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

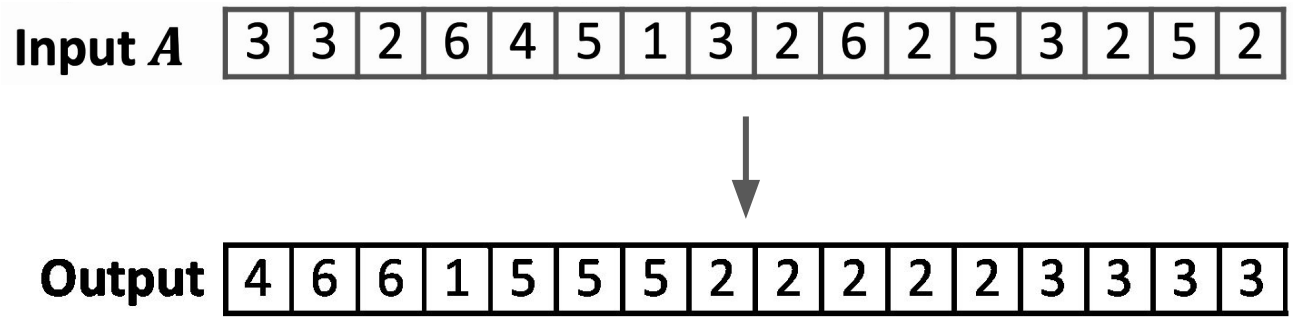
Xiaojun Dong, Yunshu Wu, Zhongqi Wang,
Laxman Dhulipala, Yan Gu, Yihan Sun

What is semisort?

Reorder an array of records so that identical keys are grouped contiguously

Keys do not need to be in sorted order

Can be solved sequentially in O(n) time with a hash table



Motivation

Example use cases of semisort and related problems:

- groupBy/Aggregation in databases
- frequency in data analytics
- shuffle in MapReduce

Theoretically efficient parallel semisort algorithm used in theoretical analysis for better bounds, but not actually used in practical implementations

Existing Algorithm - GSSB

Only existing parallel semisort algorithm at the time

- O(n) work and O(log n) span whp

Drawbacks:

- Not I/O friendly
  - Random accesses and parallel hash tables
- Interface overhead
  - Assumes records have distinct hashed keys, requiring hashing and deduplication
  - Incurs significant pre/postprocessing overhead in practice
- Not stable or internally-deterministic
  - Parallel hash tables

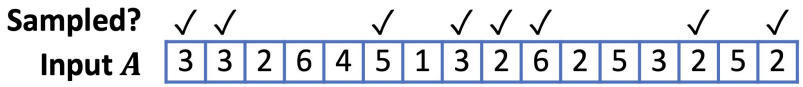
Semisort Algorithm

Builds upon GSSB

- More flexible interface
  - Accept keys of arbitrary type K
  - Collisions acceptable since compares keys directly, avoiding overhead
  - Allows for extension to histogram and collect-reduce
- Inspiration from I/O efficient parallel samplesort
  - Use auxiliary arrays to count appearance of buckets in subarrays, distribute based on counts
  - Stable and race-free

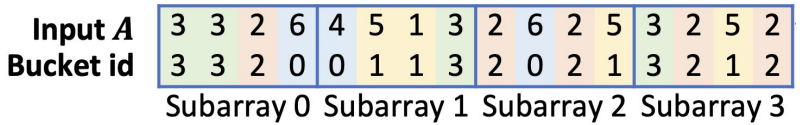
Overview

1. Sampling and Bucketing



Heavy Keys: 2, 3 (Counts ≥ 3)

2. Blocked Distributing



Count Array C	Buckets 0	1	2	3
Subarray 0	1	0	1	2
Subarray 1	1	2	0	1
Subarray 2	1	1	2	0
Subarray 3	0	1	2	1

Prefix Array X	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

3. Local Refining

Recurse on light buckets: **Output** 4 6 6 1 5 5 5 2 2 2 2 2 3 3 3 3

Notation

A	Input Array
h	Hash function
n_L	Number of light buckets
n_H	Number of heavy buckets
l	Subarray size
α	Base case threshold
n'	Problem size at current level
S	Set of samples $\lvert S\rvert = n_L \log n'$
H	Heavy table
C	Counting matrix
X	Prefix sum of C

Notation

A	Input Array
h	Hash function
$\boldsymbol{n_L}$	**Number of light buckets**
n_H	Number of heavy buckets
$\boldsymbol{l}$	**Subarray size**
α	**Base case threshold**

$\boldsymbol{n'}$	**Problem size at current level**
S	Set of samples $\lvert S\rvert = n_L \log n'$
H	Heavy table
C	Counting matrix
X	Prefix sum of C

Assumptions

- Work-span model for fork-join parallelism with binary forking
- Threads share common memory
- Two-level memory hierarchy
  - Optimal offline cache replacement policy
  - Unit cost for each cacheline load and evict

Step 1: Sampling and Bucketing

1. $S \leftarrow n_L \log n'$ uniformly sampled keys from A
2. Count occurrences of each key in S
3. Initialize heavy table H and index *id* $\leftarrow n_L$
4. **for** distinct key k in S **do** // build heavy table
5. **if** occurrences of $k \geq \log n'$ **then**
6. *H*.insert(k, *id*)
7. *id* $\leftarrow$ *id* + 1

Step 2: Blocked Distributing

1. Initialize counts C with size (n_L+n_H) x (n'/l) // #buckets x #subarrays
2. **parallel_for** $i : 0 \leq i < n'/l$ **do** // count each key in each subarray
3. **for** $j : i \cdot l \leq j < (i+1) \cdot l$ **do**
4. $id \leftarrow$ GetBucketId(key($A[j]$), H, h, n_L)
5. $C[i][id] \leftarrow C[i][id] + 1$
6. $X \leftarrow$ column-wise prefix sum of C
7. offsets $\leftarrow X[:][0]$
8. Initialize temporary array T of size n'
9. **parallel_for** $i : 0 \leq i < n'/l$ **do** // place element based on offset in X
10. **for** $j : i \cdot l \leq j < (i+1) \cdot l$ **do**
11. $id \leftarrow$ GetBucketId(key($A[j]$), H, h, n_L)
12. $T[X[i][id]] \leftarrow A[j]$
13. $X[i][id] \leftarrow X[i][id] + 1$
14. $A \leftarrow T$

Step 3: Local Refining

1. // recursively semisort light buckets
2. **parallel_for** i : 0 ≤ i < n_L **do**
3. Semisort(A[offsets[i]..A[offsets[i+1]])

Base Cases:

- semisort$_=$ uses sequential hash table with chaining
- semisort$_<$ uses standard stable comparison sort

In-Place Optimization

- Instead of copying T to A at the end of Blocked Distributing, swap the arrays
- For base case and heavy buckets, copy from T to A
  - Most cases will reach base case in two levels, so each record copied twice
- O(n) extra space

7. offsets ← $X[:][0]$
8. Initialize temporary array T of size n'
9. **parallel_for** $i : 0 \leq i < n'/l$ **do**
10. **for** $j : i \cdot l \leq j < (i+1) \cdot l$ **do**
11. $id \leftarrow$ GetBucketId(key($A[j]$), H, h, n_L)
12. $T[X[i][id]] \leftarrow A[j]$
13. $X[i][id] \leftarrow X[i][id] + 1$
14. $A \leftarrow T$

Extension to Collect-Reduce and Histogram

Collect-Reduce: compute aggregate sum of values for each key

- In Blocked Distributing, directly compute reduced values for heavy records in each subarray without distributing to corresponding buckets
- Then reduce results of all subarrays

Histogram: counts number of occurrences of each key

- Use Collect-Reduce with values of 1 for all records

Number of Recursion Levels

- $O(n/n_L)$ records in each light bucket whp
- Light bucket size shrinks by factor of $O(n_L)$ each level

→ Number of recursion levels is $\mathbf{r = O(\log_{n_L}(n/\alpha))}$ whp

Work

Total work per level across all subproblems is O(n)

- Sampling and Bucketing has O(n) samples, so O(n) work
- Blocked Distributing has O(n) to compute counts, prefix sum, and distribute

Base cases

- $\text{semisort}_=$ is O(n) total work
- $\text{semisort}_<$ is O(n log α) total work

→ Work is **O(rn)** for $\text{semisort}_=$ or **O(rn + n log α)** for $\text{semisort}_<$

Span

Sampling and Bucketing is sequential with $O(n_L \log n)$ span

Blocked Distributing step

- 2 sequential for-loops each $O(l)$ span
- Prefix sum is O(log n)

Base cases

- $\text{semisort}_=$ is O(α) span
- $\text{semisort}_<$ can be O(log n) whp in theory

→ **$O((l + n_L \log n)r + \alpha)$** for $\text{semisort}_=$ and **$O((l + n_L \log n)r + \log n)$** for $\text{semisort}_<$

I/O Cost

Assume $M/B=\Omega(n^{1/2})$, parameters $n_L=O(n^{1/4})$, $\alpha=O(n^{1/2})$, $l=O(n^{3/4})$

Number of recursive levels is $r=O(1)$ whp, so only consider top level

Sizes of C and X are $O(n^{1/2})$ and therefore fit in cache

Only additional cache misses are from A and T

- A has all serial accesses so $O(n/B)$
- T has worst case $O((n_H + n_L) \cdot (n / l) / B) = O(n^{1/2}/B)$

→ **O(n/B)** I/O cost whp, which is optimal since loading input requires O(n/B) I/Os

(M cache size, B cacheline size)

Bounds

Under same assumptions as before: parameters $n_L = O(n^{1/4})$, $\alpha = O(n^{1/2})$, $l = O(n^{3/4})$)

- Work: O(n) for $\text{semisort}_=$ and O(n log n) for $\text{semisort}_<$
- Span: $O(n^{3/4})$ for both

Experiments

Baseline Algorithms: PLSS, PLIS, IPS^{4}o, IPS^{2}Ra, GSSB, RS, PLCR

Input Distributions (default n=10^9, 64-bit keys and values):

- Uniform(μ) μ = 10^1, 10^3, 10^5, 10^7, 10^9
- Exponential(λ) λ = 1x10^{-5}, 2x10^{-5}, 5x10^{-5}, 7x10^{-5}, 1x10^{-4}
- Zipfian(s) s = 0.6, 0.8, 1, 1.2, 1.5

Average runtimes in seconds compared using geometric means

Experiment Results

In all but 4 tests, semisort$_=$ and semisort$_<$ are the best two

Overall geometric mean comparisons:

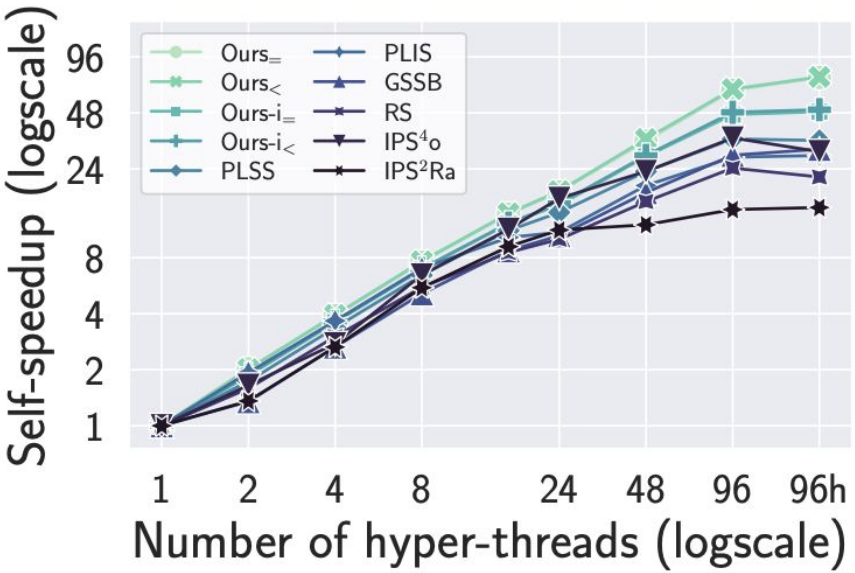
Any-type

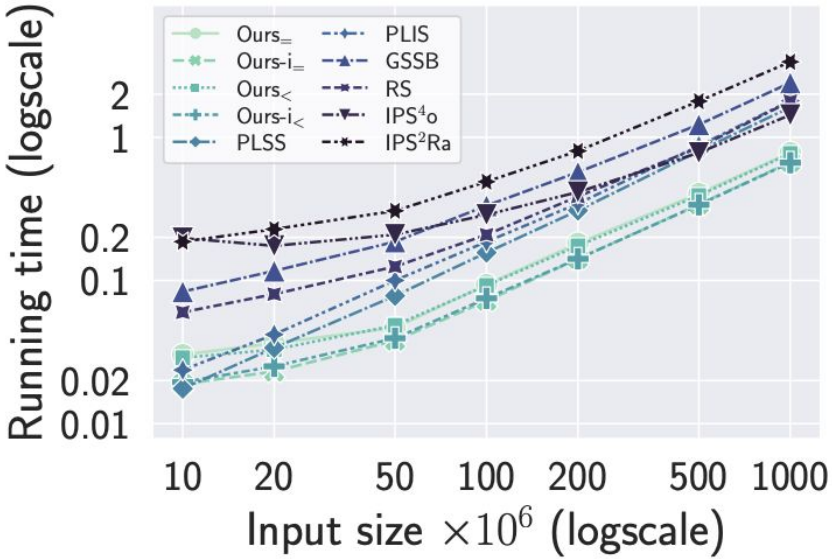
- semisort$_=$ 1.31x faster than next best
- semisort$_<$ 1.27x faster than next best

Integer-only

- semisort$_=$ 1.86x faster than next best
- semisort$_<$ 1.83x faster than next best

Experiment Results





Real-World Experiments

Graph Transpose (semisort on CSR)

- Evaluate on LiveJournal, Twitter, Cosmo50, and sd_arc
- semisort$_=$ fastest on all but LiveJournal
  - Within 15% slower than fastest on LJ
  - semisort$_<$ within 20% slower

N-Gram (semisort with first n-1 words as key and last as value)

- Evaluate on 2-gram and 3-gram from Wikipedia
- semisort$_=$ fastest on all
  - Average 15% faster than semisort$_<$ and 24% faster than best baseline

Conclusion

Strengths

- Work efficient, I/O efficient, and practical
- Thorough coverage of baselines and distributions in experiments

Weaknesses

- Needing to tune parameters (e.g. for # heavy/light buckets)
- Distribution step incurs O(n) extra space
- $O(n^{3/4})$ span compared to O(log n) of GSSB

Future Directions

- Use ideas from IPS^{4}o to reduce space usage