.24	6.11	2.79	2.79	6.67	1.28
uint64	Uniform	1.35	1.60	1.60	1.71	2.85	3.02	4.62	3.49	4.63	1.20	2.19	**1.04**
Total		1.46	1.54	1.88	1.97	2.51	2.95	3.56	4.90	4.56	1.69	2.74	**1.07**
Rank		2	3	5	6	7	9	10	13	11	4	8	1
uint32	Sorted	2.44	3.89	57.73	2.42	28.63	53.53	**1.96**	139.13	6.34	46.41	44.93	29.91
uint32	ReverseSorted	1.40	2.06	17.70	1.47	6.09	8.37	**1.03**	29.37	5.57	10.08	20.92	7.28
uint32	Zero	2.30	3.71	59.44	2.28	2.28	37.19	**2.06**	6.19	8.98	24.29	14.03	3.05
uint32	Exponential	1.49	1.77	2.03	1.82	3.66	4.04	6.67	5.91	6.51	1.38	**1.08**	1.09
uint32	Zipf	1.82	2.33	2.75	2.37	4.97	5.46	8.68	7.55	8.81	1.41	1.27	**1.12**
uint32	RootDup	1.41	1.92	2.46	2.15	1.84	2.97	1.48	7.54	3.78	1.58	1.77	**1.18**
uint32	TwoDup	2.09	2.56	2.67	2.52	4.82	5.11	5.59	5.94	5.95	1.34	1.44	**1.09**
uint32	EightDup	1.40	1.68	2.09	1.76	3.67	4.19	6.47	6.23	6.45	1.35	1.77	**1.02**
uint32	AlmostSorted	3.07	1.45	2.79	4.24	2.15	2.58	**1.06**	8.24	2.97	2.66	5.45	1.51
uint32	Uniform	1.67	2.01	2.05	2.04	3.85	4.02	5.92	4.55	5.79	1.39	**1.08**	1.20
Total		1.78	1.93	2.39	2.32	3.37	3.93	4.09	6.47	5.45	1.54	1.67	**1.16**
Rank		4	5	7	6	8	9	10	12	11	2	3	1
Pair	Sorted	1.06	1.62	16.88	**1.04**	9.36	14.67	1.04	34.54	2.30	17.51		10.48
Pair	ReverseSorted	1.13	1.21	8.47	**1.08**	3.65	4.19	1.12	13.71	6.60	6.86		4.87
Pair	Zero	1.09	1.61	13.30	**1.03**	1.07	11.63	1.08	1.94	2.71	11.09		1.21
Pair	Exponential	1.10	1.92	1.20	1.36	1.84	2.12	3.87	3.11	4.14	1.16		**1.05**
Pair	Zipf	1.48	2.72	1.64	1.86	2.64	2.83	5.02	3.87	5.50	1.46		**1.01**
Pair	RootDup	1.27	1.44	1.78	1.84	1.42	2.16	1.83	4.70	4.05	1.69		**1.03**
Pair	TwoDup	1.63	2.81	1.69	1.92	2.71	2.84	3.62	3.45	4.35	1.41		**1.01**
Pair	EightDup	1.27	2.19	1.45	1.59	2.14	2.47	4.50	3.95	4.81	1.56		**1.00**
Pair	AlmostSorted	3.20	**1.01**	2.79	4.00	2.18	2.39	2.34	6.56	4.55	3.24		1.74
Pair	Uniform	1.37	2.46	1.45	1.66	2.40	2.45	3.89	2.88	4.26	1.17		**1.03**
Total		1.52	1.97	1.66	1.91	2.15	2.45	3.40	3.94	4.50	1.57		**1.10**
Rank		2	6	4	5	7	8	9	10	11	3		1
Quartet	Uniform	1.14	1.85	1.29	1.49	1.89	1.86	3.14	2.15	3.52	**1.02**		
Rank		2	5	3	4	7	6	9	8	10	1		
100B	Uniform	1.41	1.27	1.27	1.64	1.83	1.33	2.22	1.78	3.17	**1.06**		
Rank		5	2	3	6	8	4	9	7	10	1		

The slowdowns average over the machines and input sizes with at least 2^{18} bytes.

Empirical Results – Sequential

- I1S^{2}Ra is significantly faster than the fastest radix sort competitor SkaSort
- In some cases, IppRadix is faster
- I1S^{2}Ra outperforms for Uniform inputs and Skewed inputs significantly
- In AlmostSorted and (Reverse) Sorted inputs, BlockPDQ and Timsort are faster

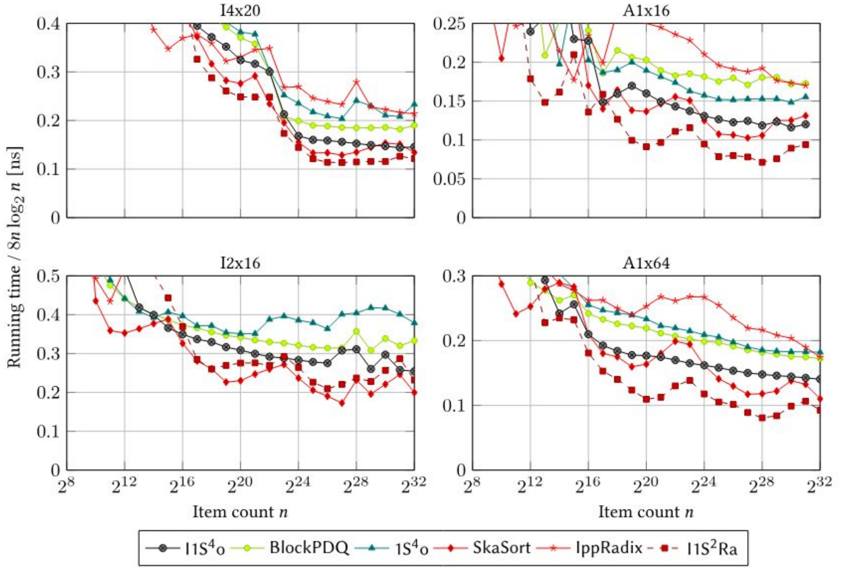


Fig. 11. Running times of sequential algorithms of uint64 values with input distribution Uniform executed on different machines. The results of DualPivot, std::sort, Timsort, QMSort, and WikiSort cannot be seen as their running times exceed the plot.

Empirical Results – Sequential

I1S^{2}Ra

- **IppRadix** is the only non-comparison-based algorithm to outperform I1S^{2}Ra at least once (for Exponential + uin32) but I1S^2 much better in other scenarios.
- Comparison-Based Algorithms:
    - Much better in Uniform and Skewed inputs
    - Almost Sorted: Slower than **BlockPDQ**
        - BlockPDQ heuristically detects and skips presorted inputs
        - But beats it in other data types
    - **Reverse Sorted**: claims can be fixed with an initial scan
- SkaSort is faster on one machine for the uniform input. It has more cache misses and branch mispredictions. Can maybe beat SkaSort if using vector instructions

I1S^{4}o

- Outperformed by others most with Reverse Sorted and Almost Sorted (can be fixed with tricks)

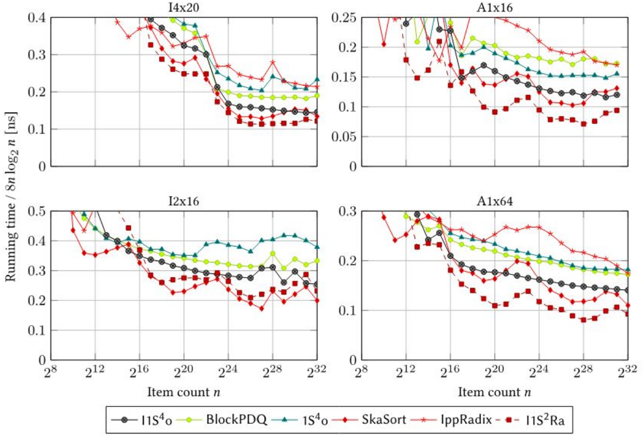


Fig. 11. Running times of sequential algorithms of uint64 values with input distribution Uniform executed on different machines. The results of DualPivot, std::sort, Timsort, QMSort, and WikiSort cannot be seen as their running times exceed the plot.

Empirical Results – Parallel

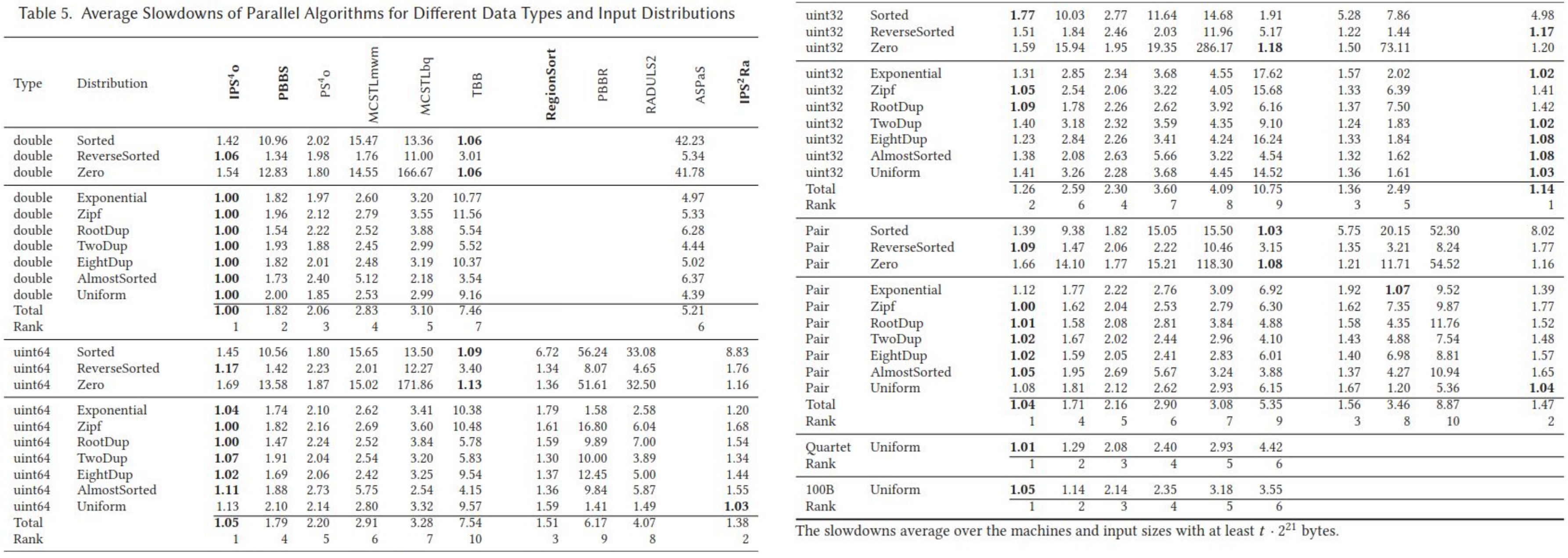


Table 5. Average Slowdowns of Parallel Algorithms for Different Data Types and Input Distributions

Type	Distribution	**IPS^{4}o**	**PBBS**	PS^{4}o	MCSTLmwm	MCSTLbq	TBB	**RegionSort**	PBBR	RADULS2	ASPaS	**IPS^{2}Ra**
double	Sorted	1.42	10.96	2.02	15.47	13.36	**1.06**				42.23	
double	ReverseSorted	**1.06**	1.34	1.98	1.76	11.00	3.01				5.34	
double	Zero	1.54	12.83	1.80	14.55	166.67	**1.06**				41.78	
double	Exponential	**1.00**	1.82	1.97	2.60	3.20	10.77				4.97	
double	Zipf	**1.00**	1.96	2.12	2.79	3.55	11.56				5.33	
double	RootDup	**1.00**	1.54	2.22	2.52	3.88	5.54				6.28	
double	TwoDup	**1.00**	1.93	1.88	2.45	2.99	5.52				4.44	
double	EightDup	**1.00**	1.82	2.01	2.48	3.19	10.37				5.02	
double	AlmostSorted	**1.00**	1.73	2.40	5.12	2.18	3.54				6.37	
double	Uniform	**1.00**	2.00	1.85	2.53	2.99	9.16				4.39	
Total		**1.00**	1.82	2.06	2.83	3.10	7.46				5.21	
Rank		1	2	3	4	5	7				6	
uint64	Sorted	1.45	10.56	1.80	15.65	13.50	**1.09**	6.72	56.24	33.08		8.83
uint64	ReverseSorted	**1.17**	1.42	2.23	2.01	12.27	3.40	1.34	8.07	4.65		1.76
uint64	Zero	1.69	13.58	1.87	15.02	171.86	**1.13**	1.36	51.61	32.50		1.16
uint64	Exponential	**1.04**	1.74	2.10	2.62	3.41	10.38	1.79	1.58	2.58		1.20
uint64	Zipf	**1.00**	1.82	2.16	2.69	3.60	10.48	1.61	16.80	6.04		1.68
uint64	RootDup	**1.00**	1.47	2.24	2.52	3.84	5.78	1.59	9.89	7.00		1.54
uint64	TwoDup	**1.07**	1.91	2.04	2.54	3.20	5.83	1.30	10.00	3.89		1.34
uint64	EightDup	**1.02**	1.69	2.06	2.42	3.25	9.54	1.37	12.45	5.00		1.44
uint64	AlmostSorted	**1.11**	1.88	2.73	5.75	2.54	4.15	1.36	9.84	5.87		1.55
uint64	Uniform	1.13	2.10	2.14	2.80	3.32	9.57	1.59	1.41	1.49		**1.03**
Total		**1.05**	1.79	2.20	2.91	3.28	7.54	1.51	6.17	4.07		1.38
Rank		1	4	5	6	7	10	3	9	8		2
uint32	Sorted	**1.77**	10.03	2.77	11.64	14.68	1.91	5.28	7.86			4.98
uint32	ReverseSorted	1.51	1.84	2.46	2.03	11.96	5.17	1.22	1.44			**1.17**
uint32	Zero	1.59	15.94	1.95	19.35	286.17	**1.18**	1.50	73.11			1.20
uint32	Exponential	1.31	2.85	2.34	3.68	4.55	17.62	1.57	2.02			**1.02**
uint32	Zipf	**1.05**	2.54	2.06	3.22	4.05	15.68	1.33	6.39			1.41
uint32	RootDup	**1.09**	1.78	2.26	2.62	3.92	6.16	1.37	7.50			1.42
uint32	TwoDup	1.40	3.18	2.32	3.59	4.35	9.10	1.24	1.83			**1.02**
uint32	EightDup	1.23	2.84	2.26	3.41	4.24	16.24	1.33	1.84			**1.08**
uint32	AlmostSorted	1.38	2.08	2.63	5.66	3.22	4.54	1.32	1.62			**1.08**
uint32	Uniform	1.41	3.26	2.28	3.68	4.45	14.52	1.36	1.61			**1.03**
Total		1.26	2.59	2.30	3.60	4.09	10.75	1.36	2.49			**1.14**
Rank		2	6	4	7	8	9	3	5			1
Pair	Sorted	1.39	9.38	1.82	15.05	15.50	**1.03**	5.75	20.15	52.30		8.02
Pair	ReverseSorted	**1.09**	1.47	2.06	2.22	10.46	3.15	1.35	3.21	8.24		1.77
Pair	Zero	1.66	14.10	1.77	15.21	118.30	**1.08**	1.21	11.71	54.52		1.16
Pair	Exponential	1.12	1.77	2.22	2.76	3.09	6.92	1.92	**1.07**	9.52		1.39
Pair	Zipf	**1.00**	1.62	2.04	2.53	2.79	6.30	1.62	7.35	9.87		1.77
Pair	RootDup	**1.01**	1.58	2.08	2.81	3.84	4.88	1.58	4.35	11.76		1.52
Pair	TwoDup	**1.02**	1.67	2.02	2.44	2.96	4.10	1.43	4.88	7.54		1.48
Pair	EightDup	**1.02**	1.59	2.05	2.41	2.83	6.01	1.40	6.98	8.81		1.57
Pair	AlmostSorted	**1.05**	1.95	2.69	5.67	3.24	3.88	1.37	4.27	10.94		1.65
Pair	Uniform	1.08	1.81	2.12	2.62	2.93	6.15	1.67	1.20	5.36		**1.04**
Total		**1.04**	1.71	2.16	2.90	3.08	5.35	1.56	3.46	8.87		1.47
Rank		1	4	5	6	7	9	3	8	10		2
Quartet	Uniform	**1.01**	1.29	2.08	2.40	2.93	4.42					
Rank		1	2	3	4	5	6					
100B	Uniform	**1.05**	1.14	2.14	2.35	3.18	3.55					
Rank		1	2	3	4	5	6					

The slowdowns average over the machines and input sizes with at least $t \cdot 2^{21}$ bytes.

Empirical Results – Parallel

- Focuses on one machine (I4x20) because it had tasks with more than n/t elements regularly
- IPS^{4}o has the fastest running times and is faster than the fastest comparison-based competitor PBBS
- IPS^{4}o is significantly faster than its fastest radix sort competitor RegionSort (except for some uint32 cases)

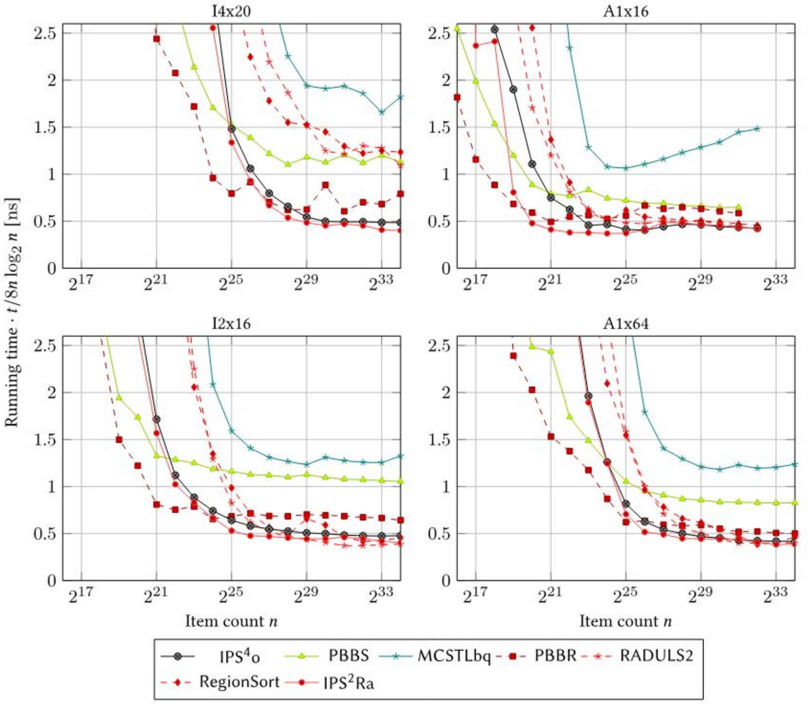


Fig. 15. Running times of parallel algorithms sorting uint64 values with input distribution Uniform executed on different machines.

Empirical Results – Parallel

IPS^{4}o

- Much faster except for some easy cases (Sorted, ReverseSorted, Zero) and uint32 data types. There's an overhead in detecting easy cases.
- **RegionSort** is notable faster for TwoDup with uin32 inputs.
- Notice: much faster than **PS^{4}o** (in-place helps!)
- **RADULS2** is good for uniform inputs but not as good if skewed.

IPS^{2}Ra

- Slightly better than **RegionSort**!
- Outperforms IPS^{4}o for uint32 instances → branchless decision tree is not a limiting factor for uint32

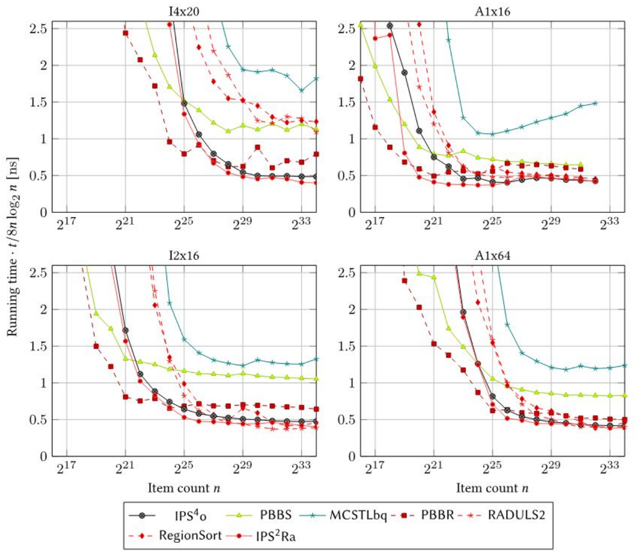


Fig. 15. Running times of parallel algorithms sorting uint64 values with input distribution Uniform executed on different machines.

Directions for Future Work

ENGINEERING

- *Use not in-place algorithms for small inputs*
- *Explore different base sorts*
- *Precompute a lookup table?*
- *Speed up with vector instructions!*

THEORY

- *More scalable variant of IPS^{4}o with span O(log^2 n)*
- *Make it sublinear?*
- *Formal verification of the algorithm*

Strengths and Weaknesses

+ Thorough comparison of IPS^{4}o with other sorting algorithms

+ Promising results on their analyses

+ Theoretical memory and complexity guarantees

- Lengthy, research-prototype codebase

- Complex algorithm, handles many edge cases

Discussion Questions

- Turns out sort() did not get replaced with IPS^{4}o (yet??). What will it take to replace sort()?
  - Would it makes sense to have a very performant sort which is thousands of lines of code?
  - Given that they implement tricks under the hood (ex: checking if there are patterns if it's in reverse order or the same value many, many times), do you think that some of the benchmark results could be misleading?

- What does it take to be the "best" sorting algorithm?