A Top-Down Parallel Semisort

Yan Gu
Julian Shun
Yihan Sun
Guy Blelloch

Carnegie Mellon University

What is semisort?

key	45	12	45	61	28	61	61	45	28	45
Value	2	5	3	9	5	9	8	1	7	5

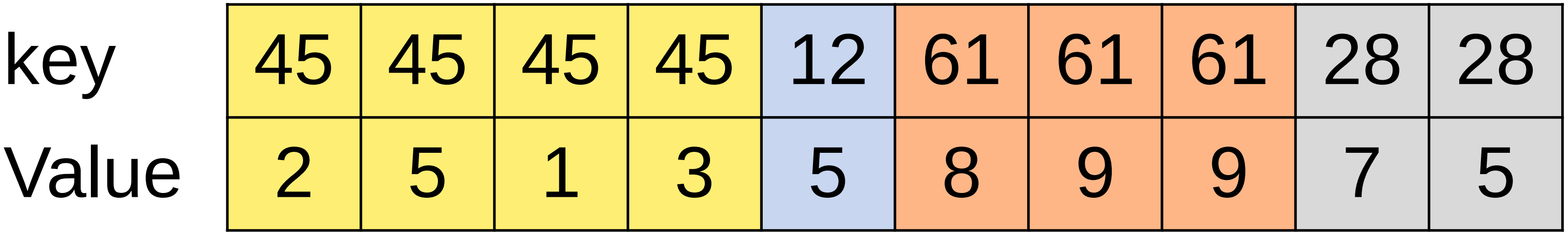
- Input:
  - An array of records with associated keys
  - Assume keys can be hashed to the range $[n^k]$
- Goal:
  - All records with equal keys should be adjacent

What is semisort?

key	12	61	61	61	45	45	45	45	28	28
Value	5	8	9	9	2	5	1	3	7	5

- Input:
  - An array of records with associated keys
  - Assume keys can be hashed to the range $[n^k]$
- Goal:
  - All records with equal keys should be adjacent

What is semisort?



key	45	45	45	45	12	61	61	61	28	28
Value	2	5	1	3	5	8	9	9	7	5

- Input:
  - An array of records with associated keys
  - Assume keys can be hashed to the range $[n^k]$
- Goal:
  - All records with equal keys should be adjacent
  - Different keys are not necessarily sorted
  - Records with equal keys do not need to be sorted by their values

What is semisort?

key	45	45	45	45	12	61	61	61	28	28
Value	1	5	3	2	5	8	9	9	7	5

- Input:
  - An array of records with associated keys
  - Assume keys can be hashed to the range $[n^k]$
- Goal:
  - All records with equal keys should be adjacent
  - Different keys are not necessarily sorted
  - Records with equal keys do not need to be sorted by their values

Why is parallel semisort important?

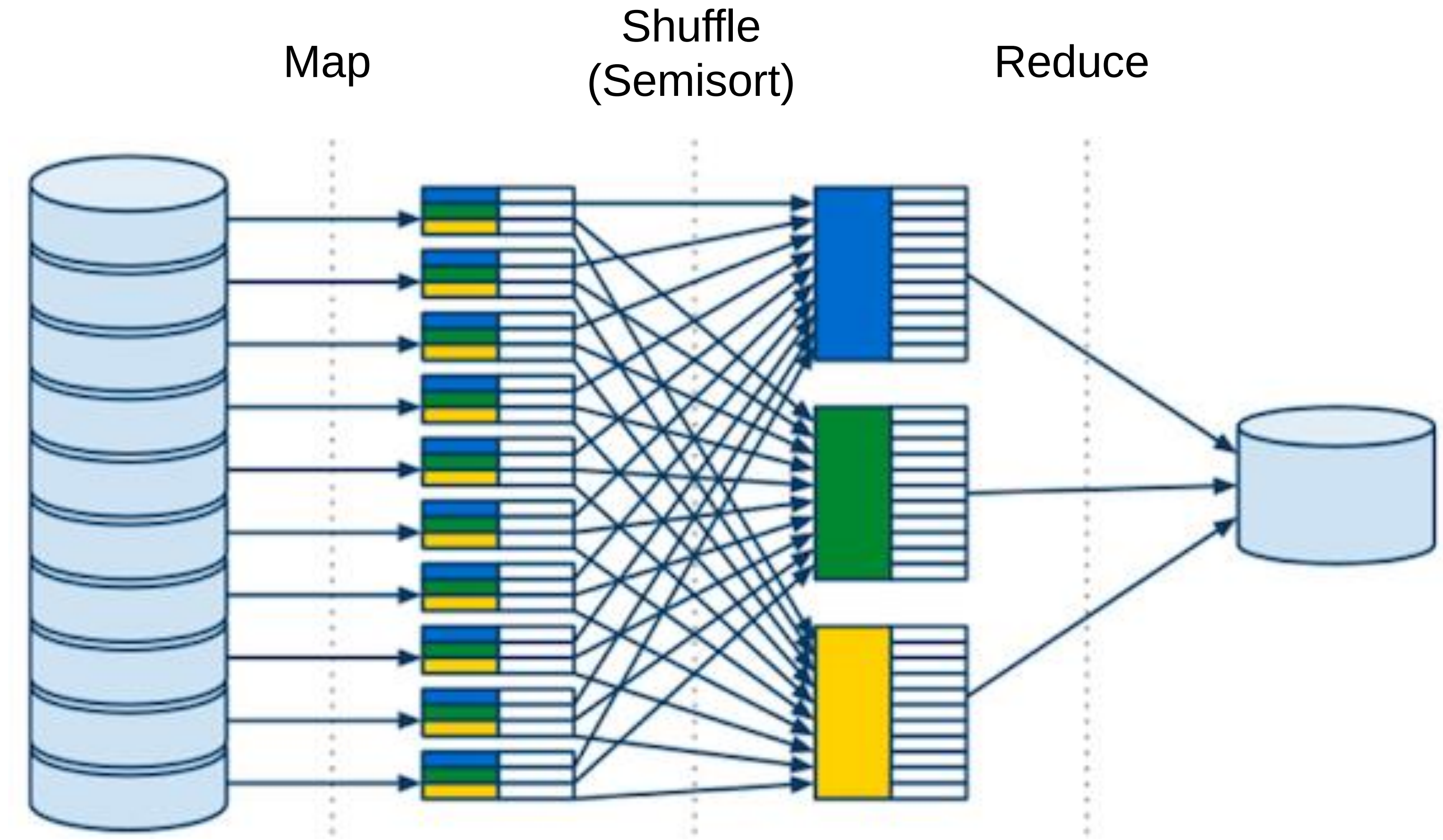
- **The simulation of PRAM model – concurrent write [Valiant 1990]**
  - Key: memory addresses
  - Value: operations

Thread	Concurrent writes
1	a[3]=71
2	a[1]=99
3	a[2]=19
4	a[3]=10
5	a[5]=50
6	a[3]=12
7	a[1]=16

Thread	Sorted operations	Result
4	a[3]=10	**a[3]=71**
1	a[3]=71	
6	a[3]=12	
5	a[5]=50	**a[5]=50**
7	a[1]=16	**a[1]=99**
2	a[1]=99	
3	a[2]=19	**a[2]=19**

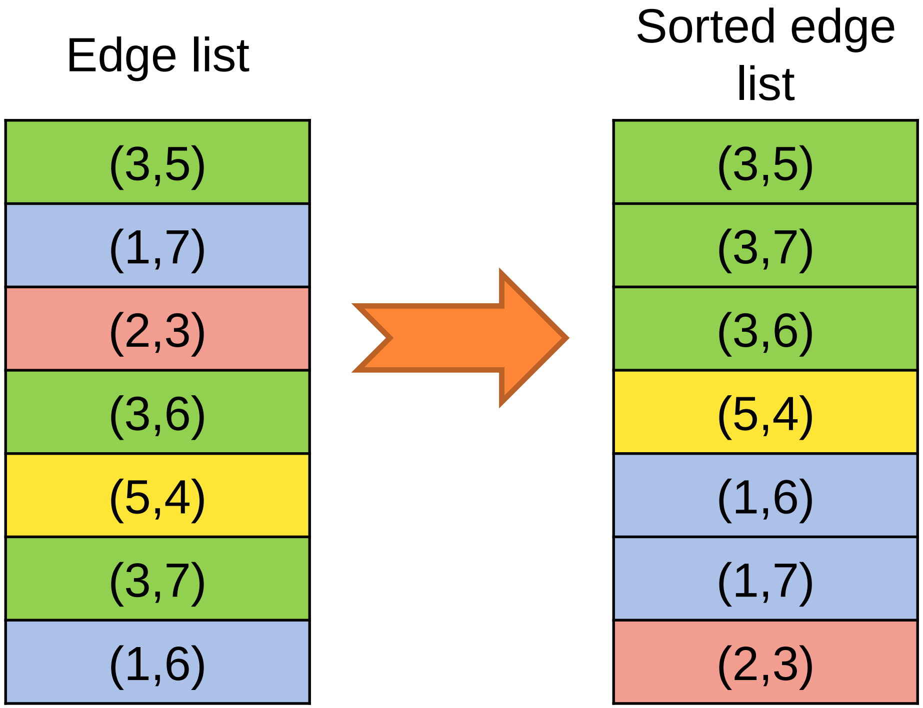
Why is parallel semisort important?

- **The map-(semisort-)reduce paradigm**

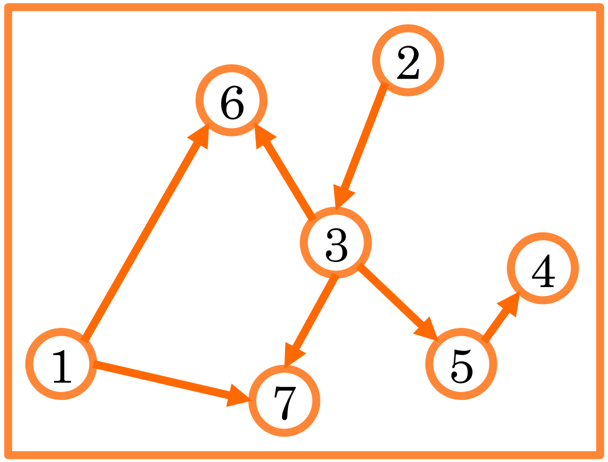


Why is parallel semisort important?

- **The map-(semisort-)reduce paradigm**
- **Generate adjacency array for a graph**



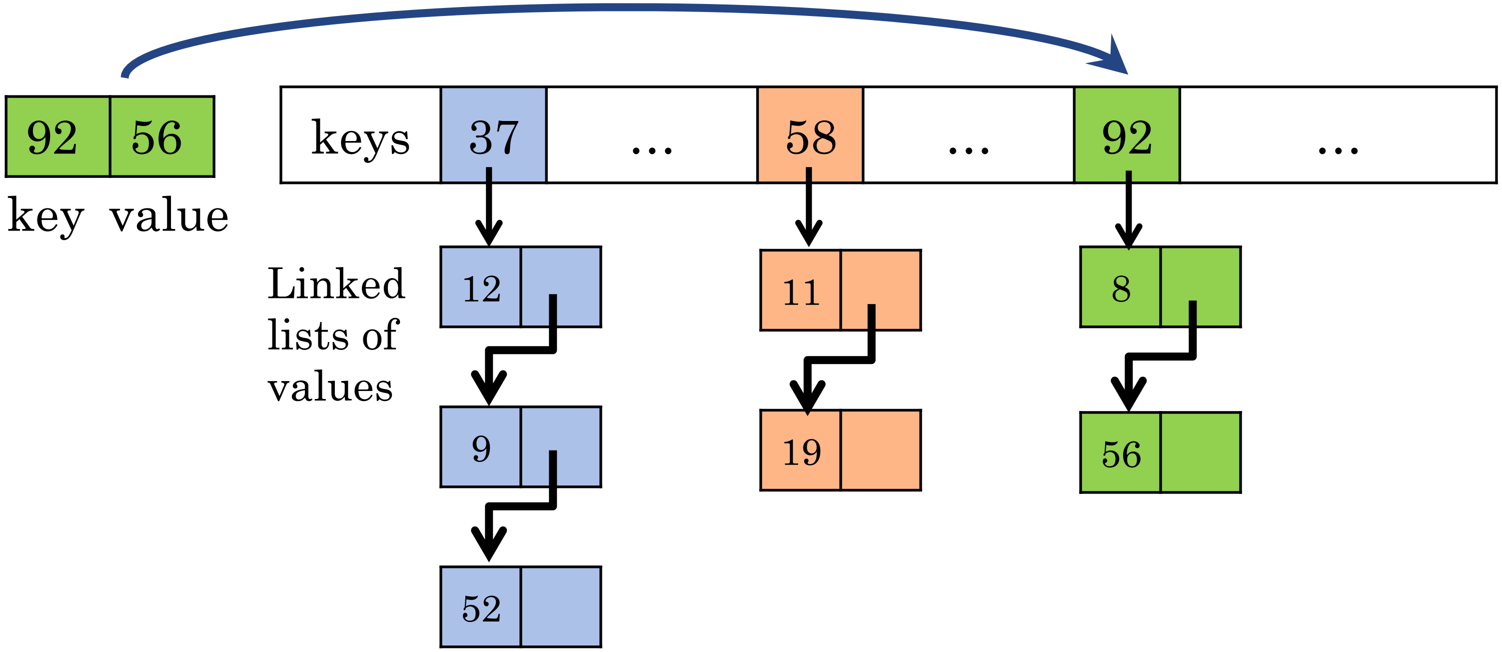
Edge list	Sorted edge list
(3,5)	(3,5)
(1,7)	(3,7)
(2,3)	(3,6)
(3,6)	(5,4)
(5,4)	(1,6)
(3,7)	(1,7)
(1,6)	(2,3)



Why is parallel semisort important?

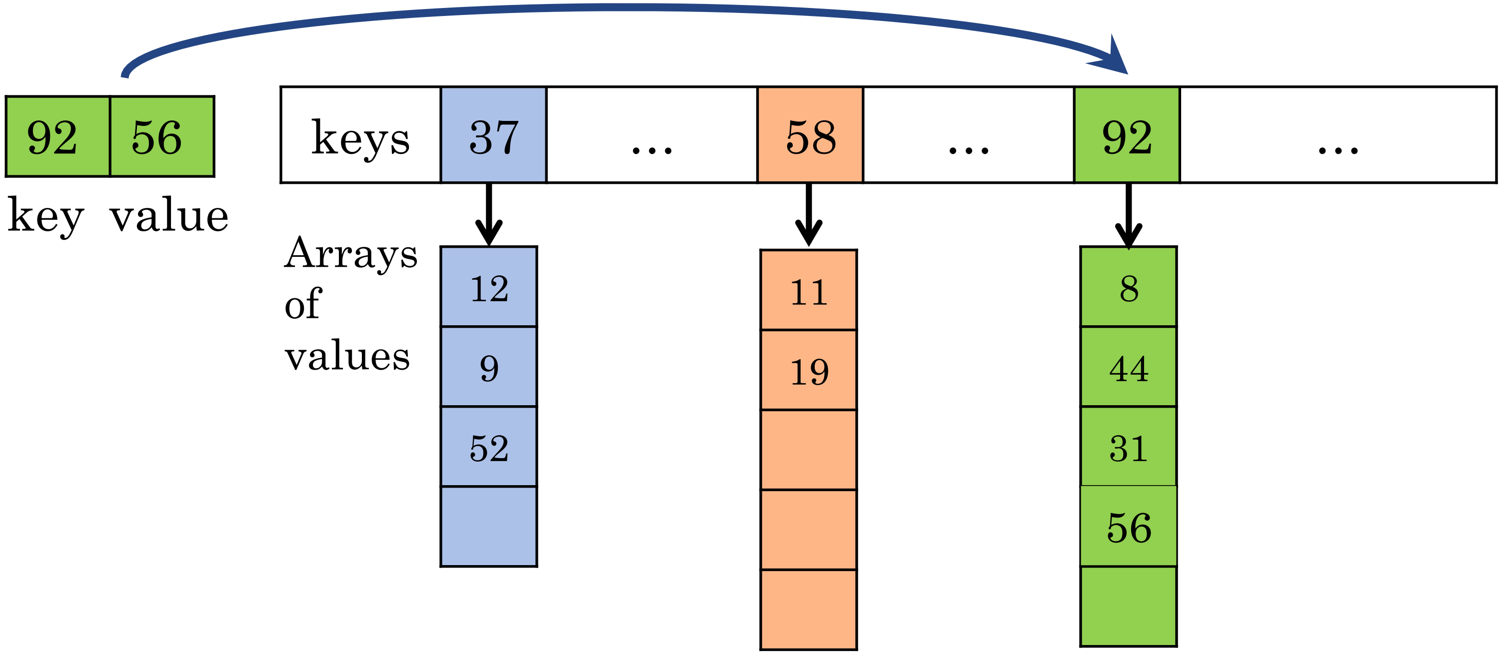
- **The map-(semisort-)reduce paradigm**
- **Generate adjacency array for a graph**
- **Other applications:**
  - In database, the relational join operation
  - Gather words that differ by a deletion in edit-distance application
  - Collect shared edges based on endpoints in Delaunay triangulation
  - Etc.

Attempts – Sequentially Hash Table With Open Addressing

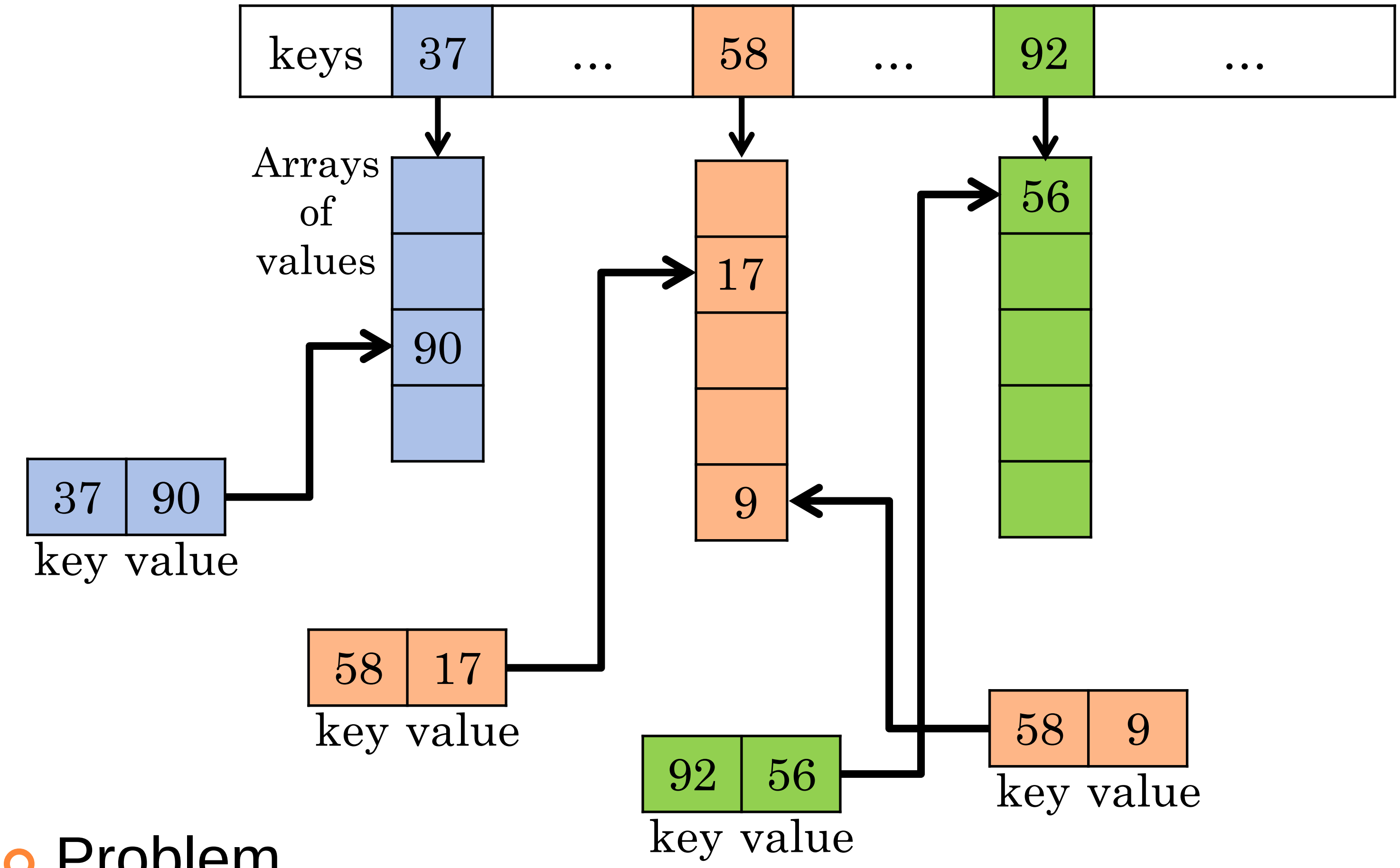


- Problem:
  - Maintaining linked lists in parallel can be hard

Attempts – Sequentially Pre-allocated array



Attempts - Parallelized Pre-allocated array



- Problem
  - Need to pre-count the number of each key

Attempts – In parallel

- Comparison-based sort
  - $O(\mathrm{n}\log n)$ work
  - Not work-efficient

☹

- Radix-sort (probably the best work-efficient option previously)
  - $O(n^{\epsilon})$ depth
  - Not highly-parallelized

☹

Attempts – In parallel

- R&R integer sort [Rajasekaran and Reif 1989]: sort n records with keys in the range $[n]$ in $O(n)$ work and $O(\log n)$ depth
  - Linear work and logarithmic depth
  - Should map keys to range $[n]$
  - Too much global data movement – practically inefficient
    - Hashing and packing – 1 time
    - Random radix sort – 1 time
    - Deterministic radix sort – 2 times

☹

How to design an efficient semisort?

- Theoretically efficient:
  - Linear work
  - Logarithmic depth
- Practically efficient:
  - Less data communication
  - Cache-friendly
- Space efficient:
  - Linear space

Our Top-Down Parallel Semisort Algorithm

Key insight: estimate key count from samples

- Once the count of each key is known, we can pre-allocate an array for each key

- The exact number is hard to compute - estimate the upper bound by **sampling**
  - Those appearing many times: we could make reasonable estimations from the sample

  - Those with few samples: hard to estimate precisely

  - Solution: Treat **“heavy”** keys and **“light”** keys differently

Our parallel semisort algorithm

- 1. Select a sample S of keys and sort it
  - Sample rate $\Theta(1/\log n)$

- 2. Partition S into **heavy keys** and **light keys**
  - **Heavy:** appears $= \Omega(\log n)$ times; will be assigned an individual bucket
  - **Light:** appears $= O(\log n)$ times. We evenly partition the hash range to $n/\log^2 n$ buckets for them

- 3. Scatter each record into its associated bucket
  - The only global data communication

- 4. Semisort light key buckets
  - Performed locally

- 5. Pack and output

Heavy vs. Light…Why?

- [Rajasekaran and Reif 1989]If the records are sampled with probability $p = 1/\log n$, and for a key i which appears a_i times in the original array, and $\boldsymbol{c_i}$ times in the sample:

  - $c_i = \Omega(\log n)$, then $a_i = \Theta(c_i \log n)$ w.h.p.
  - $c_i = O(\log n)$, then $a_i = O(\log^2 n)$ w.h.p.

  (Can be proved using Chernoff bounds)

Estimate upper bounds for the counts a_i

- Key insight: if the records are sampled with probability $p = 1/\log n$, and key i has:
  - $c_i = \Omega(\log n)$ samples, then $a_i = \Theta(c_i \log n)$ w.h.p.
  - $c_i = O(\log n)$ samples, then $a_i = O(\log^2 n)$ w.h.p.

- $u_i = c' \max(\log^2 n, c_i \log n)$
  - c' is a sufficiently large constant to provide the high probability bound

Estimate upper bounds for the counts a_i

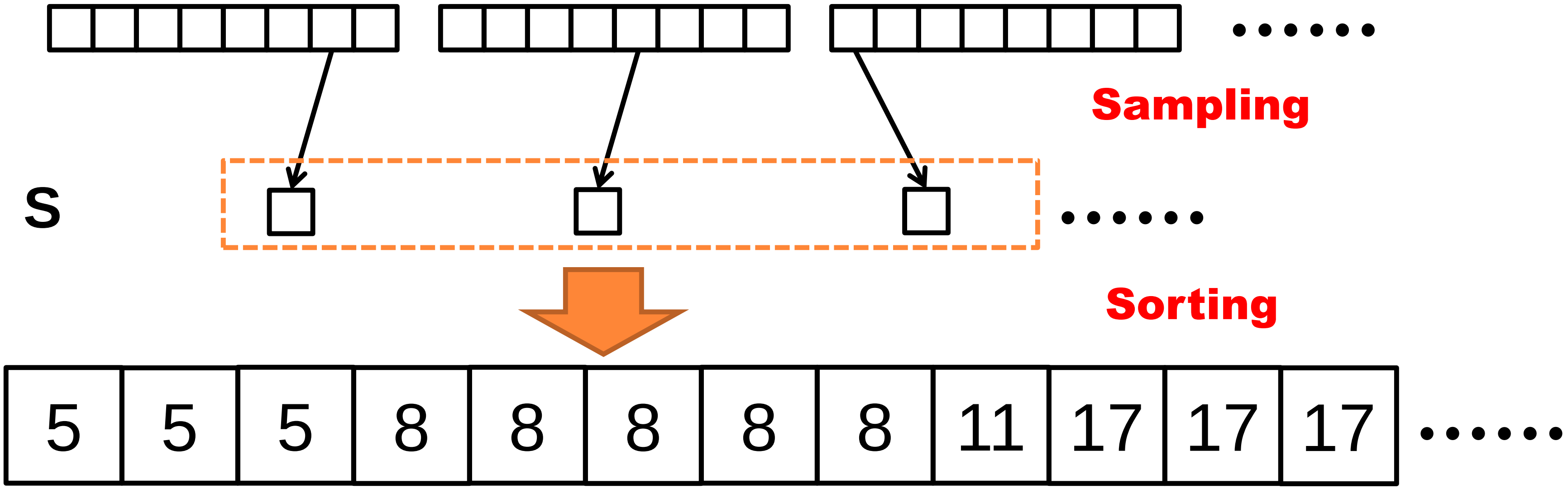
- Key insight: if the records are sampled with probability $p = 1/\log n$, and key i has:
  - $c_i = \Omega(\log n)$ samples, then $a_i = \Theta(c_i \log n)$ w.h.p.
  - $c_i = O(\log n)$ samples, then $a_i = O(\log^2 n)$ w.h.p.

- Extreme case: all samples are of the same key
  - $c_i = \frac{n}{\log n} \Rightarrow u_i = O(n)$
  - $c_i = 0 \quad \Rightarrow u_i = O(\log^2 n)$
  - Require keys to be in range $[n/\log^2 n]$

- Solution: combine light keys
  - evenly partition the hash range to $n/\log^2 n$ intervals as buckets

Phase 1: Sampling and sorting

1. Select a sample S of keys with probability $p = \Theta(1/\log n)$
2. Sort S



……

Sampling

S

……

Sorting

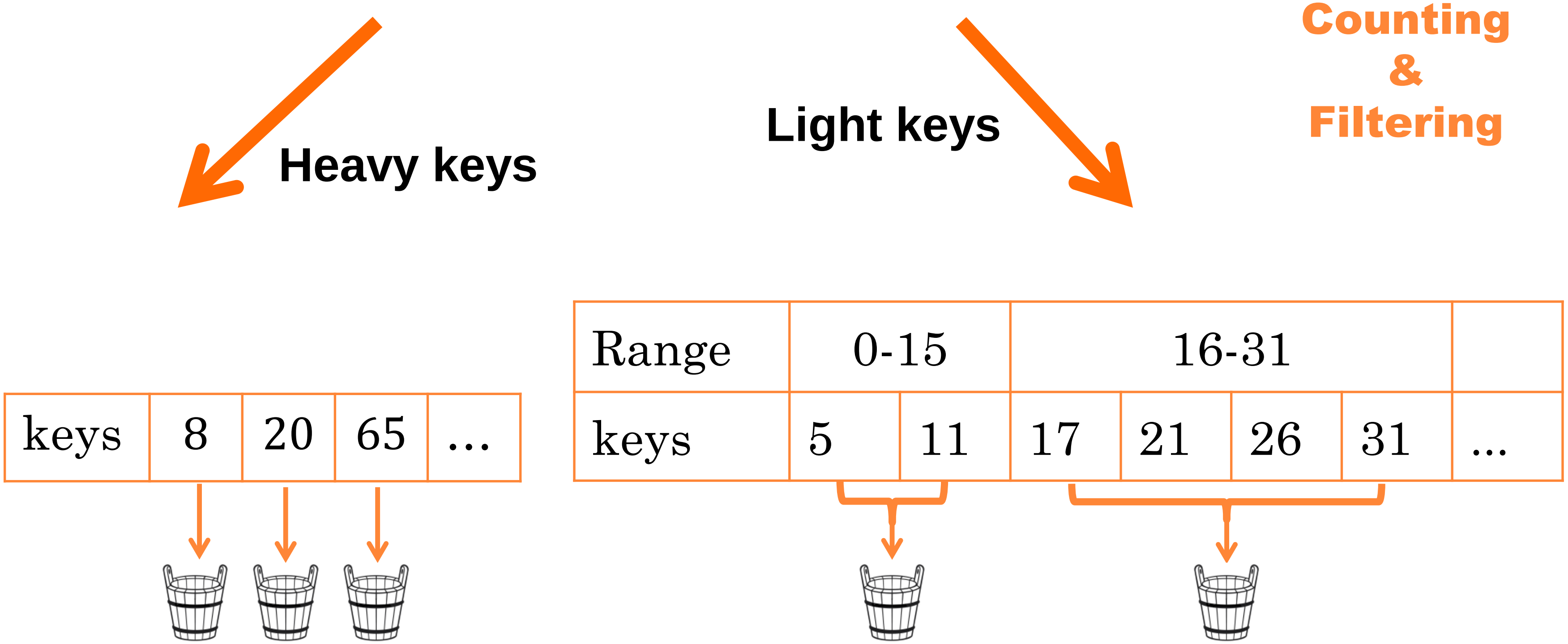
5 5 5 8 8 8 8 8 11 17 17 17 ……

(Counting)

Phase 2: Array Construction

Sorted samples:

5	5	5	8	8	8	8	8	11	17	17	17	……



Heavy keys

Light keys

Counting & Filtering

keys	8	20	65	…

Range	0-15		16-31				
keys	5	11	17	21	26	31	...

Phase 2: Array Construction

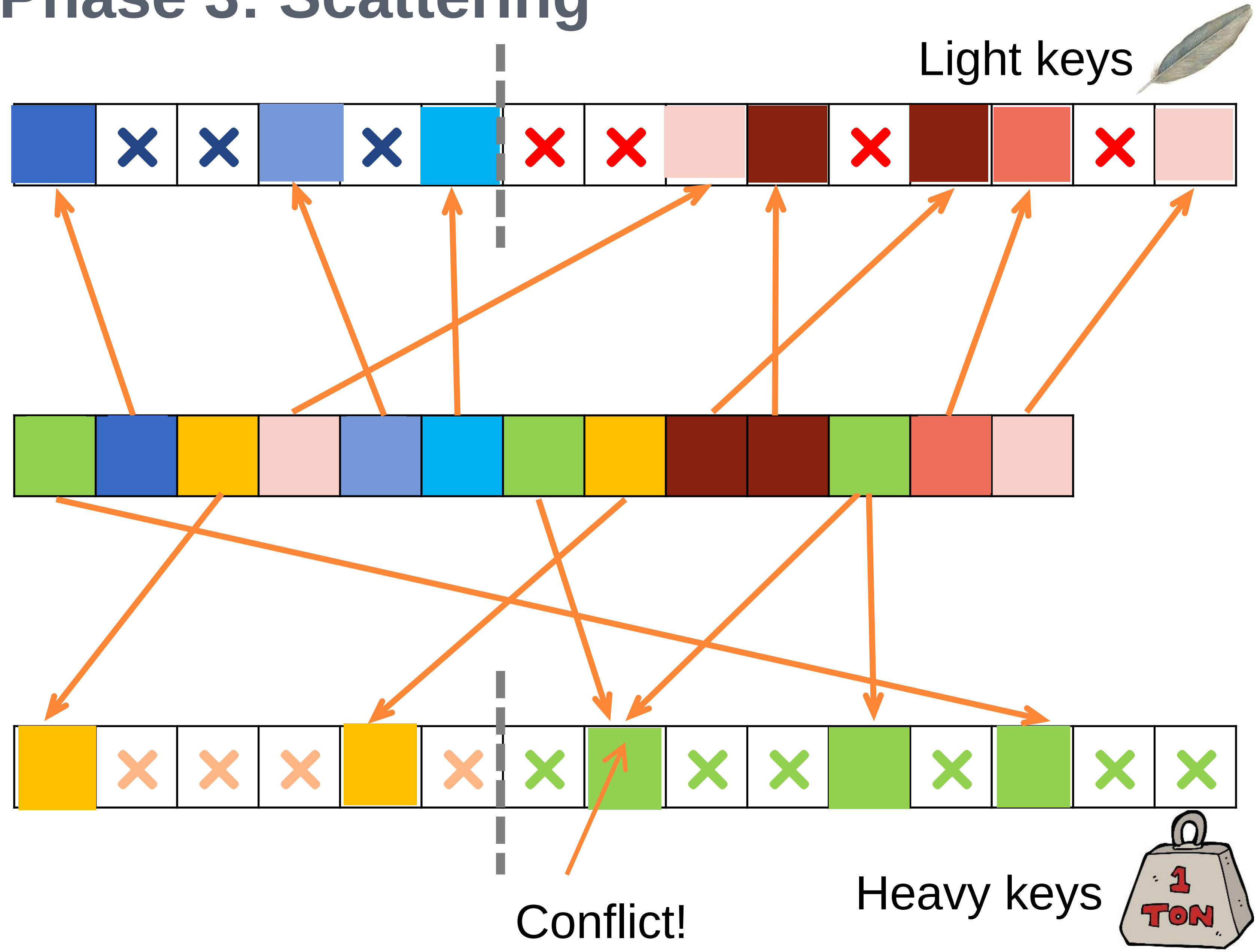
Heavy Keys				
keys	k_1	k_2	k_3	…
# samples	c_1	c_2	c_3	…

Array length	$f(c_1)$	$f(c_2)$	$f(c_3)$	…

Light Keys									
keys	k'_1	k'_2	k'_3	k'_4	k'_5	k'_6	k'_7	k'_8	k'_9	…
# samples	c'_1	c'_2	c'_3	c'_4	c'_5	c'_6	c'_7	c'_8	c'_9	…

Array length	$f(c'_1 + c'_2)$	$f(c'_3 + \cdots + c'_6)$	$f(c'_7 + c'_8 + c'_9)$	…

Phase 3: Scattering

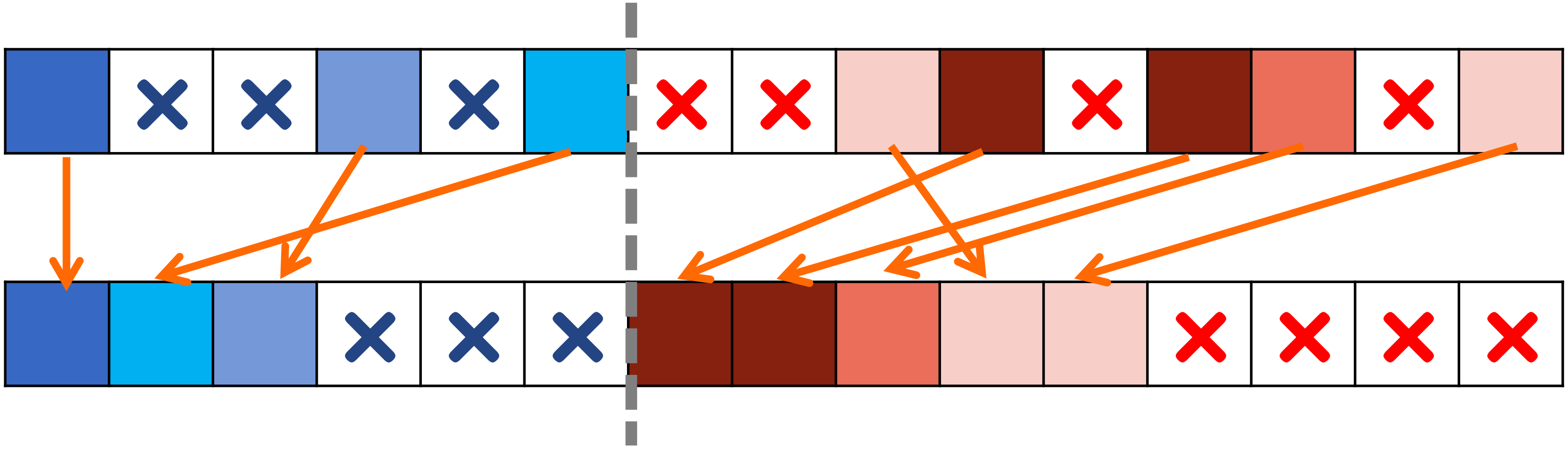


Light keys

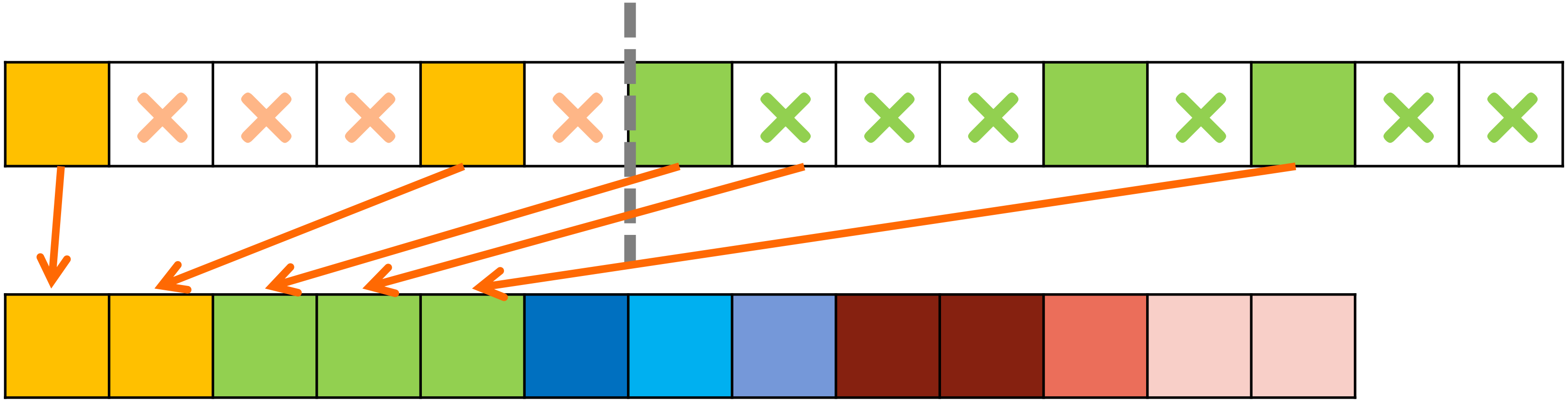
Heavy keys

Conflict!

Phase 4: Local sort



Phase 5: Packing



Size Estimation for Arrays
- High Probability

- Now consider an array that has s samples. We define the following size-estimation function:

$$\boldsymbol{f(s) = \left(s + c\, ln\, n + \sqrt{c^2\, ln^2\, n + 2sc\, ln\, n}\right)/p}$$

where $p = \Theta\left(\frac{1}{\log n}\right)$ is the sampling probability and c is a constant, to be an upper bound of the size of the array

- Lemma 1: If there are s samples of an array, the probability that number of records is more than $f(s)$ is at most n^{-c}

Size estimation for arrays
- Linear Space in Expectation

$$\boldsymbol{f(s) = \left(s + c\,ln\,n + \sqrt{c^2\,ln^2\,n + 2sc\,ln\,n}\right)/p}$$

- Lemma 1: If there are s samples of an array, the probability that number of records is more than $f(s)$ is at most n^{-c}

- Corollary 1: The probability that f gives an upper bound on all buckets is at least $1 - n^{-c+1}/\log^2 n$

- **Lemma 2: $\sum_i f(s_i) = \Theta(n)$ holds in expectation**

Comparison with R&R integer sort

- R&R algorithm:
  - Preprocessing: hashing and packing – global data movement
  - Three times bottom-up radix sort – global data movement

- Our parallel semisort:
  - Sample and sort – on a small set
  - Bucket construction – more about calculations
  - **Scatter: the only global data communication**
  - Local sort: performed locally
  - Pack: performed locally

Experiments

Experimental setup

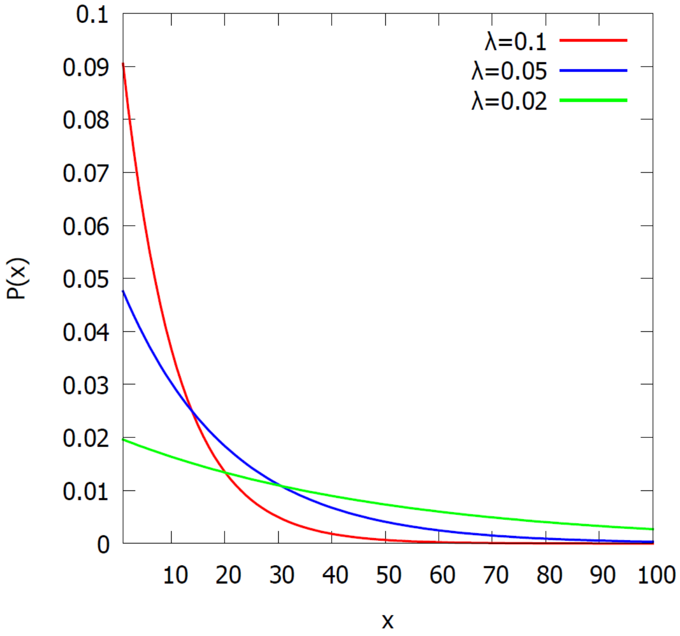
- Experiments are run on a 40-core (with 2-way HT, 40h) machine with 2.4GHz Intel 10-core E7-8879 Xeon processors, with a 1066MHz bus and 30MB L3 cache
- Our code are compiled with **`g++`** 4.8.0 with **`–O2`** flag, and parallelized with **`Cilk+`**, which is supported by **`g++`**
- We use parallel hash table with linear probing [Shun and Blelloch 2014]
- We compare to the parallel STL sort [Singler et al. 2007], parallel radix sort and sample sort from Problem Based Benchmark Suite [Shun et al. 2012]

The parallel semisort algorithm

	Notation	Value
Array length	n	$10^7 - 10^9$
Hashed key range	n^k	2^{63}
Sample rate	$p = \Theta\left(\frac{1}{\log n}\right)$	$\frac{1}{16}$
Threshold to distinguish heavy keys from light keys	$\Omega(\log n)$	16
# buckets for light key	$\Theta\left(\frac{n}{\log^2 n}\right)$	2^{16}

Input distribution

- Uniform distribution (parameter: m. range of integers are from $[m]$)
- Exponential distribution (parameter: λ. mean $1/\lambda$, variance $1/\lambda^2$)



Exponential distribution

Input distribution

- The different distributions and parameters are used to control the ratio of heavy keys.

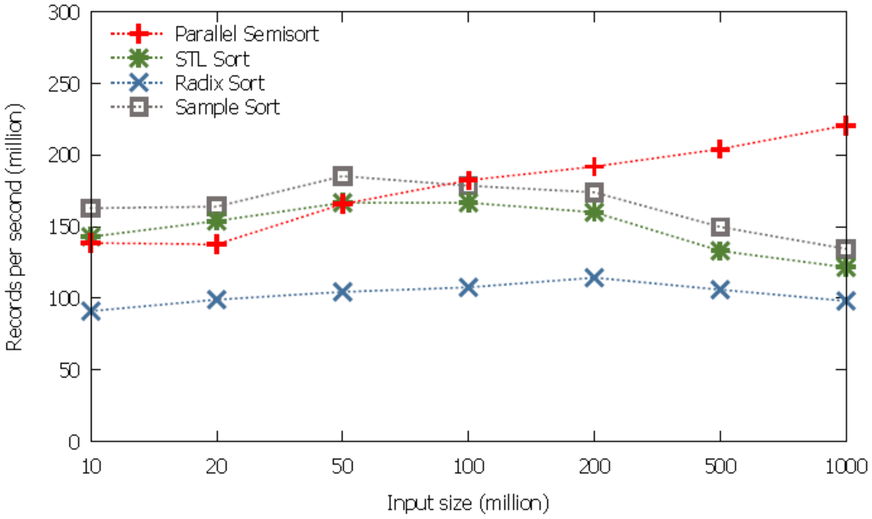
- Uniform distribution (parameter: m. range of integers are from $[m]$)
- Exponential distribution (parameter: λ. mean $1/\lambda$, variance $1/\lambda^2$)

- Two representative distributions:
  - Uniform distribution with $\mathrm{m} = n$ (0% heavy keys)
  - Exponential distribution with $\lambda = n/1000$ (70-80% heavy keys)

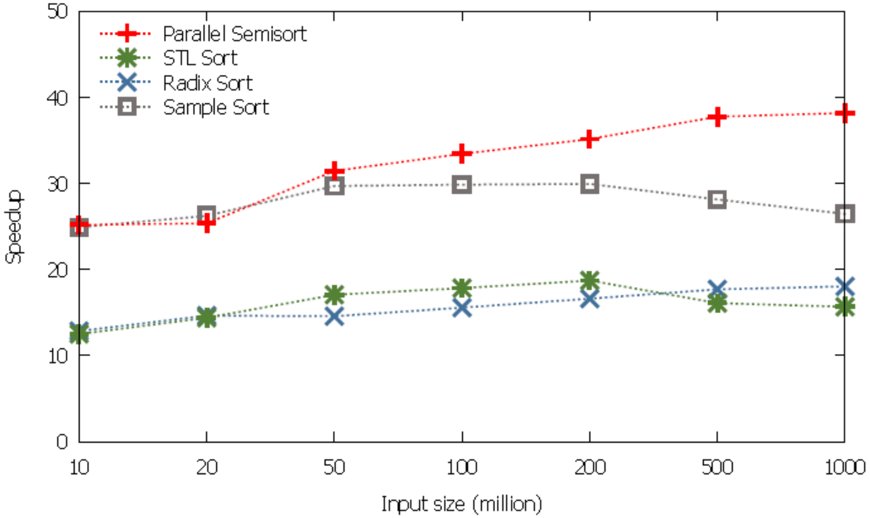
Efficiency & Scalability

Our parallel semisort outperforms STL sort, sample sort and radix sort.

- Number of threads: 40 cores with hyperthreading
- Array length: 10^8
- Distribution: exponential



Records per second

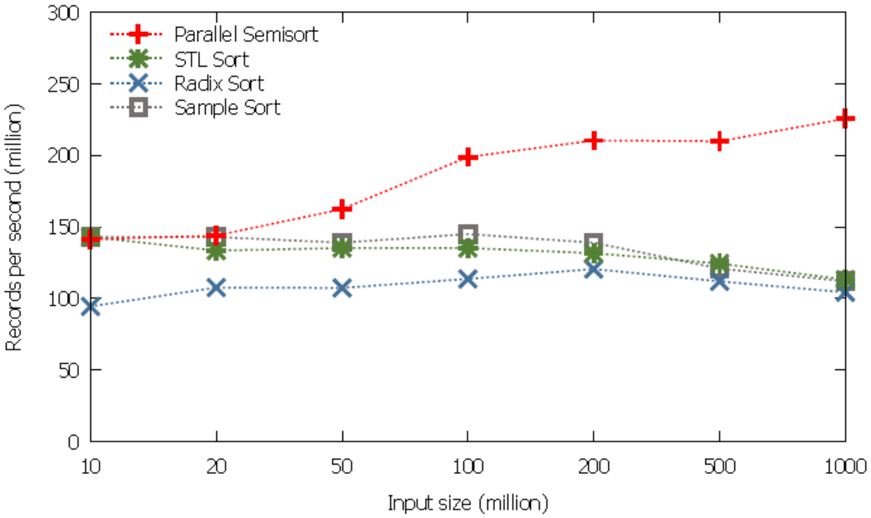


Parallel speedup

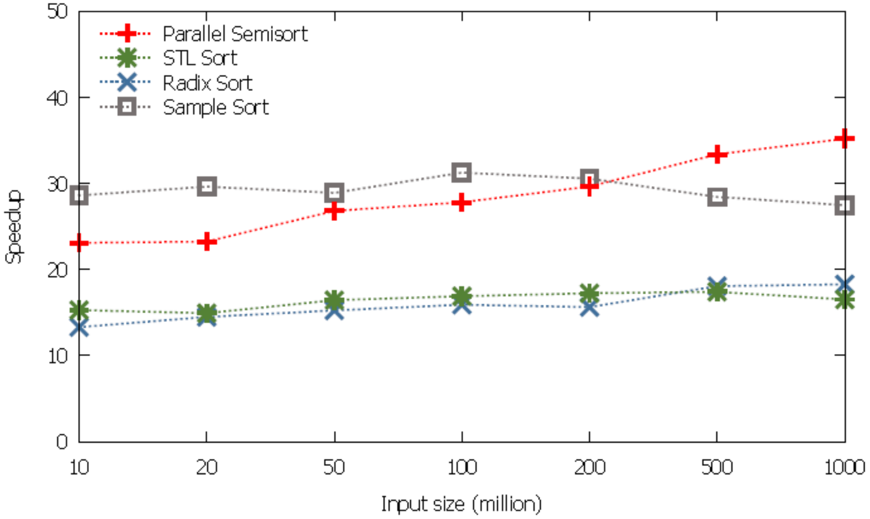
Efficiency & Scalability with input size

Our parallel semisort outperforms STL sort, sample sort and radix sort.

- Number of threads: 40 cores with hyperthreading
- Array length: 10^8
- Distribution: uniform



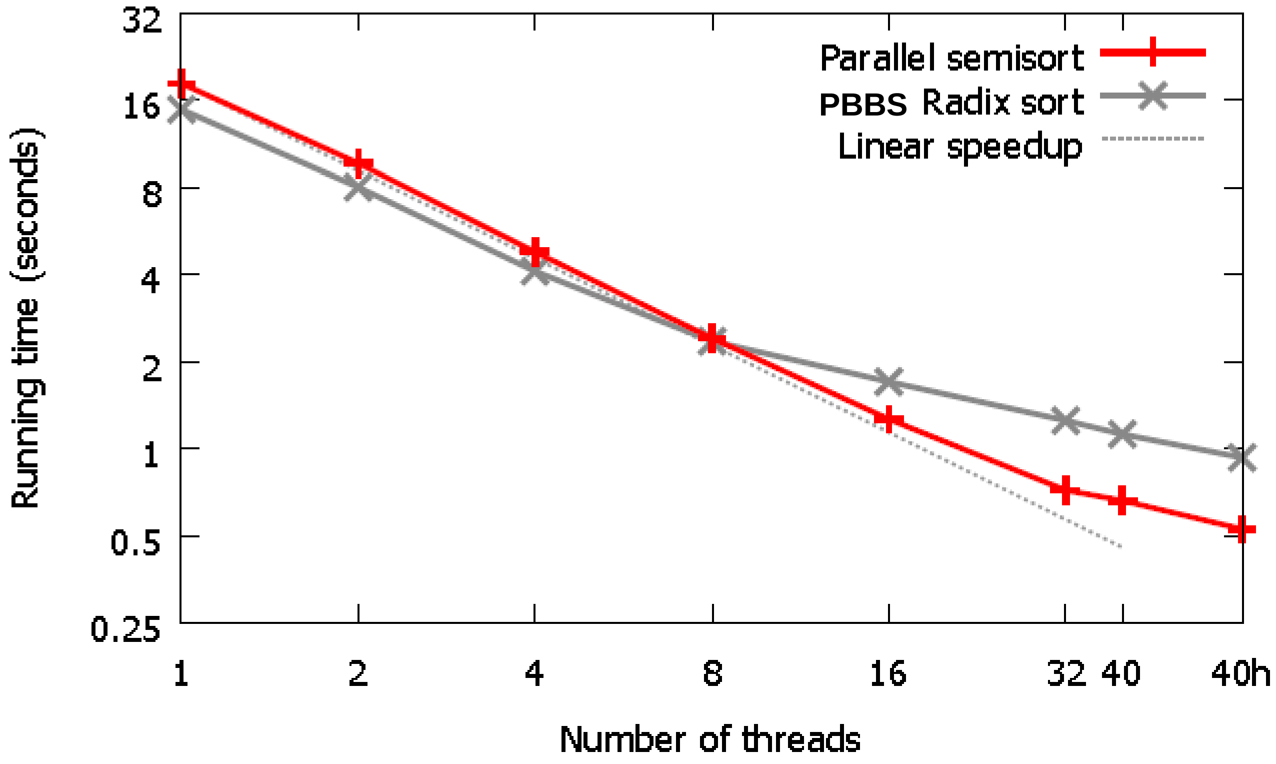
Records per second



Parallel speedup

Parallel Performance
Linear speedup

- PBBS radix sort [Shun et al 2012]

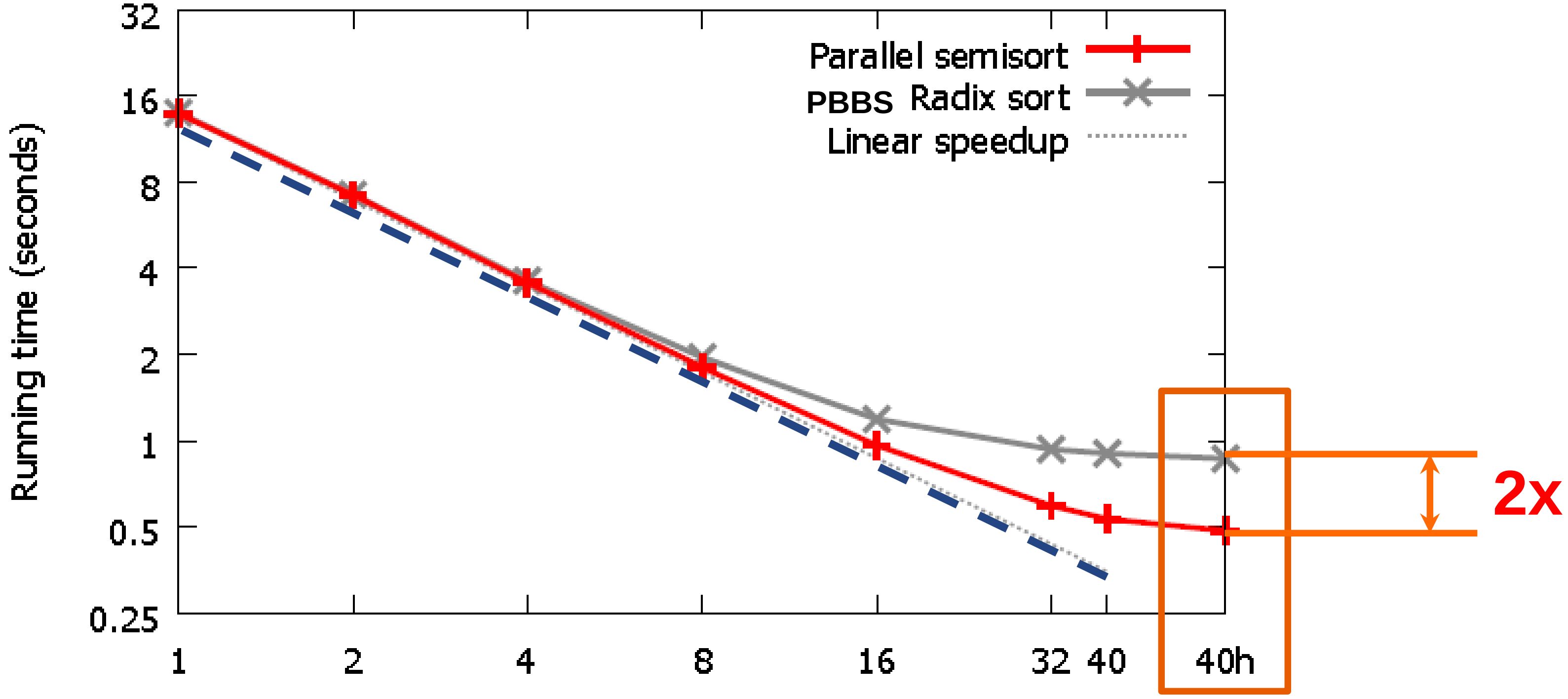


Uniform Distribution

- Radix sort proposed in [Polychroniou and Ross 2014]
  - Crashed on exponential distribution

Parallel performance
Linear speedup

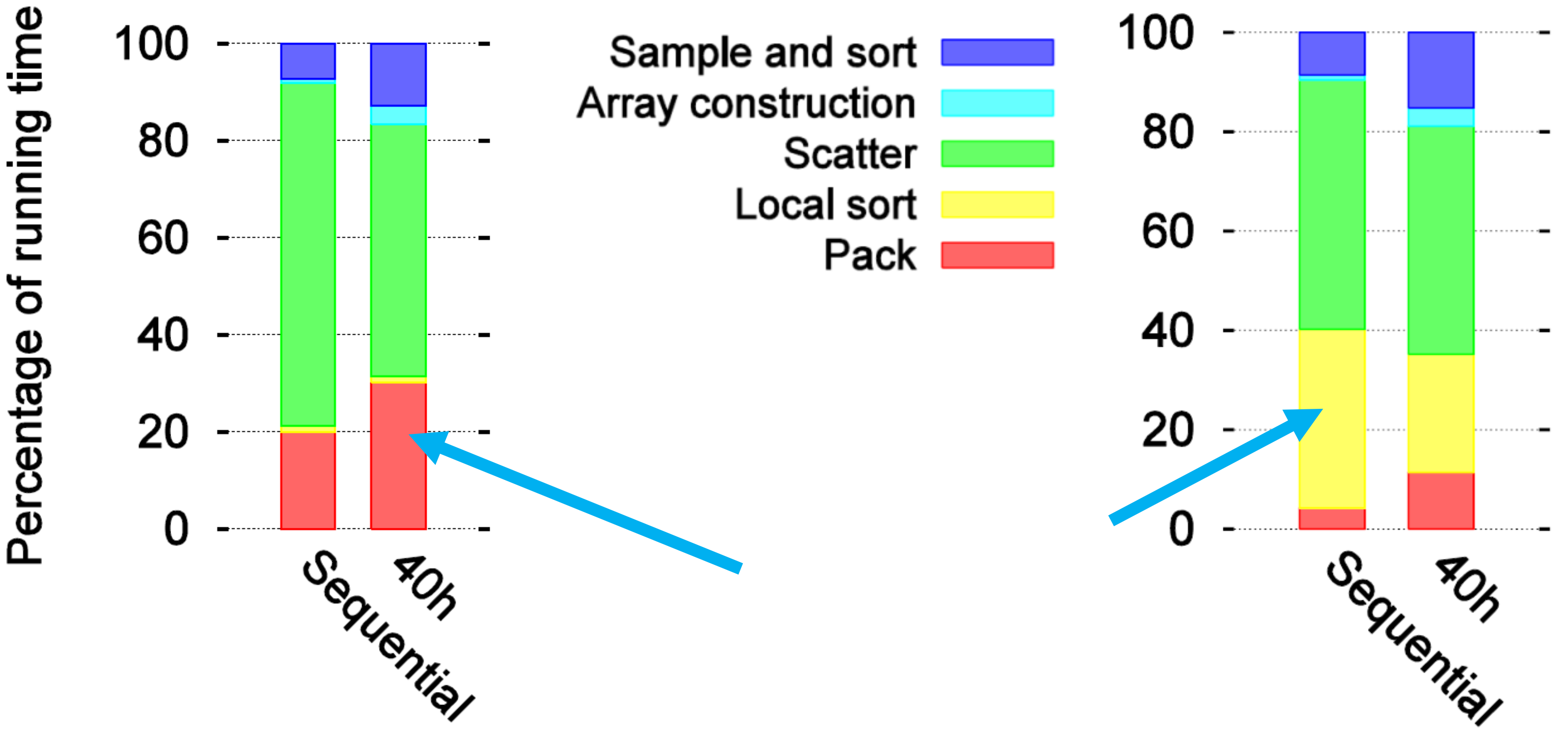
- We show the running time of our algorithm and the radix sort with varying number of threads
- The input contains 10^8 records



Exponential Distribution

(40 cores with hyperthreading)

Breakdown of running time



Exponential

Uniform

Other experiments -
The stabability

- We also have more experiments on testing the stability with different distributions
  - Three different distributions
  - 17 cases in total

- We refer you to our paper to see the details.

Conclusion

Conclusion

- We introduced a parallel algorithm for semisorting that is:
  - **Theoretically efficient**: requires linear work and space, and logarithmic depth.
  - **Practically efficient**: achieves good parallel speedup on various input distributions and input size, and outperforms a similarly-optimized radix sort and other commonly-used sorts.

Thank you.