High Performance and Flexible Parallel Algorithms for Semisort and Related Problems

Presented by Luca Musk

Authors

Xiaojun Dong
UC Riverside

Yunshu Wu
UC Riverside

Zhongqi Wang
University of Maryland
(this is their LinkedIn photo)

Laxman Dhulipala
University of Maryland

Yan Gu
UC Riverside

Yihan Sun
UC Riverside

Semisort

Goal: Given a list of records A and a key function mapping each record to a key h, we want to produce an A' such that all records with the same key are contiguous.

[1,2,1,1,3,4,5,7,3,3,3,7,6] -> [2,1,1,1,3,3,3,4,5,7,7,6]

Can be done efficiently using hash tables and has been heavily explored in parallel algorithms literature

- However, efficient descriptions of semisort are rarely implemented

Why do we want semisort?

Core application: Collect and Reduce Algorithms

- Collect all elements of a particular key, map them to respective values, and reduce them
- Specific case: Generate histogram of your data

Generally useful in many parallel algorithms

- Many currently opt to use a parallel sort at the moment to resolve semisort, which is less work efficient

The Primary Competitor: GSSB

Designed 8 years prior

Only know implementation satisfying

- O(n) work and space
- O(logn) span
- Both with high probability

Unfortunately, has not been widely used

A Top-Down Parallel Semisort

Yan Gu
Carnegie Mellon University
yan.gu@cs.cmu.edu

Julian Shun
Carnegie Mellon University
jshun@cs.cmu.edu

Yihan Sun
Carnegie Mellon University
yihans@cs.cmu.edu

Guy E. Blelloch
Carnegie Mellon University
guyb@cs.cmu.edu

GSSB Specifics

Core Steps

1. Sample keys with rate= O(1/logn) and sort (in parallel)
2. Break heavy and light keys
3. Randomly scatter records into their corresponding buckets
4. Semisort the light buckets
5. Pack all results together

Algorithm 1 Parallel Semisort

Input: An array A with n records each containing a key.
Output: An array A' storing the records of A in semisorted order.

1: Hash each key into the range $[n^k]$ $(k > 2)$
2: Select a sample S of the hashed keys, independently with probability $p = \Theta(1/\log n)$.
3: Sort S.
4: Partition S into two sets H and L, where H contains the records with keys that appear at least $\delta = \Theta(\log n)$ times in S (heavy keys), and L contains the remaining records (light keys).
5: Create a hash table T which maps each heavy key to its associated array.
6: **Heavy keys:**
   (a) For each distinct hashed key in H allocate an appropriately-sized array for it.
   (b) Insert the records in A associated with heavy keys (which can be checked by hash table lookup in T) into their associated array.
7: **Light keys:**
   (a) Evenly partition the hash range into $\Theta(n/\log^2 n)$ buckets, and create an appropriately-sized array for each bucket by counting light keys in S.
   (b) Insert the records in A associated with light keys (which again can be checked by hash table lookup in T) into a random location in the array of its associated bucket.
   (c) Semisort each bucket.
8: Pack all of the arrays into a contiguous output array A'.

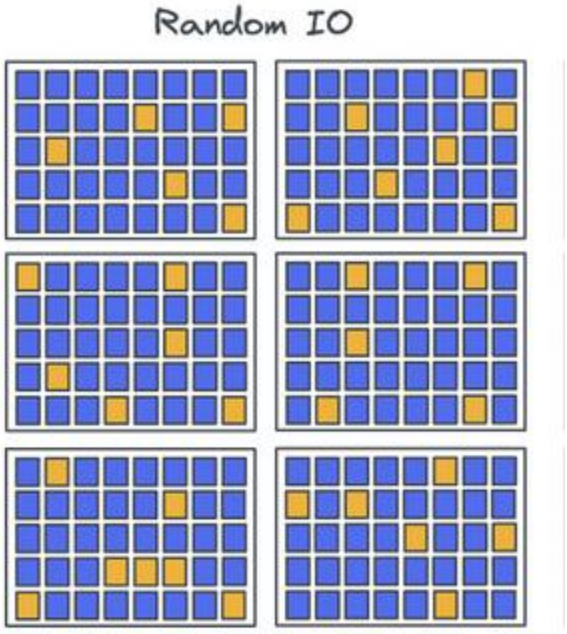
Core Issue: Random Access and Inflexibility

Scattering places each record in a random place in its corresponding array

- Not IO efficient
- Requires more space to be allocated
- Done to have placement be in parallel

Also Inflexible

- Requires all keys have a unique hash id
- Requires removing collisions before running



Parameters

nL = Light bucket count
nH = heavy bucket count
n = total problem size
n’ = subproblem size
l = subarray size

Input:	
$A[1..n]$	input array of records in universe U
K	key type of records
$key(\cdot)$	$key : U \mapsto K$ extracts the key of a record
$=_K$	(or =) equality test on keys
$<_K$	(or <) less-than test on keys
$h(\cdot)$	user hash function; $h : K \mapsto [0, n^\kappa]$
Tunable Parameters:	
l	subarray size
α	base case threshold
$n_L = 2^b$	number of light buckets
Other notations used in the algorithm and description:	
n'	problem size of the current recursion
S	the set of samples. $\lvert S\rvert = n_L \log n$
n_H	number of heavy buckets, $n_H = O(n_L)$
H	heavy table; Maps heavy keys to bucket ids
C	counting matrix
X	(column-major) prefix sum of C

Table 1: Notations and parameters used in our algorithms.

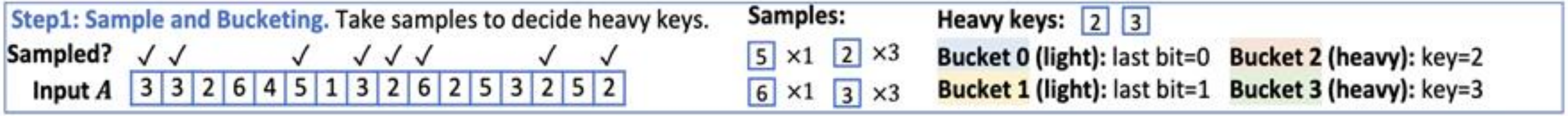
The Paper’s Approach: Sampling and Bucketing

1. Sample nLlogn’ keys
    a. nL is a parameter, not simply O(n/log^2n)
        i. Enables tuning, paper found 2^10 was best
2. Setup light buckets
    a. If key appears less than logn times
    b. Key k maps to bucket h(k) mod nL
3. Setup heavy keys
    a. Key appears logn times or more
    b. Map keys to their bucket id in H for later
4. Now have nL + nH buckets

Algorithm 1: The Semisort Algorithm

Input: The input array A, a user hash function h, and a comparison function COMP (= or <). The original (top-level) input size is n, and the current subproblem size is n'.
Output: The semisort result in A (in-place)
Parameters: $n_L = 2^b$: number of light buckets.
α: base case threshold.
l: subarray size.

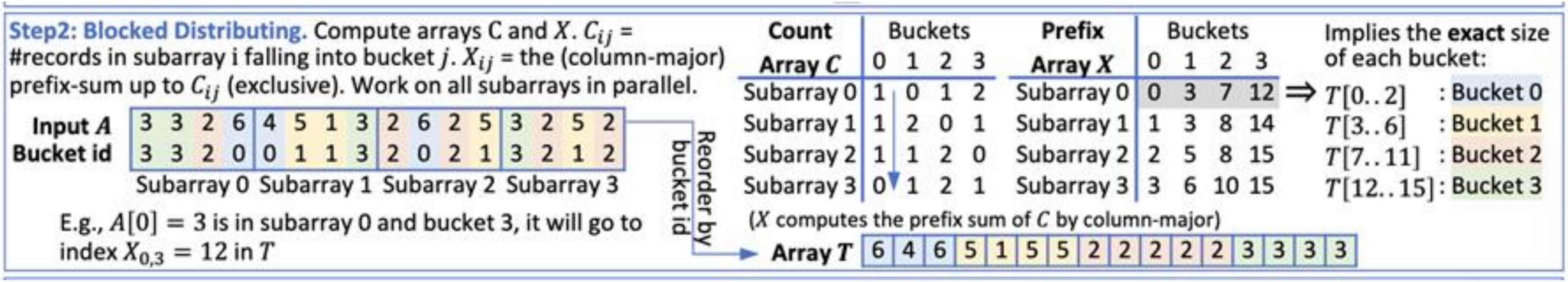
```
if |A| < α then return BaseCase(A, h, COMP)            // Base cases
Sampling and Bucketing:
S ← n_L log n' sampled keys from A
Count the occurrences of each key in S
Initialize the heavy table H
id ← n_L
// This for-loop can also be performed in parallel theoretically
for each distinct key k ∈ S do
    if the occurrences of k in S is at least log n then
        H.insert(k, id)        // Assign bucket id i to heavy key k
        id ← id + 1
n_H ← number of distinct keys in H
```



The Paper's Approach: Blocked Distributing

1. Partition n' into l n'/l sub arrays
2. Count occurrences of bucket C for each subarray
3. Initialize an array T of size n'
4. Compute offsets per subarray per bucket X
5. Move each record to its corresponding bucket in T

```
n_H ← number of distinct keys in H
Blocked Distributing:
Initializing matrix C[][] with size (n_L + n_H) × (n'/l)
parallel_for i : 0 ≤ i < n'/l do                         // For each subarray
    for j : i·l ≤ j < (i+1)·l do
        id ← GetBucketId(key(A[j]), H, h, n_L)
        // C[i][id]: #records falling into bucket id in subarray i
        C[i][id] ← C[i][id] + 1
Initialize T of size n'
// X[i][j]: offset in T for record in subarray i going to bucket j
Compute X[i][j] ← Σ_{j'<j or (j'=j, i'<i)} C[i'][j']
parallel_for i : 0 ≤ i ≤ n_L + n_H do
    offsets[i] ← X[i][0]
parallel_for i : 0 ≤ i < n'/l do                         // For each subarray
    for j : i·l ≤ j < (i+1)·l do
        id ← GetBucketId(key(A[j]), H, h, n_L)
        T[X[i][id]] ← A[j]
        X[i][id] ← X[i][id] + 1
A ← T                                    // Avoided in implementation, see Sec. 3.4
```



The Paper’s Approach: Local Refining

1. In last step, recursively semisort light buckets in parallel
2. Base case is decided by parameter α
    a. Tunable, aiming to ensure our data fits in the cache
    b. Semisort=: Semisort using hash tables with chaining for stability
    c. Semisort<: Uses a standard comparison sort
3. As an added optimization, one can recurse on T and store the result in A

```
A ← T                                   // Avoided in implementation, see Sec. 3.4
Local Refining:
parallel_for i : 0 ≤ i < n_L do         // Only for light buckets
  | SEMISORT(A[offsets[i]..offsets[i + 1]], h, COMP)
return A

Function GETBUCKETID(k, H, h, n_L)
  | if k is found in H then return the heavy id of k in H
  | else return h(k) mod n_L            // h(·) is the hash function
```

GetBucketId is more relevant for the last page but enables one to quickly get the bucket id for a key

Analysis: Work

THEOREM 3.2. *The work of semisort$_=$ is $O(rn)$ whp. The work of semisort$_<$ is $O(rn + n\log\alpha)$ whp.*

- O(n) work in sampling and bucketing.
- C and X have O(n) size if nL < l, and distributing them also takes O(n) work.
- Each recursive step breaks into nL O(n/nL) subproblems, and so does O(n) work.
- We therefore have O(rn) work before base case.
  - For semisort=, the base case is O(n) work.
  - For semisort<, the base case is O(nlogα).

Analysis: Span

THEOREM 3.3. *The span of semisort$_=$ is* $O((l+n_L \log n)r+\alpha)$ whp. *The span of semisort$_<$ is* $O((l+n_L \log n)r+\log n)$ whp.

- Bucketing and Sampling is sequential (O(nLlogn) span, though can be parallel)
- Blocked distribution has O(l) span due to parallel for loops and prefix sum incurs O(logn) span.
- O((l + nLlogn)r) span before base cases
  - Semisort=: Sequential hash tables, so O(α)
  - Semisort<: O(logn) span due to sort (though this implementation is sequential)

The Algorithm's Successes

Flexible interface

- Doesn't require a collision-free hash to get bucket ids since heavy key bucket ids are determined by the key themselves, not h(k).

Lower space usage

- No need to lower load factor for random scattering, so usage is n
- IO efficient by using smaller nL so C and X fit in cache

Stable and race free

- GSSB has races due to parallel hash tables and is unstable, preventing some collect reduce functions

Experimental Results: The Competitors

GSSB

Samplesort

- Quicksort with many pivots
- ParlayLib and IPS4o

Integersorts

- Compare to parallel integer sorts in ParlayLib, RegionSort, and IPS4Ra

Results: Comparisons

The paper’s semisort is quite successful

- Semisort< and semisort= are typically the top 2 and have the least cache misses
- On average, other algorithms are at least 25% slower

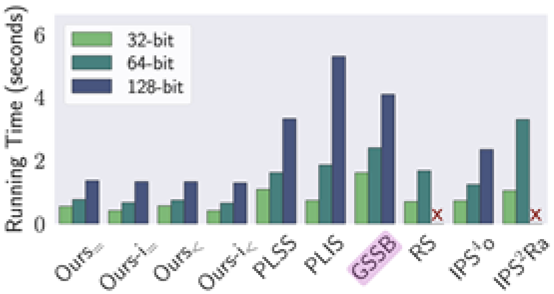
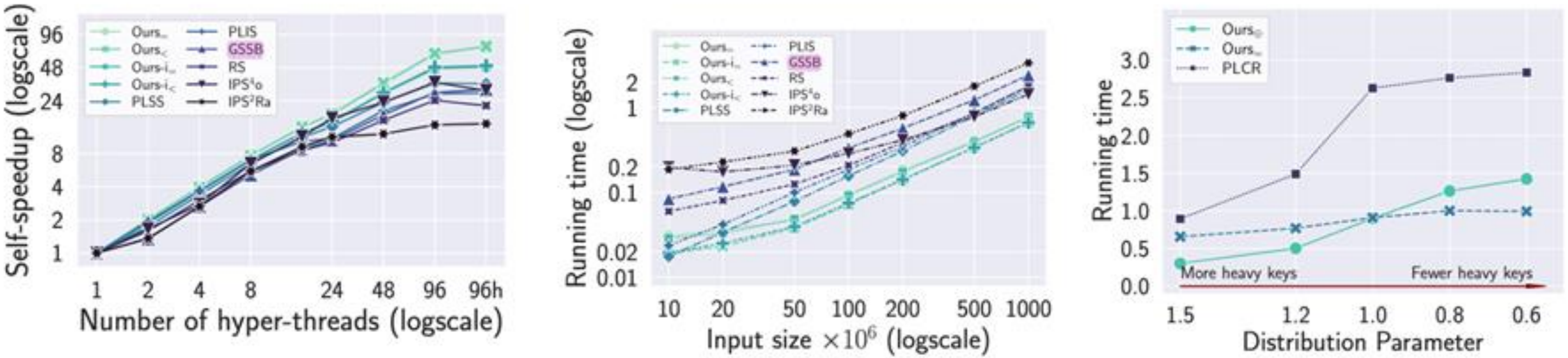


Figure 4: Running time of our semisort implementations and other implementations with different key-lengths on Zipfian-1.2. $n = 10^9$. We put crosses on RS and IPS^{2}Ra because they do not support 128-bit keys.



(a) (b) (c)

Results Per Distribution

		Any input type				Integer input type					
		$\text{Ours}_=$	$\text{Ours}_<$	PLSS	IPS^4o	$\text{Ours-i}_=$	$\text{Ours-i}_<$	PLIS	GSSB	RS	IPS^2Ra
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

Results: Applications

Graph Transpose

- Equivalent to semisorting CSR
- Semisort= is the fastest on average, and is at most 15% slower

N-grams

- Processing groups of n consecutive words
- Semisort= is fastest

Strengths, Weaknesses, and Next Steps

Strong analysis and results

- Flexible and practical design
- Good results on simulated and real-world inputs

Some minor weaknesses

- Could better explain how certain design choices resolve issues in GSSB
- Hard to compare the actual strength with so few semisort specific implementations
- Tunable but not the best parallelism (O(n/l) versus GSSB's O(n/logn))

Future Work

- Include changes from IPS4o to make the algorithm further in place.
- Experiment with a non-cache aware implementation with a simpler base case

Discussion Questions

1. Where else do we see potential applications for semisort or collect-reduce?
2. Are the comparisons with pure sorting algorithms appropriate here?
3. Do you see this algorithm as likely to be implemented and used in practical applications?