High-Performance and Flexible Parallel Algorithms for Semisort and Related Problems

Xiaojun Dong[1], Yunshu Wu[1], Zhongqi Wang[2], Laxman Dhulipala[2], Yan Gu[1], Yihan Sun[1]

Full version: https://arxiv.org/abs/2304.10078

Code: https://github.com/ucrparlay/Parallel-Semisort

[1]

[2]

Computational Model

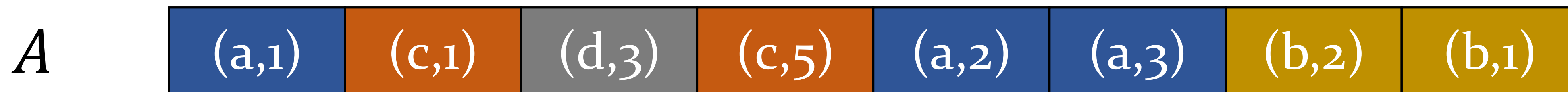
- Shared-memory multi-core setting
- Work-span model
- Work: the total number of operations
- Span (depth): the length of the longest dependency chains

Problem definitions: semisort

- Given a sequence of records A and an equality test "$=$" on keys
- Goal: all records with equal keys are adjacent

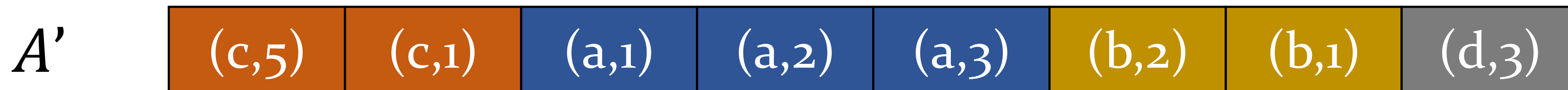
A is a sequence of (key, value) pairs

A



Reorder them into A'

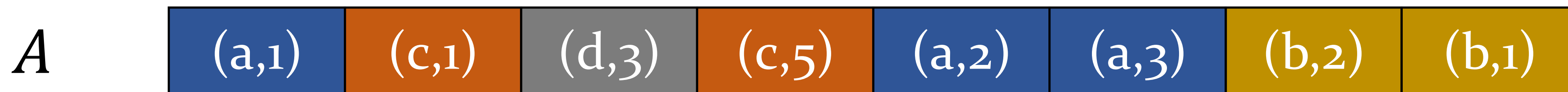
A'

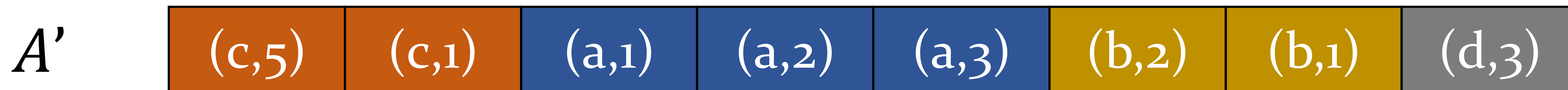


Problem definitions: semisort

- Given a sequence of records A and an equality test "$=$" on keys
- Goal: all records with equal keys are adjacent
- Distinct keys are not necessarily sorted

A is a sequence of (key, value) pairs

A


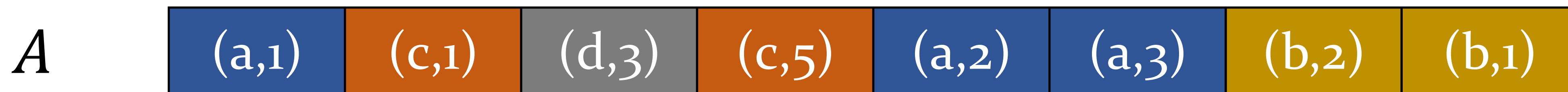
A'


Problem definitions: semisort

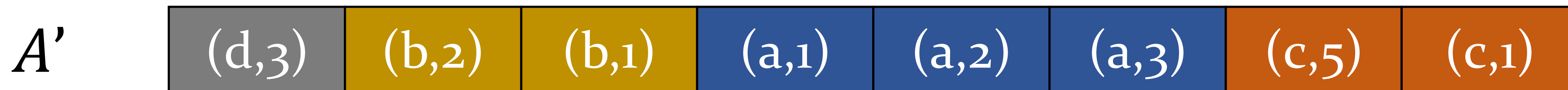
- Given a sequence of records A and an equality test "$=$" on keys
- Goal: all records with equal keys are adjacent
- Distinct keys are not necessarily sorted
- Records with equal keys do not need to be sorted by their values

A is a sequence of (key, value) pairs

A



A'



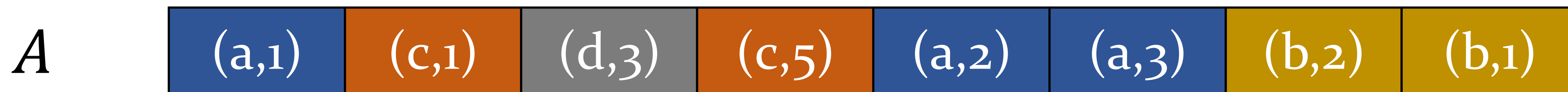
Problem definitions: semisort

- Given a sequence of records A and an equality test "$=$" on keys
- Goal: all records with equal keys are adjac
- Distinct keys are not necessarily sorted
- Records with equal keys do not need to b

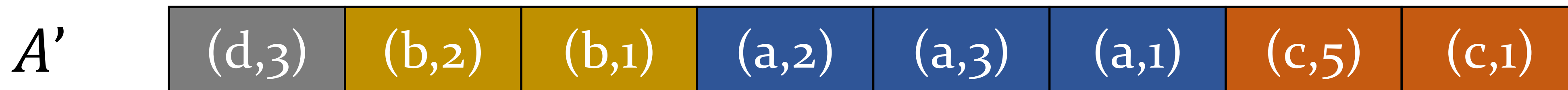
And a user-defined function to hash the keys to integers

A is a sequence of (key, value) pairs

A



A'



Problem definitions: histogram and collect-reduce

- Histogram: count the number of occurrences of each key

A is a sequence of (key, value) pairs

key	c	a	b	d
occurrences	2	3	2	1

Problem definitions: histogram and collect-reduce

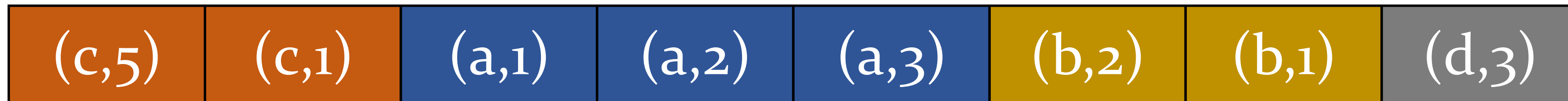
- Histogram: count the number of occurrences of each key
- Collect-reduce: computes the aggregate "sum" of each key
  - The "sum" can be any binary associative functions (e.g., plus, min, and xor)

A is a sequence of (key, value) pairs

key	c	a	b	d
aggregate "sum"	6	6	3	3

Problem definitions: histogram and collect-reduce

- Histogram: count the number of occurrences of each key
- Collect-reduce: computes the aggregate "sum" of each key
  - The "sum" can be any binary associative functions (e.g., plus, min, and xor)
  - Commutativity is also needed if semisorting is not stable



Histogram

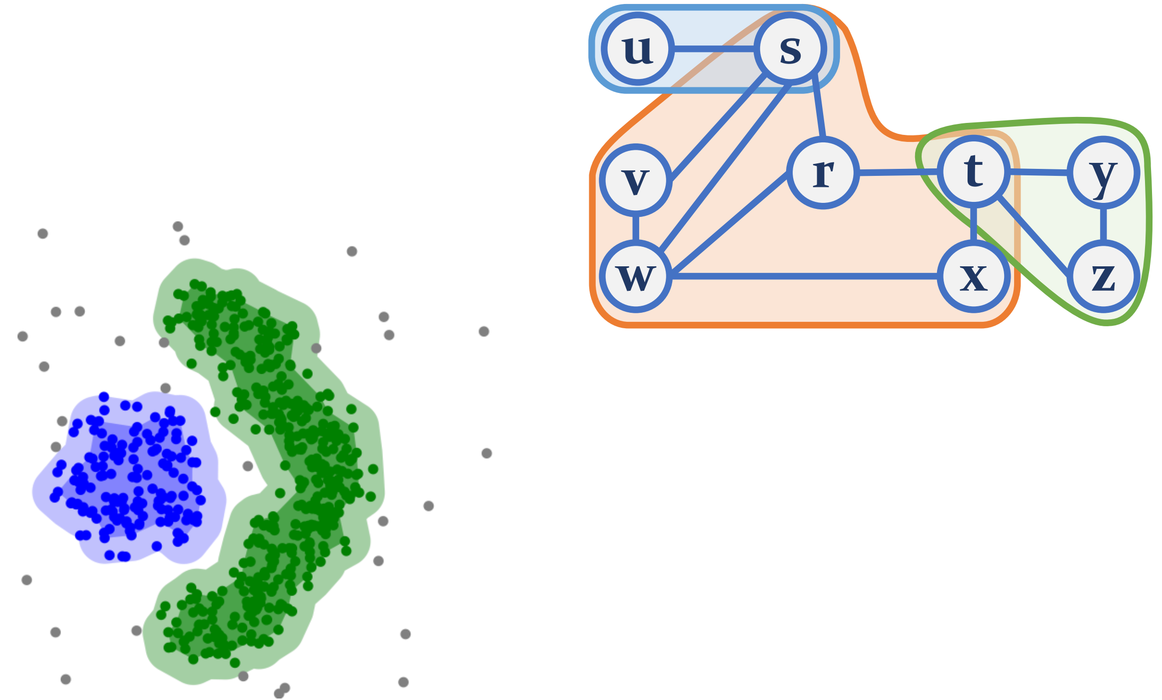
Collect-Reduce

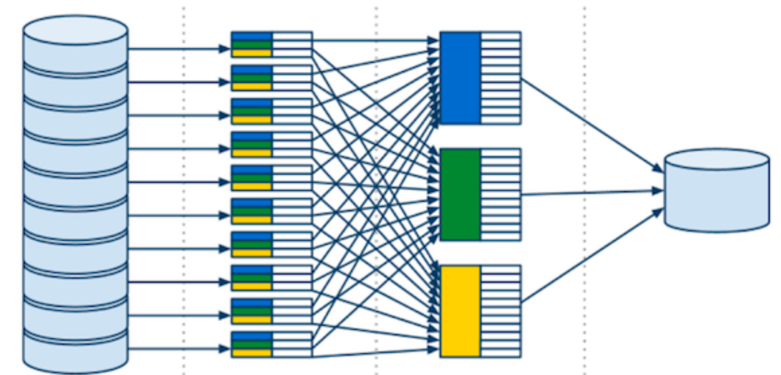
key	c	a	b	d
occurrences	2	3	2	1

key	c	a	b	d
aggregate "sum"	6	6	3	3

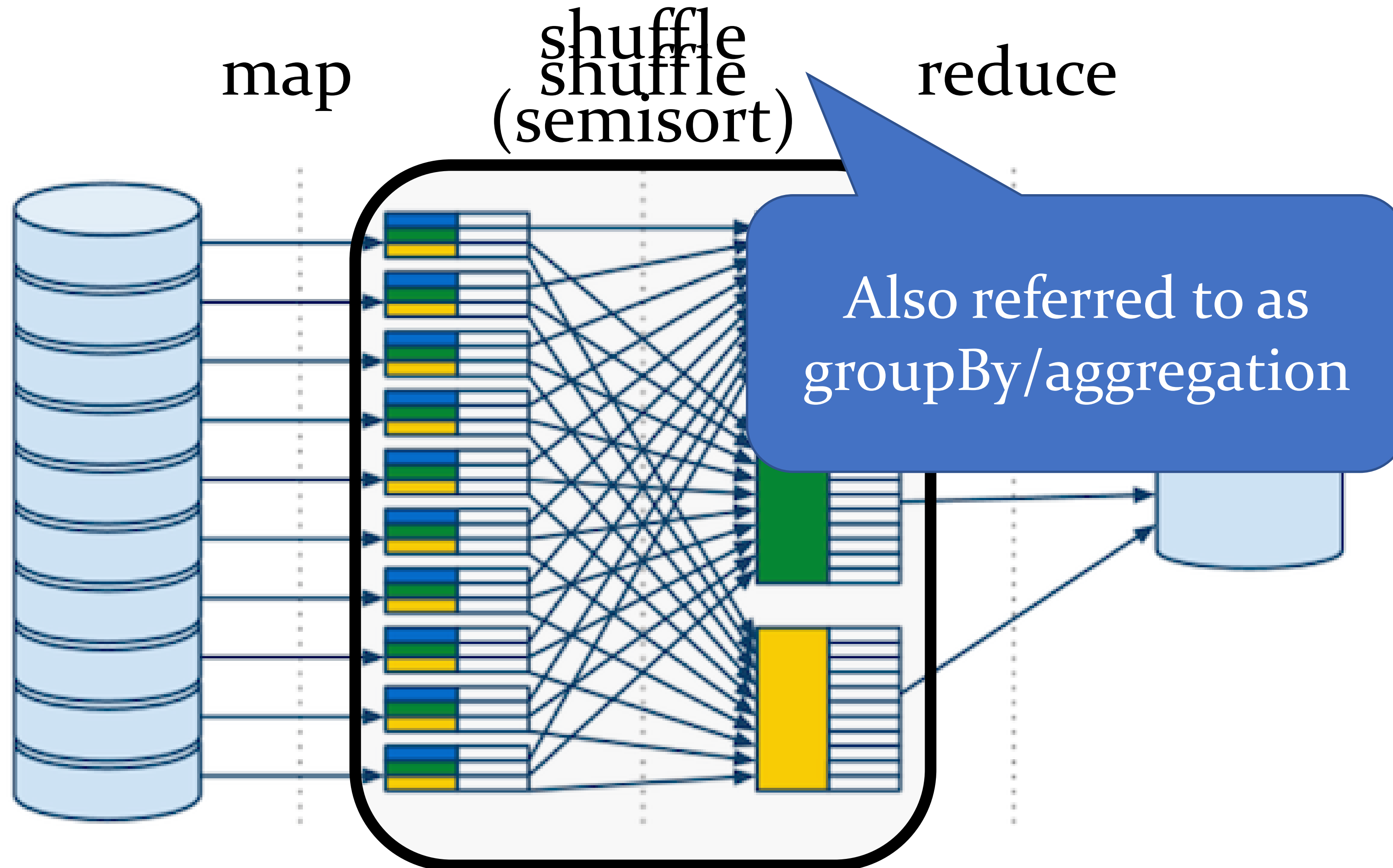
Semisorting is widely used in real world

- Graph algorithms
  - [AAB19, BGSS20, DBS17, DBS21, DGSZ21, Shun20, SS20, DWGS23]
- Computational geometry
  - [BGSS18, BGSS20, WGS20, WYGS21]
- Databases
  - [DHS20, DMK+20, QDT+21]



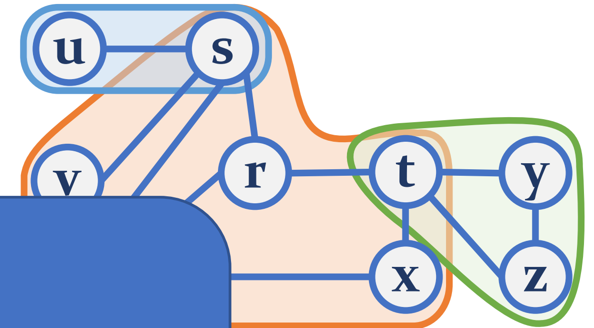


Semisorting is widely used in real world

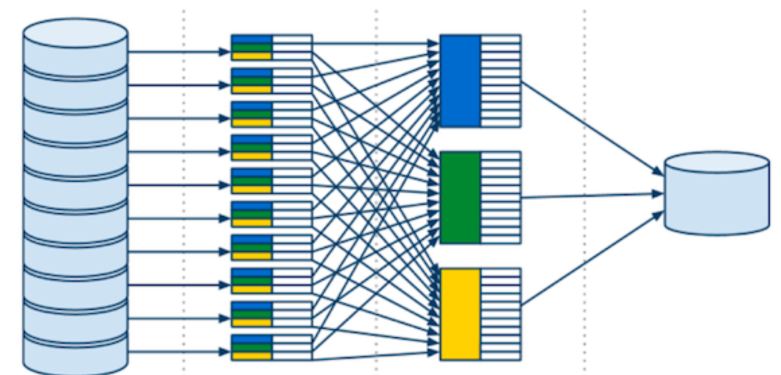


Comparison-based sort as an alternative?

- Graph algorithms
  - [AAB19, BGSS20, DBS17, DBS21, DGS…
- Comp…
  - [BG…
    WY…
- Databases
  - [DHS20, DMK+20, QDT+21]



Semisort can achieve linear work!



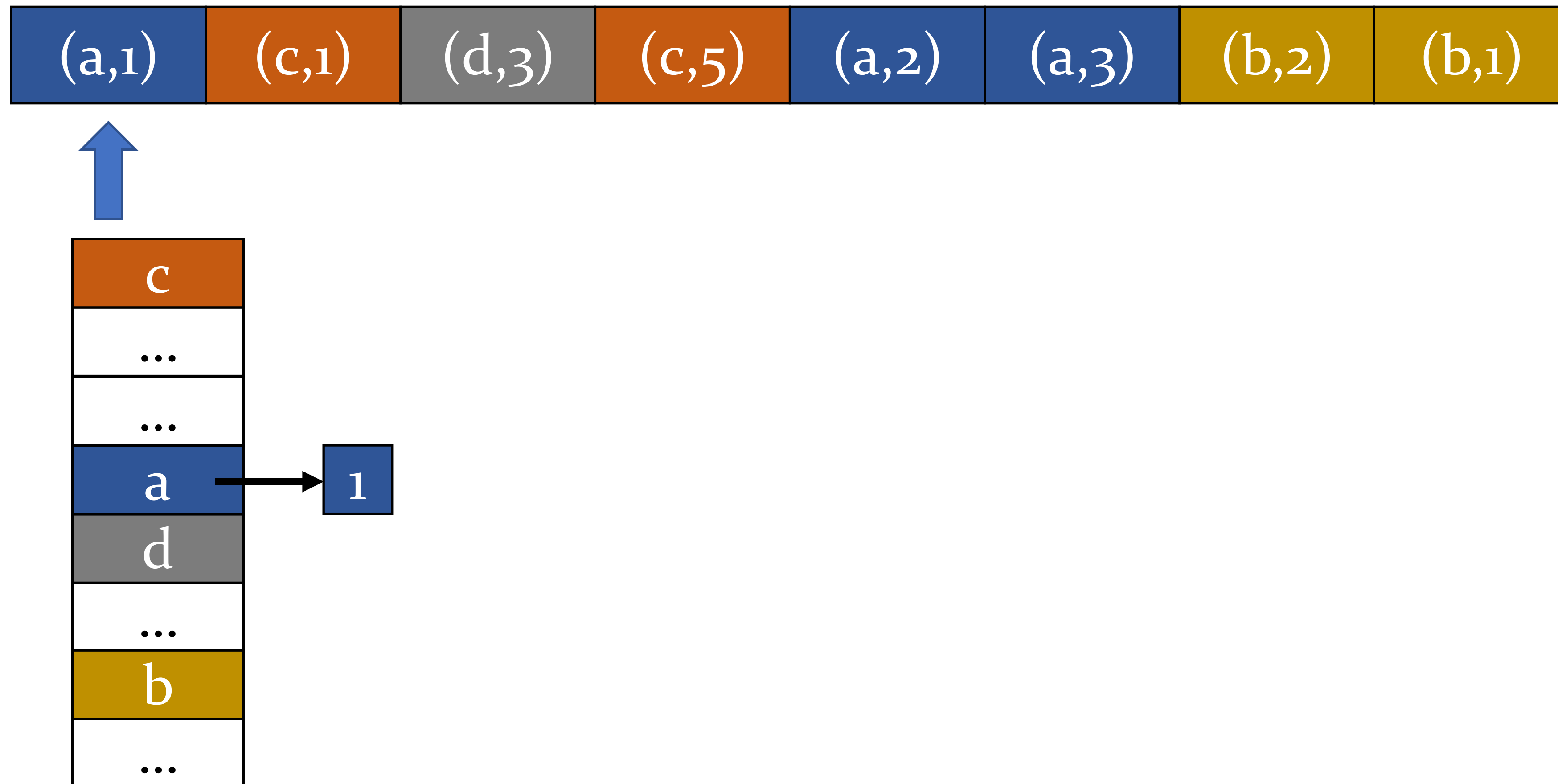
Sequential semisort

Semisorting is easy in sequential!

- Semisorting n elements in $O(n)$ work using a hash table

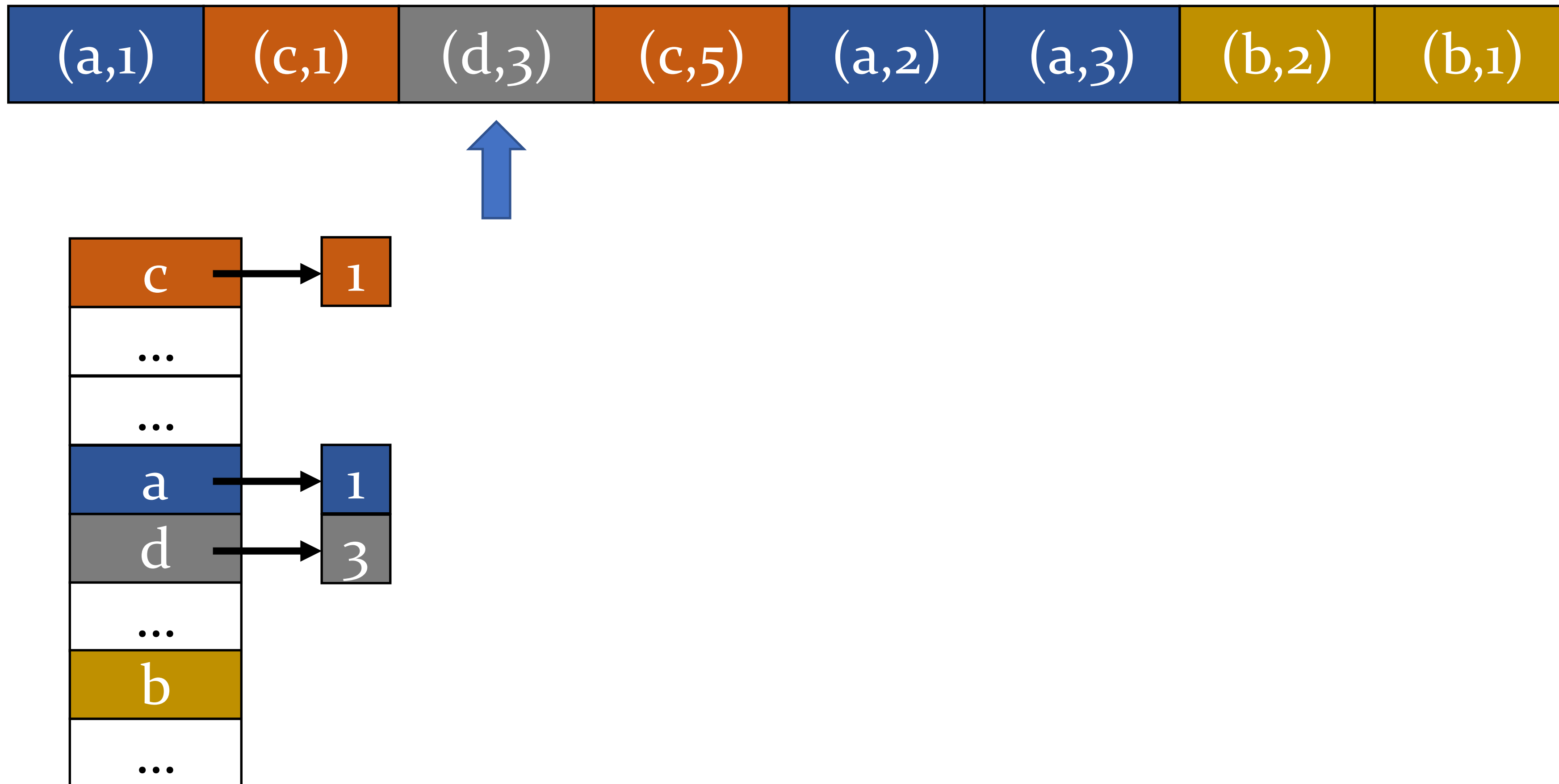
Semisorting is easy in sequential!

- Semisorting n elements in $O(n)$ work using a hash table



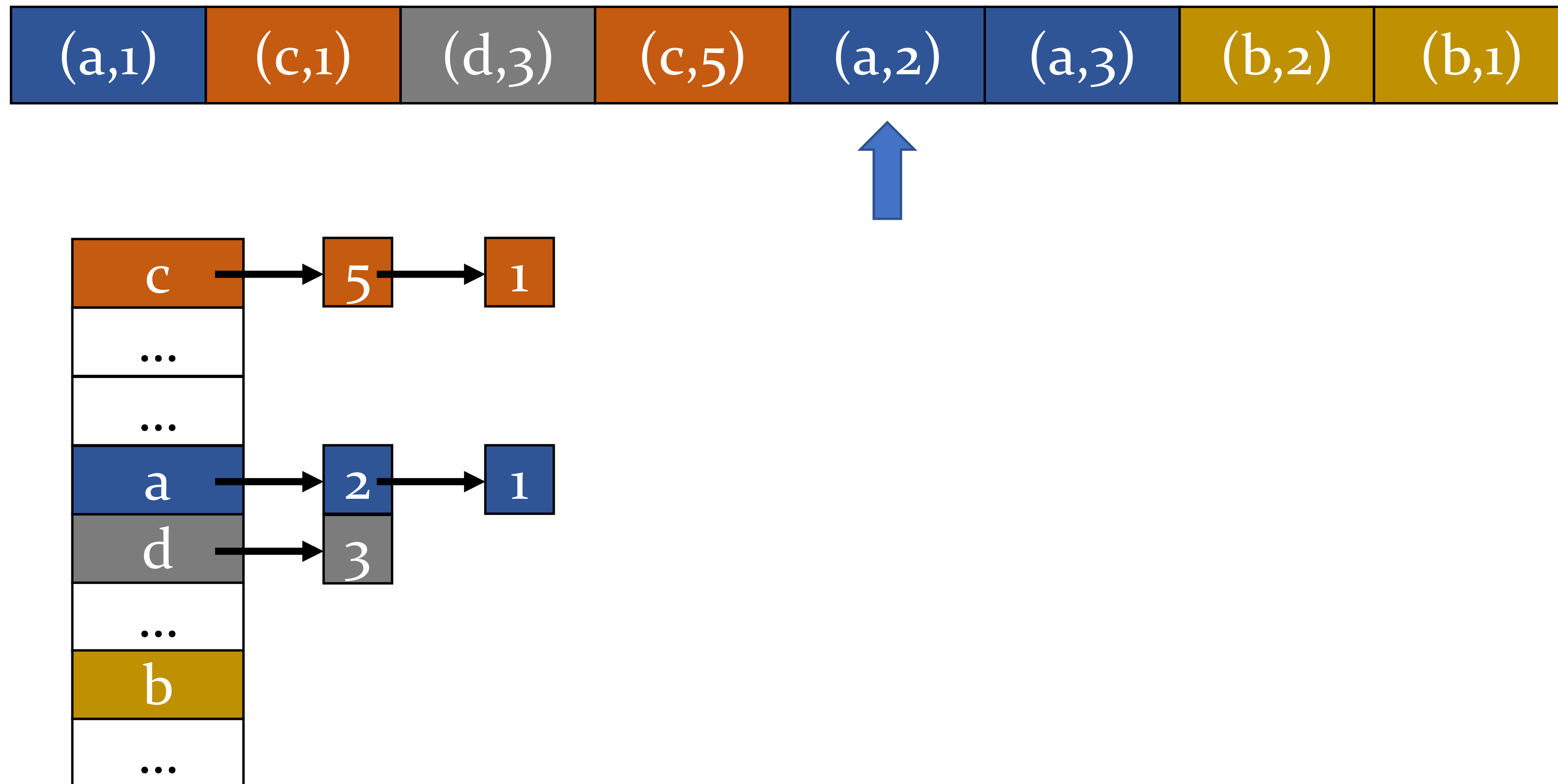
Semisorting is easy in sequential!

- Semisorting n elements in $O(n)$ work using a hash table



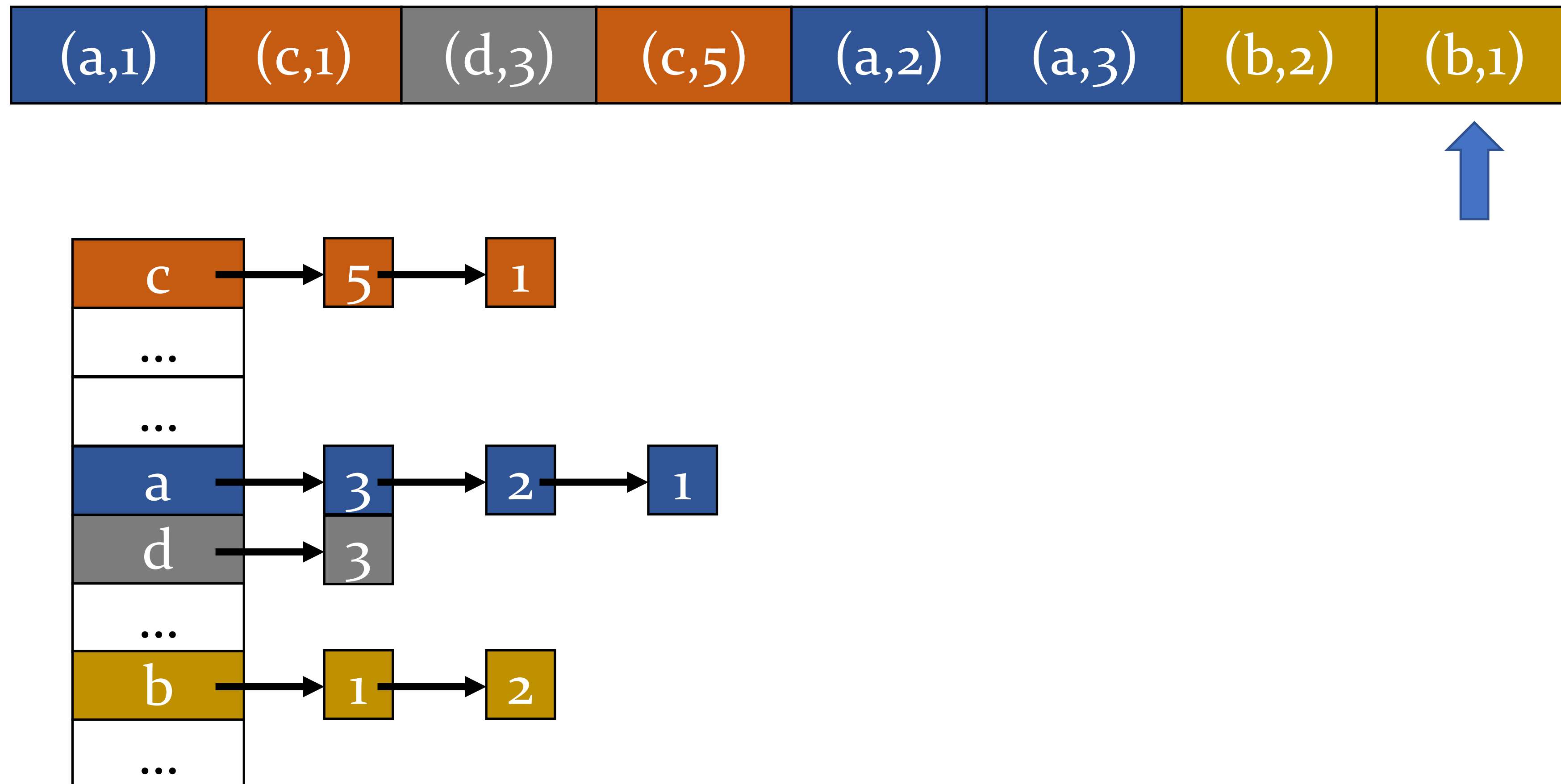
Semisorting is easy in sequential!

- Semisorting n elements in $O(n)$ work using a hash table



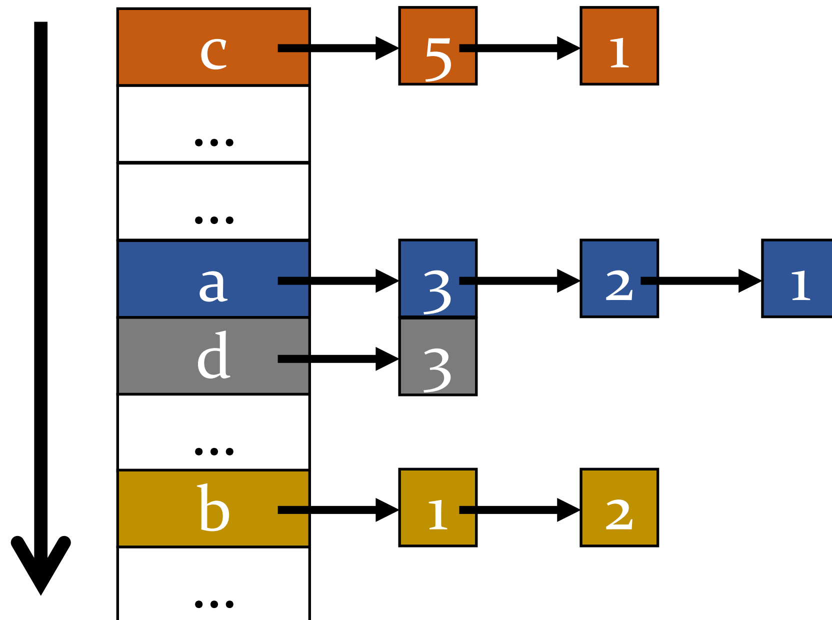
Semisorting is easy in sequential!

- Semisorting n elements in $O(n)$ work using a hash table



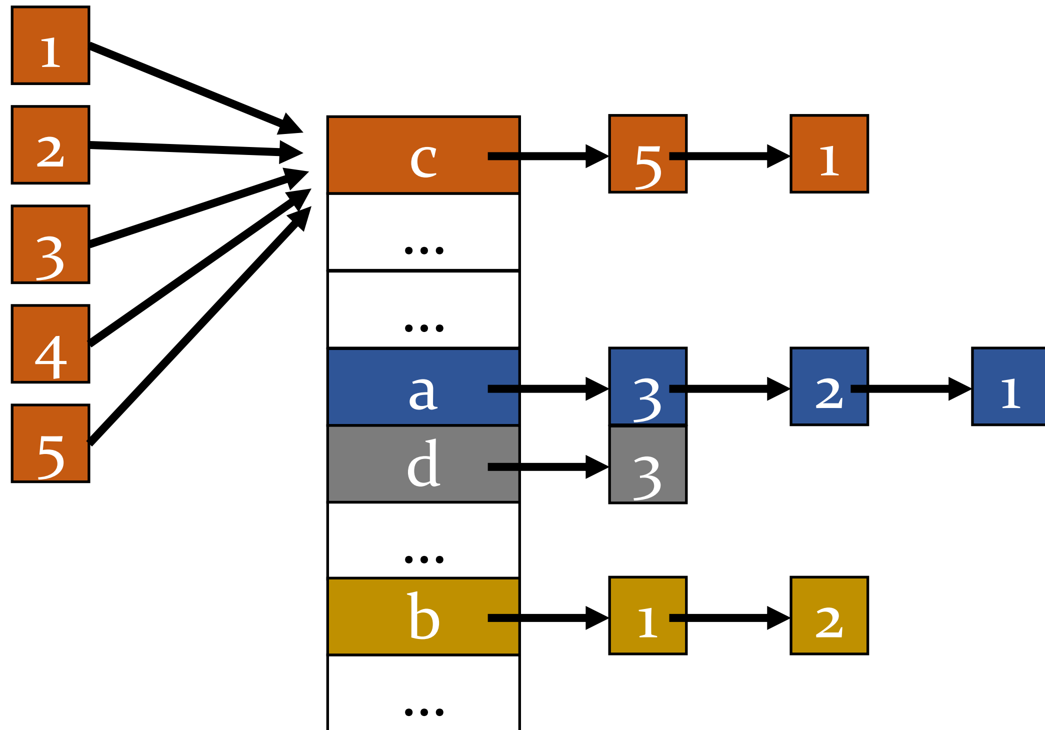
Semisorting is easy in sequential!

- Semisorting n elements in $O(n)$ work using a hash table
- Pack the records into a consecutive array
- However, maintaining linked lists in parallel can be hard



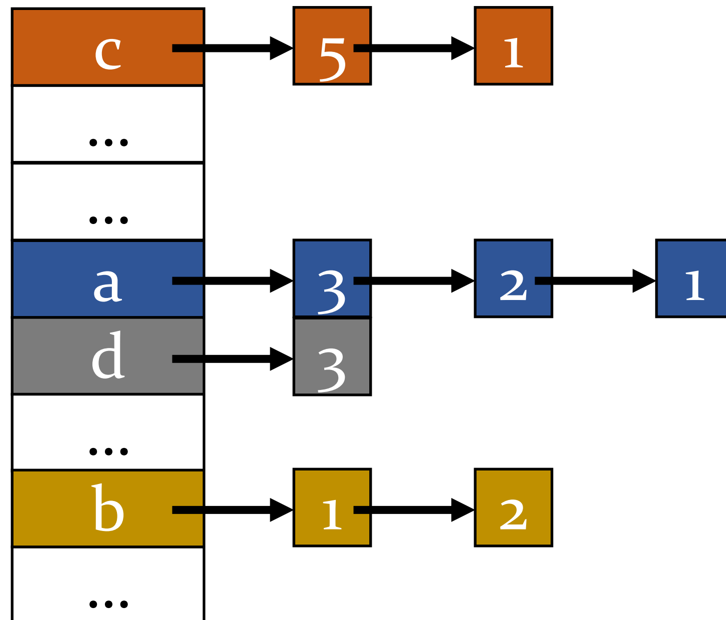
Semisorting is easy in sequential!

- However, maintaining linked lists in parallel can be hard
- Concurrent insertions can cause heavy contention



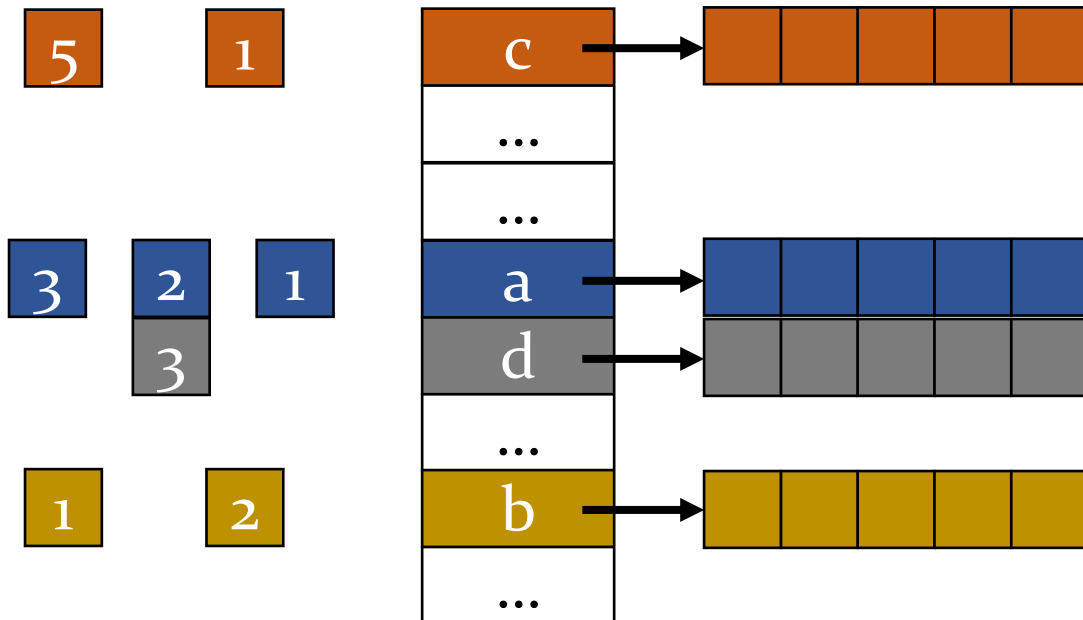
Parallel attempt: random scatter

- Pre-allocate a bucket for each key



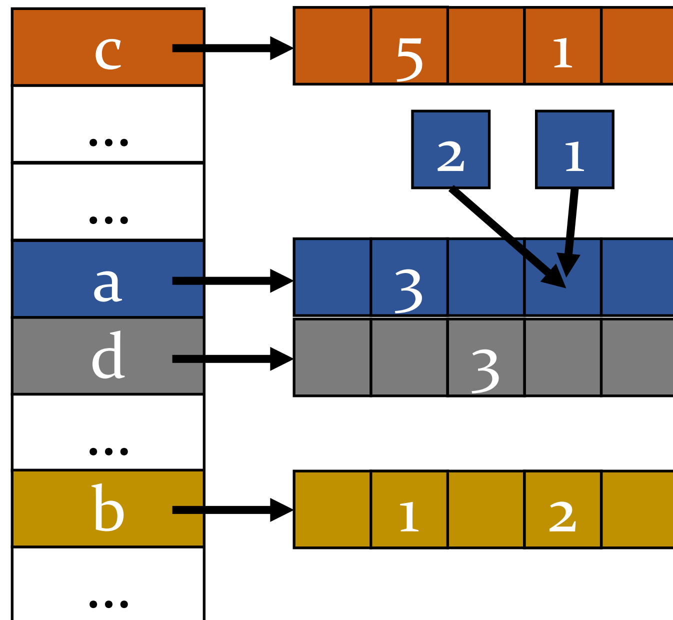
Parallel attempt: random scatter

- Pre-allocate a bucket for each key
- Insert keys simultaneously into random locations



Parallel attempt: random scatter

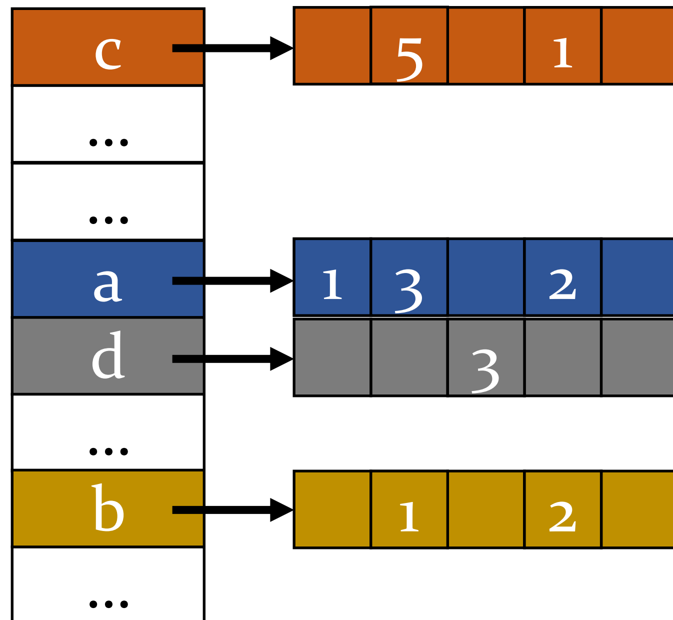
- Pre-allocate a bucket for each key
- Insert keys simultaneously into random locations



When conflicts happen, choose a new random position

Parallel attempt: random scatter

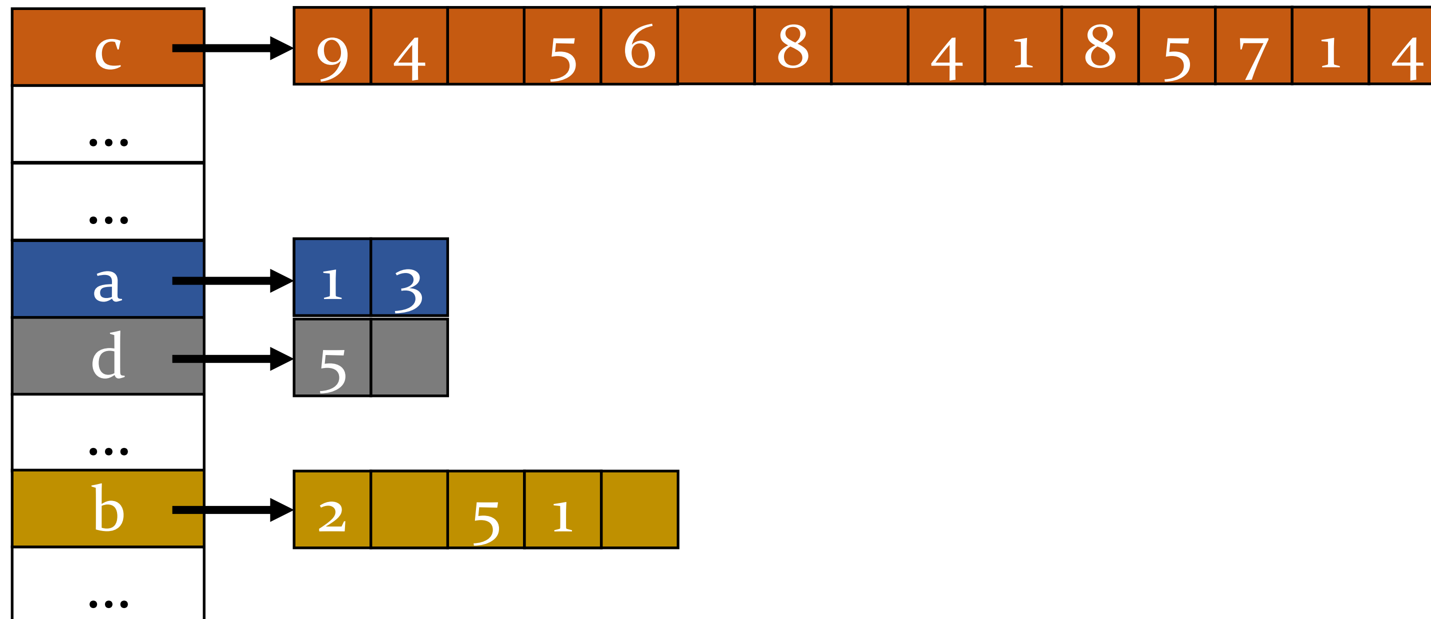
- Pre-allocate a bucket for each key
- Insert keys simultaneously into random locations



When conflicts happen,
choose a new random position

Keys may need buckets of different sizes…

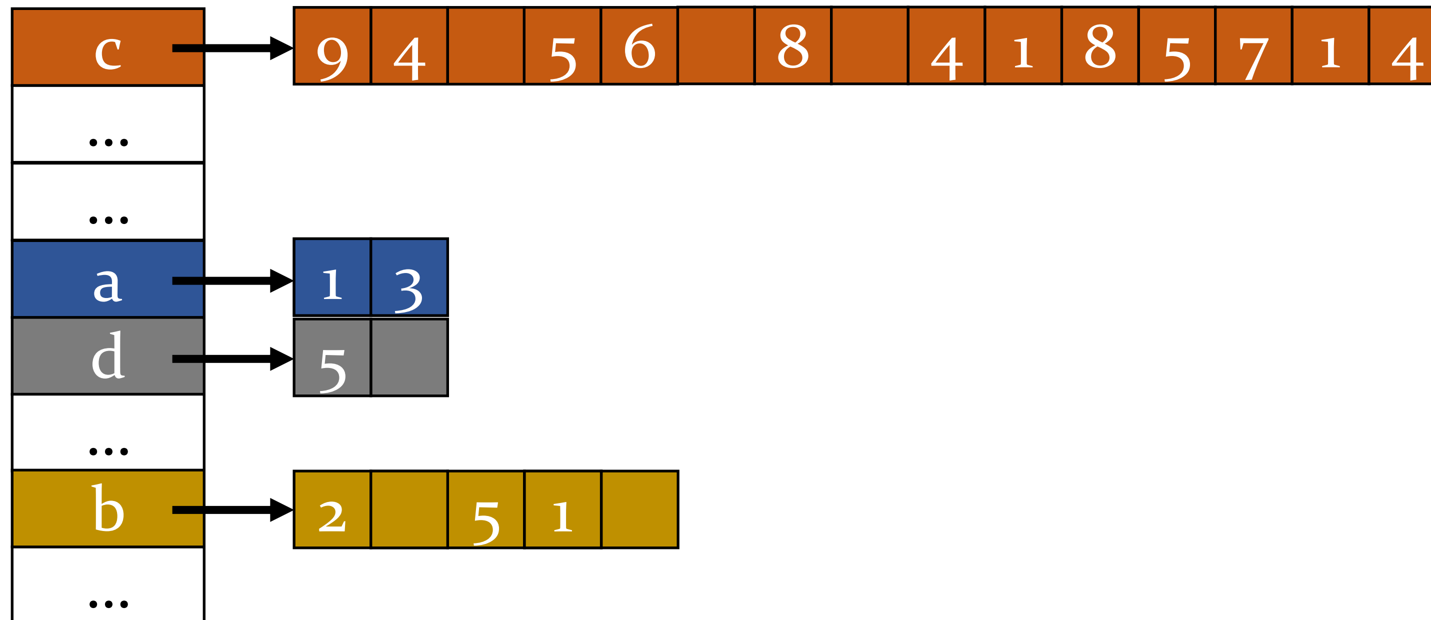
- There can be a key with $O(n)$ records
- There can also be $O(n)$ keys with only $O(1)$ record
- If we pre-allocate a bucket of size n for each key, the complexity can degenerate to $O(n^2)$



Existing parallel semisort

Theoretically-efficient parallel semisort

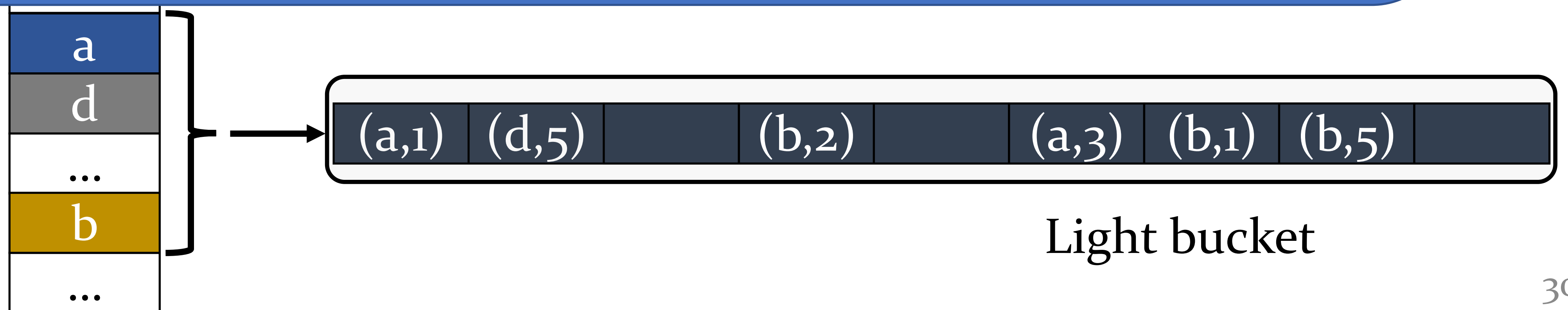
- [GSSB15] Approach: heavy buckets vs. light buckets
  - Each heavy key uses one bucket
  - Multiple light keys share one bucket



Theoretically-efficient parallel semisort

- [GSSB15] Approach: heavy buckets vs. light buckets

GSSB can be implemented in $O(n)$ expected work and space, and $O(\log n)$ span w.h.p.



The GSSB semisort algorithm

- 1. Select a sample S of keys and sort it
  - Partition S into heavy keys and light keys

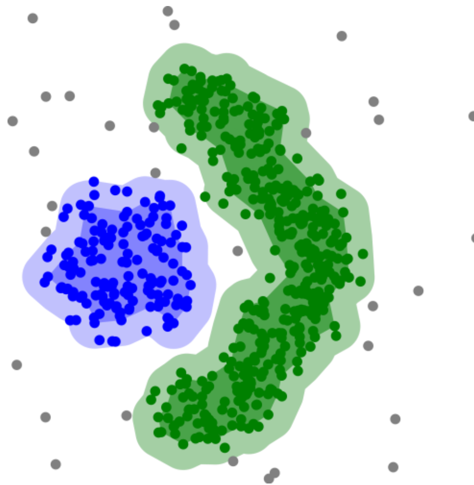
- 2. Scatter each record into its associated bucket

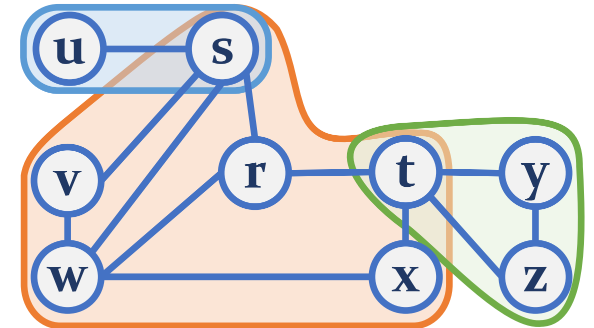
- 3. Semisort light key buckets locally

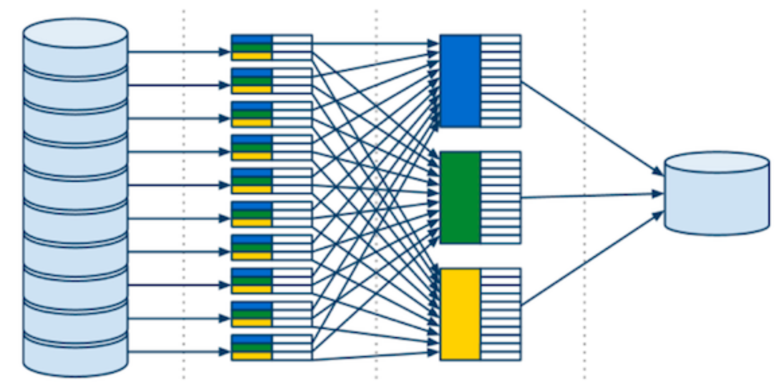
- 4. Pack and output

GSSB is widely "used"

- Graph algorithms
  - [AAB19, BGSS20, DBS17, DBS21, DGSZ21, Shun20, SS20, DWGS23]
- Computational geometry
  - [BGSS18, BGSS20, WGS20, WYGS21]
- Databases
  - [DHS20, DMK+20, QDT+21]
- However, most of these papers
  - Use GSSB in the theoretical analysis
  - Use comparison sort in implementations







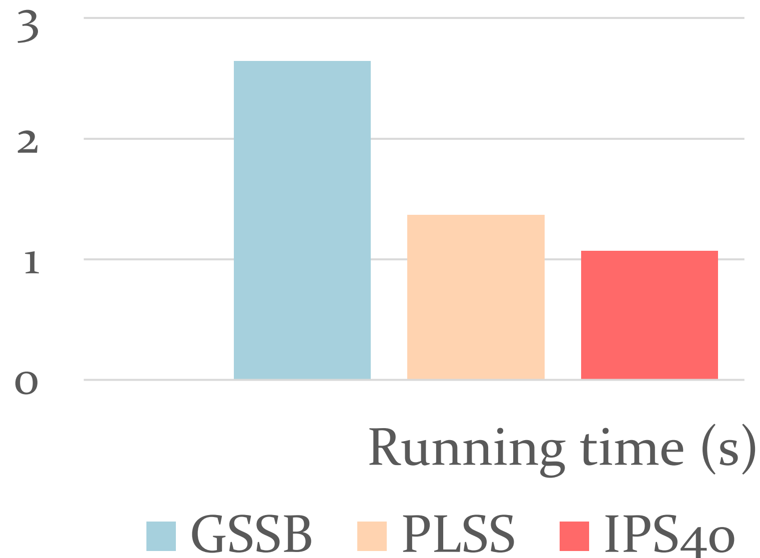
GSSB is widely cited, but never used in practice

- Semisort is asymptotically simpler than comparison-based sort
- However, GSSB is slower than comparison-based sort by up to 60%

GSSB is widely cited, but never used in practice

- However, GSSB is slower than comparison-based sort by up to 60%
- Average running time across multiple distributions in seconds
  - Lower is better

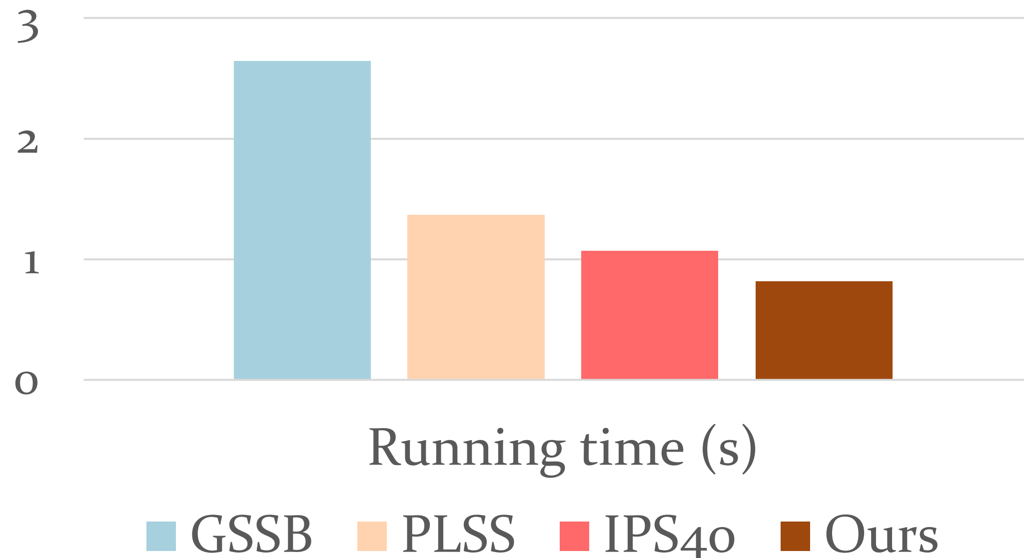
GSSB: GSSB semisort
(Gu et al., SPAA '15)
PLSS: ParlayLib sample sort
(Blelloch et al., SPAA '20)
IPS4o: IPS4o sample sort
(Axtmann et al., ESA '17)



GSSB is widely cited, but never used in practice

- However, GSSB is slower than comparison-based sort by up to 60%
- Average running time across multiple distributions in seconds
  - Lower is better
- Our semisort implementation is the fastest

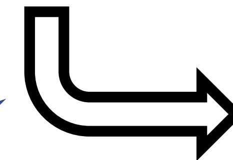
GSSB: GSSB semisort
(Gu et al., SPAA '15)
PLSS: ParlayLib sample sort
(Blelloch et al., SPAA '20)
IPS4o: IPS4o sample sort
(Axtmann et al., ESA '17)



Restrictions of the GSSB implementation

- 1. Select a sample S of keys and sort it
  - Partition S into heavy keys and light keys
- 2. Scatter each record into its associated bucket
- 3. Semisort light key buckets locally
- 4. Pack and output

Use random scatter
☹ I/O inefficient
☹ Not race-free
☹ Use substantial extra space



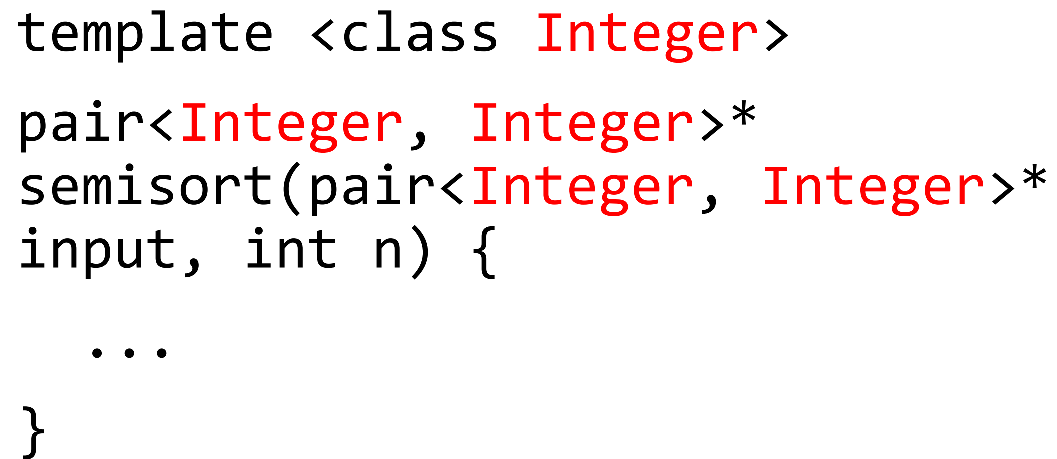
☺ I/O-efficient
☺ race-free

Ideas borrowed from existing sorting algorithms

Restrictions of the GSSB implementation

- 1. Select a sample S of keys and sort it
  - Partition S into heavy keys and light keys
- 2. Scatter each record into its associated bucket
- 3. Semisort light key buckets locally
- 4. Pack and output

GSSB interface vs. our interface



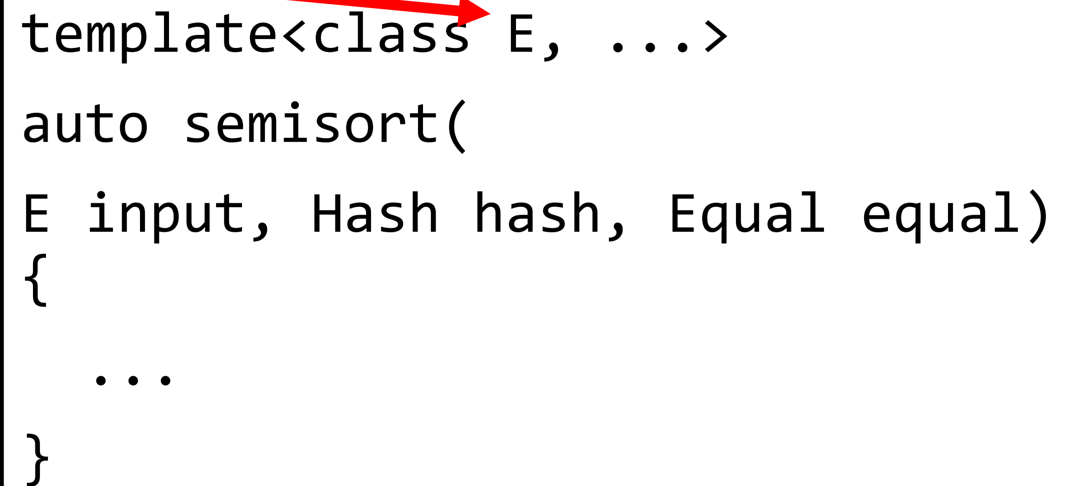
GSSB interface vs. our interface

```
// Hash input type to integers
template <class Integer>
pair<Integer, Integer>*
semisort(pair<Integer, Integer>*
input, int n) {
  ...
}
// map integers back to input
```

GSSB interface vs. our interface

Support arbitrary data type!

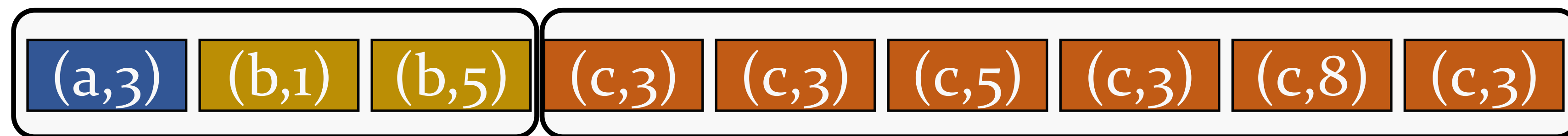
```
// Hash input type to integers
template <class Integer>
void semisort(pair<Integer,
Integer>* input, int n) {
  ...
}
// map integers back to input
```



Our semisort algorithm

Our new algorithm: sampling and bucketing

- Sample a $n_L \log n$ of keys
- Use comparison sort to count the occurrences
- Estimate keys are heavy or light
  - Light: appears $< c \log n$ times
  - Heavy: appears $\geq c \log n$ times

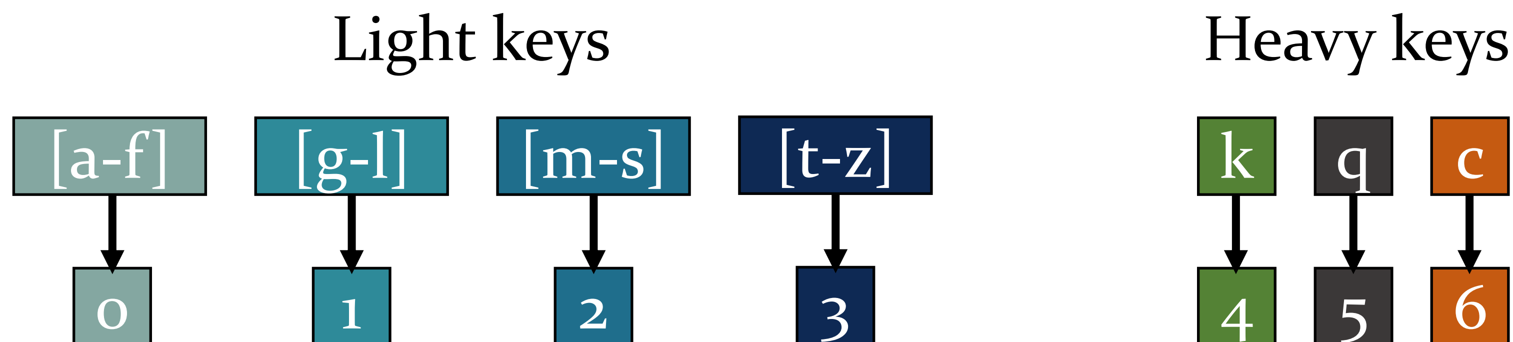


Light keys

Heavy keys

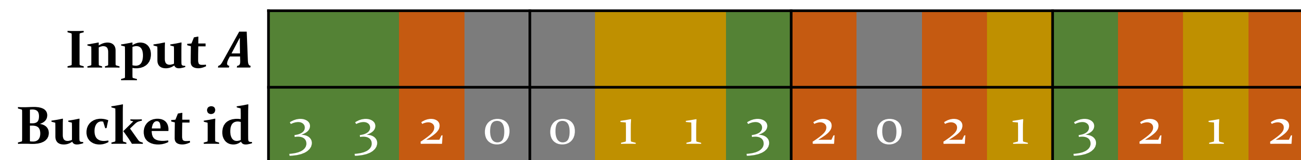
Our new algorithm: sampling and bucketing

- Estimate keys are heavy or light
  - Light: appears $< c \log n$ times
  - Heavy: appears $\geq c \log n$ times
- Assign a bucket id to each key
- b buckets in total. #light buckets ≈ #heavy buckets = $O(b)$



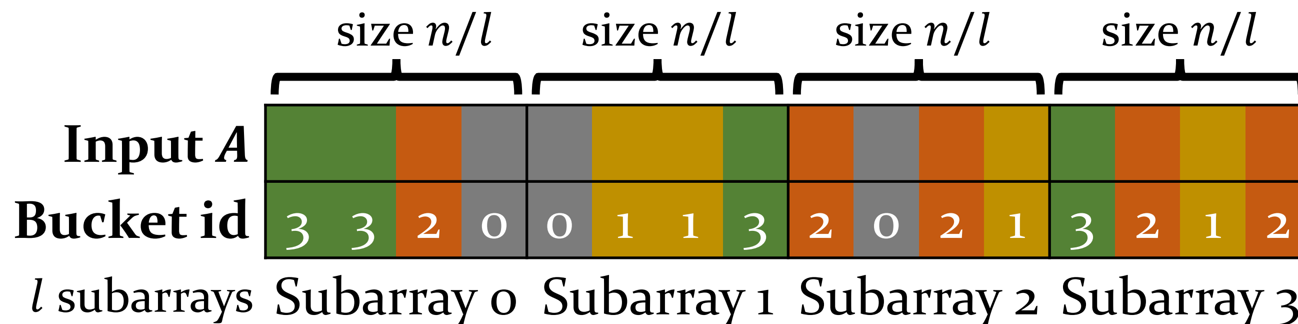
Our new algorithm: blocked distributing

- We borrowed the idea from sample sort [BGS10], use *exact* counting
- Goal: distribute such that keys in the same bucket are contiguous



Our new algorithm: blocked distributing

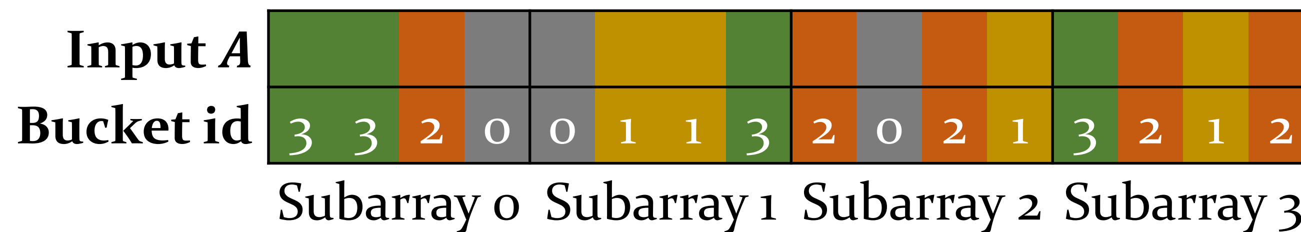
- We borrowed the idea from sample sort [BGS10], use *exact* counting
- Goal: distribute such that keys in the same bucket are contiguous
- Divide the input array into l subarrays
- Initialize a counting matrix C with size $b \times l$
- For each subarray, count the number of records falling into each bucket



Counting Matrix C	Buckets 0	1	2	3
Subarray 0	0	0	0	0
Subarray 1	0	0	0	0
Subarray 2	0	0	0	0
Subarray 3	0	0	0	0

Our new algorithm: blocked distributing

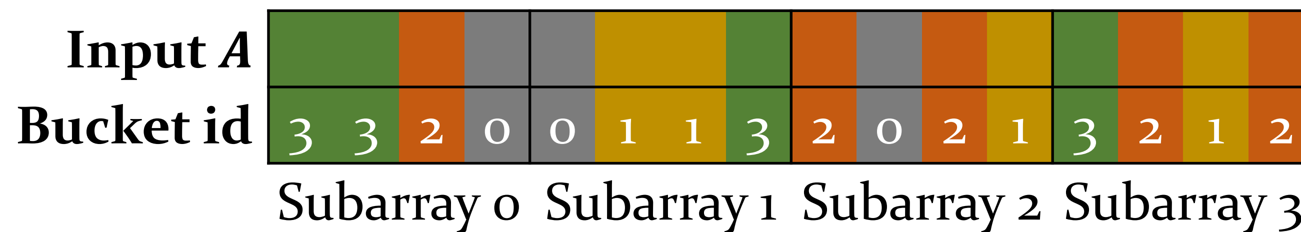
- Distribute A into A' using the counting matrix
- All heavy keys are already in their final destination



Counting Matrix C	Buckets 0	1	2	3
Subarray 0	0	3	7	12
Subarray 1	1	3	8	14
Subarray 2	2	5	8	15
Subarray 3	3	6	10	15

Our new algorithm: blocked distributing

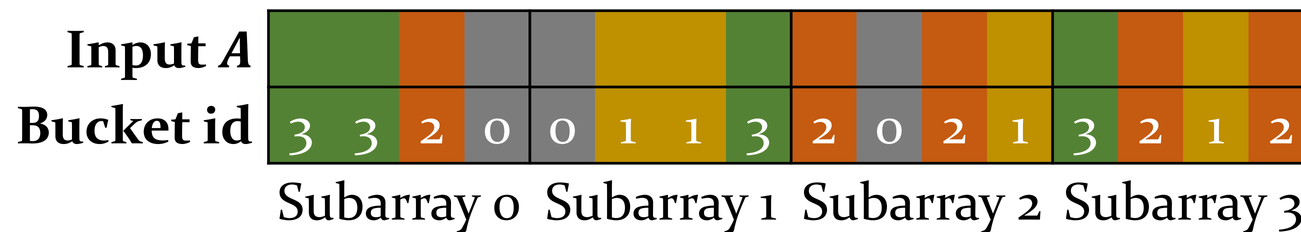
- Distribute A into A’ using the counting matrix
- All heavy keys are already in their final destination



Counting Matrix C	Buckets 0	1	2	3
Subarray 0	0	3	7	12
Subarray 1	1	3	8	14
Subarray 2	2	5	8	15
Subarray 3	3	6	10	15

Our new algorithm: blocked distributing

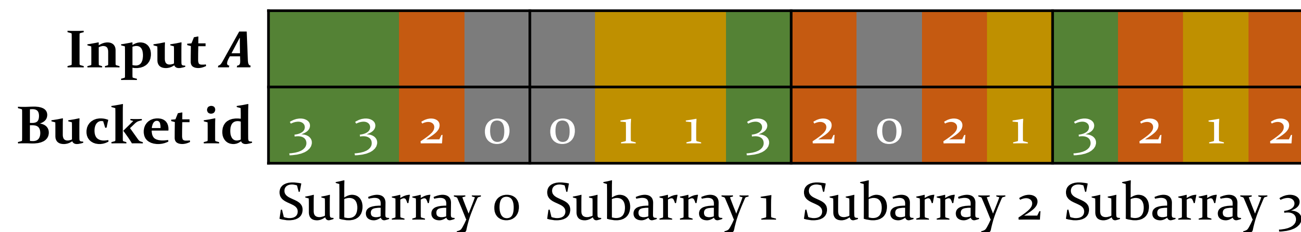
- Distribute A into A' using the counting matrix
- All heavy keys are already in their final destination
- Light keys need to be refined

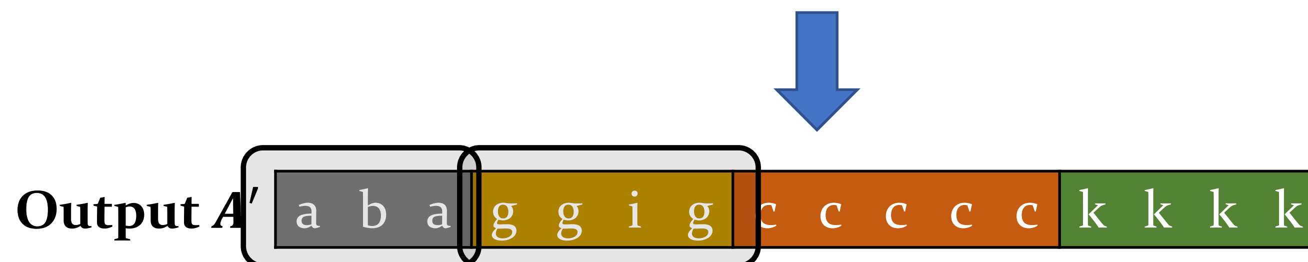


Counting Matrix C	Buckets 0	1	2	3
Subarray 0	0	3	7	12
Subarray 1	1	3	8	14
Subarray 2	2	5	8	15
Subarray 3	3	6	10	15

Our new algorithm: local refining

- Distribute A into A' using the counting matrix
- All heavy keys are already in their final destination
- Light keys need to be refined
- Semisort each light bucket locally in parallel





Counting Matrix C	Buckets 0	1	2	3
Subarray 0	0	3	7	12
Subarray 1	1	3	8	14
Subarray 2	2	5	8	15
Subarray 3	3	6	10	15

Heavy keys are easier to be processed

- Heavy keys don’t need to be refined
- Light keys need to be further reordered

Light keys

Heavy keys

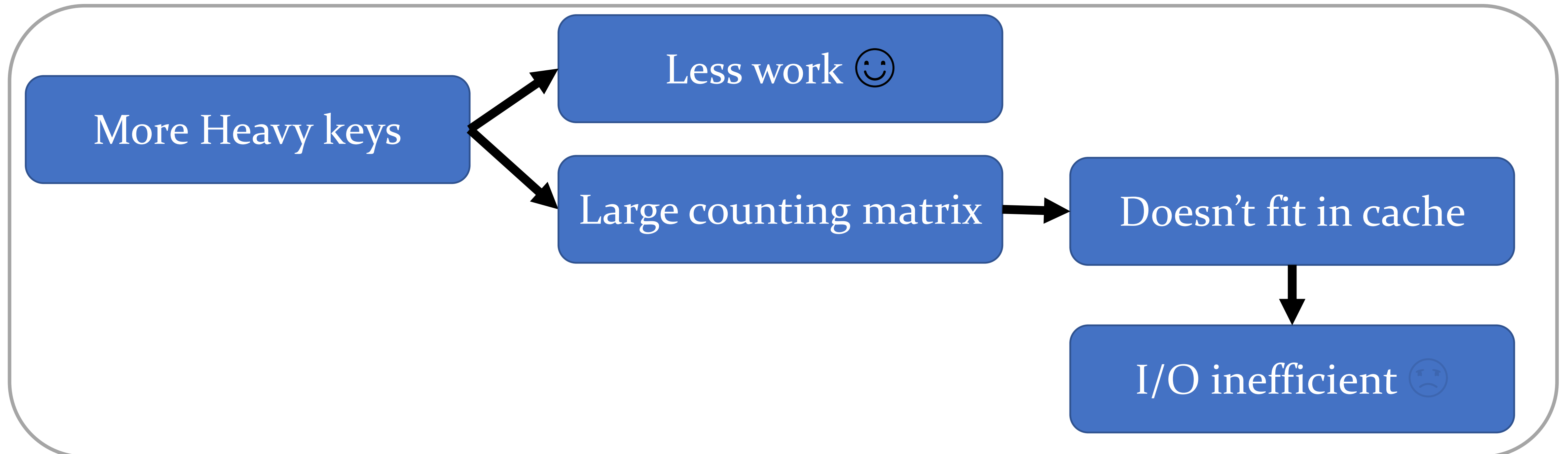
Heavy keys are easier to be processed

- Heavy keys don't need to be refined
- Light keys need to be further reordered
- We want to detect as many heavy keys as possible
- Size of the counting matrix: $b \times l$
  - Increase #buckets b will result in larger counting matrix

a	b	a	g	g	i	g	c	c	c	c	c	c	c	c	c	c	c	c	k	k	k	k	k	k	k	k

Large counting matrix vs. small counting matrix

- Semisorting can be recursive!



Our new algorithm: local refining

- Distribute A into A' using the counting matrix
- All heavy keys are already in their final destination
- Light keys needed to be refined
- Semisort each light bucket recursively in parallel

Input A																
Bucket id	3	3	2	0	0	1	1	3	2	0	2	1	3	2	1	2

Subarray 0 Subarray 1 Subarray 2 Subarray 3

Output A'	a	b	a	g	g	i	g	c	c	c	c	c	k	k	k	k

Counting Matrix C	Buckets 0	1	2	3
Subarray 0	0	3	7	12
Subarray 1	1	3	8	14
Subarray 2	2	5	8	15
Subarray 3	3	6	10	15

Our new algorithm: local refining

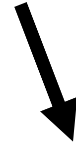
- We don’t have to detect all heavy keys in the first recursion
  - Medium-level heavy keys can be detected in the recursion
  - Fewer heavy buckets → smaller counting matrix → fit in cache
- Recurse until the subproblem size is small enough
  - Semisort$_=$: use hash table with open chaining in base case
  - Semisort$_<$: use comparison-based sort in base case

g is a heavy key in the second-level recursion

Theoretical analysis

- We carefully choose the parameters such that we can preserve some theoretical guarantees
- Assume $M > \sqrt{n}$
  - Semisort$_=$ has $O(n)$ work, $O(n^{3/4})$ span, and $O(n/B)$ I/O cost whp.
  - Semisort$_<$ has $O(n \log n)$ work, $O(n^{3/4})$ span, and $O(n/B)$ I/O cost whp.
- Semisort$_<$ can sometime achieve better performance in practice.
- **Empirically better parallel scalability** than some existing implementations with polylog span
  - Large span is a result of applying granularity control

We have discussed this



High-Performance and Flexible Parallel Algorithms for Semisort and Related Problems

Flexibility of our semisort algorithms

- Allows arbitrary input types
  - Resolve conflicts under the hood
- Natural support for histogram and collect-reduce
  - Stable and (internally-)deterministic
- Easy to use, compatible with ParlayLib

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
- 3 distributions (uniform, exponential, Zipfian) are tested
  - Each with 5 different parameters
- 4 our algorithms:
  - $Ours_=$: our semisort= algorithm
  - $Ours_<$: our semisort< algorithm
  - Ours-$i_=$: our integer version $semisort_=$ algorithm
  - Ours-$i_<$: our integer version $semisort_<$ algorithm

ParlayLib: https://github.com/cmuparlay/parlaylib

6 competitors

- 2 sample sorts
  - IPS^{4}o: IPS^{4}o sample sort (Axtmann et al., ESA ’17)
  - PLSS: ParlayLib sample sort (Blelloch et al., SPAA ’20)
- 3 integer sorts
  - 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)
- 1 semisort
  - GSSB semisort (Gu et al., SPAA ’15)

Performance heatmap

- Each column is an algorithm, each row is a distribution-parameter instance
- Any/integer input type
- Relative speedup over the fastest algorithm in each instance on 96 cores
  - Smaller/green is better
- Geometric mean on distributions of the same type

		Any input type				Integer input type					
		Ours$_{=}$	Ours$_{<}$	PLSS	IPS^{4}o	Ours-i$_{=}$	Ours-i$_{<}$	PLIS	GSSB	RS	IPS^{2}Ra
Uniform	10	1.03	1.00	1.59	1.22	1.06	1.00	3.12	4.51	2.07	6.13
	10^3	1.00	1.00	1.32	1.03	1.00	1.00	1.97	5.97	2.21	2.40
	10^5	1.00	1.00	1.82	1.51	1.00	1.00	1.73	3.45	2.01	1.50
	10^7	1.00	1.00	1.43	1.06	1.09	1.00	1.28	2.86	1.97	1.18
	10^9	1.00	1.15	1.57	1.11	1.00	1.36	1.15	2.86	1.43	1.15
	AVG	1.01	1.03	1.54	1.18	1.03	1.06	1.73	3.77	1.92	1.97
Exponential	1	1.00	1.00	1.73	1.28	1.01	1.00	2.67	3.55	2.21	1.53
	0.7	1.00	1.00	1.76	1.35	1.00	1.00	2.38	3.55	2.14	1.45
	0.5	1.01	1.00	1.80	1.42	1.01	1.00	2.13	3.53	2.02	1.45
	0.2	1.00	1.00	1.74	1.51	1.01	1.00	1.67	3.57	1.98	1.46
	0.1	1.02	1.00	1.66	1.44	1.02	1.00	1.51	3.52	1.89	1.40
	AVG	1.01	1.00	1.74	1.40	1.01	1.00	2.03	3.54	2.05	1.46
Zipfian	1.5	1.00	1.01	3.04	2.28	1.03	1.00	3.68	3.89	2.67	10.1
	1.2	1.00	1.01	1.95	1.58	1.01	1.00	2.71	3.69	2.55	4.97
	1	1.00	1.08	1.36	1.16	1.00	1.03	1.70	3.25	1.89	2.04
	0.8	1.00	1.15	1.49	1.11	1.00	1.04	1.21	2.96	1.56	1.21
	0.6	1.00	1.16	1.57	1.12	1.00	1.06	1.17	2.88	1.47	1.15
	AVG	1.00	1.08	1.80	1.39	1.01	1.03	1.89	3.31	1.97	2.70
AVG		1.00	1.04	1.69	1.32	1.02	1.03	1.88	3.54	1.98	1.98

1 1.1 1.2 1.5 2 4 >4

AVG = Geometric Mean

Performance heatmap

- Our algorithm is at least one of the fastest
  - The other one is always competitive
- On average, our algorithm achieves at least 25% speedup
  - 3.4x faster than GSSB

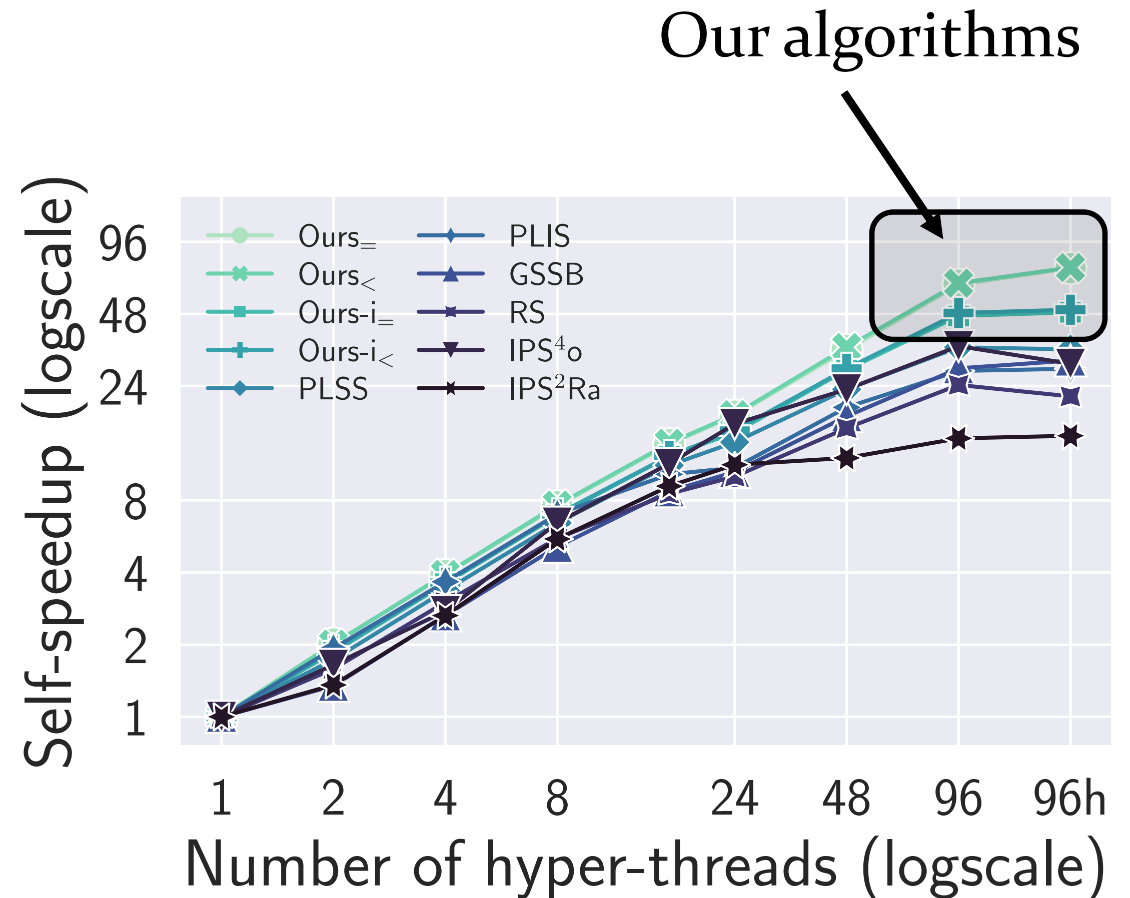
		Any input type				Integer input type					
		Ours$_=$	Ours$_<$	PLSS	IPS^{4}o	Ours-i$_=$	Ours-i$_<$	PLIS	GSSB	RS	IPS^{2}Ra
Uniform	10	1.03	1.00	1.59	1.22	1.06	1.00	3.12	4.51	2.07	6.13
	10^3	1.00	1.00	1.32	1.03	1.00	1.00	1.97	5.97	2.21	2.40
	10^5	1.00	1.00	1.82	1.51	1.00	1.00	1.73	3.45	2.01	1.50
	10^7	1.00	1.00	1.43	1.06	1.09	1.00	1.28	2.86	1.97	1.18
	10^9	1.00	1.15	1.57	1.11	1.00	1.36	1.15	2.86	1.43	1.15
	AVG	1.01	1.03	1.54	1.18	1.03	1.06	1.73	3.77	1.92	1.97
Exponential	1	1.00	1.00	1.73	1.28	1.01	1.00	2.67	3.55	2.21	1.53
	0.7	1.00	1.00	1.76	1.35	1.00	1.00	2.38	3.55	2.14	1.45
	0.5	1.01	1.00	1.80	1.42	1.01	1.00	2.13	3.53	2.02	1.45
	0.2	1.00	1.00	1.74	1.51	1.01	1.00	1.67	3.57	1.98	1.46
	0.1	1.02	1.00	1.66	1.44	1.02	1.00	1.51	3.52	1.89	1.40
	AVG	1.01	1.00	1.74	1.40	1.01	1.00	2.03	3.54	2.05	1.46
Zipfian	1.5	1.00	1.01	3.04	2.28	1.03	1.00	3.68	3.89	2.67	10.1
	1.2	1.00	1.01	1.95	1.58	1.01	1.00	2.71	3.69	2.55	4.97
	1	1.00	1.08	1.36	1.16	1.00	1.03	1.70	3.25	1.89	2.04
	0.8	1.00	1.15	1.49	1.11	1.00	1.04	1.21	2.96	1.56	1.21
	0.6	1.00	1.16	1.57	1.12	1.00	1.06	1.17	2.88	1.47	1.15
	AVG	1.00	1.08	1.80	1.39	1.01	1.03	1.89	3.31	1.97	2.70
AVG		1.00	1.04	1.69	1.32	1.02	1.03	1.88	3.54	1.98	1.98

1 1.1 1.2 1.5 2 4 >4

AVG = Geometric Mean

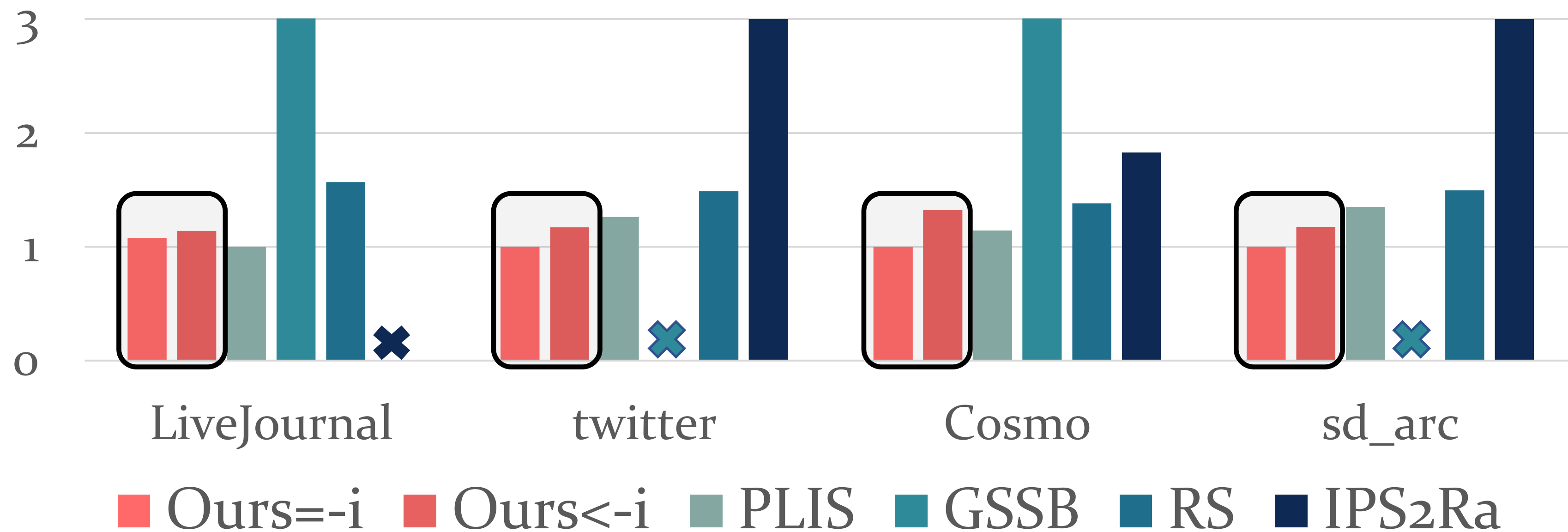
Parallel scalability

- Self-relative speedup from 1 core to 96 cores
  - Larger is better
  - In log scale
- Our algorithms achieve the top-tier speedup (almost linear)
  - Although our algorithms do not have polylog span
- The self-speedup of $\text{semisort}_=$ and $\text{semisort}_<$ is 50–80×



Graph transposing

- Transpose a directed graph $G = (V, E)$ into $G^T = (V, E^T)$ where $E^T = \{(u, v): (v, u) \in E\}$, and semisort by the first endpoint
- Parallel running time normalized to the fastest algorithm (Smaller is better)
- Our algorithm is the fastest on 3 graphs out of 4



We have an 8-page in-depth experimental study in the full version

- Input size scalability
- Varying key lengths
- Parallel scalability
- 2 real-world applications
- Collect reduce

Summary

Summary

- High-performance
  - Almost achieve the best performance in all our experiments
  - Up to 2.5x speedup
  - Highly parallelized
- Flexible
  - Allow any input type
  - Natural support for histogram and collect-reduce
  - Easy to use, compatible with ParlayLib
- Future work
  - Theory: semisort algorithms with linear work, polylog span, and I/O efficiency
  - Practice: the ideas in semisort can be applied to sample/integer sort

Summary

- High-performance
  - Almost achieve the best performance in all our experiments
  - Up to 2.5x speedup
  - Highly parallelized
- Flexible
  - Allow any input type
  - Natural support for histogram and collect-reduce
  - Easy to use, compatible with ParlayLib
- Full version: https://arxiv.org/abs/2304.10078
- Code: https://github.com/ucrparlay/Parallel-Semisort
- Contact: Xiaojun Dong from UC Riverside (xdong038@ucr.edu)