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

Xiaojun Dong
University of California, Riverside
xdong038@ucr.edu

Yunshu Wu
University of California, Riverside
ywu380@ucr.edu

Zhongqi Wang
University of Maryland, College Park
zqwang@umd.edu

Laxman Dhulipala
University of Maryland, College Park
laxman@umd.edu

Yan Gu
University of California, Riverside
ygu@cs.ucr.edu

Yihan Sun
University of California, Riverside
yihans@cs.ucr.edu

Abstract

Semisort is a fundamental algorithmic primitive widely used in the design and analysis of efficient parallel algorithms. It takes input as an array of records and a function extracting a *key* per record, and reorders them so that records with equal keys are contiguous. Since many applications only require collecting equal values, but not fully sorting the input, semisort is broadly applicable, e.g., in string algorithms, graph analytics, and geometry processing, among many other domains. However, despite dozens of recent papers that use semisort in their theoretical analysis and the existence of an asymptotically optimal parallel semisort algorithm, most implementations of these parallel algorithms choose to implement semisort by using comparison or integer sorting in practice, due to potential performance issues in existing semisort implementations.

In this paper, we revisit the semisort problem, with the goal of achieving a high-performance parallel semisort implementation with a flexible interface. Our approach can easily be extended to two related problems, *histogram* and *collect-reduce*. Our algorithms achieve strong speedups in practice, and importantly, outperform state-of-the-art parallel sorting and semisorting methods for almost all settings we tested, with varying input sizes, distribution, and key types. On average (geometric means), our semisort implementation is at least 1.27× faster the *best of the tested baselines*. We also test two important applications with real-world data, and show that our algorithms improve the performance (up to 2.13×) over existing approaches. We believe that many other parallel algorithm implementations can be accelerated using our results.

CCS Concepts

• **Theory of computation → Parallel algorithms**; **Shared memory algorithms**; **Sorting and searching**.

Keywords

Semisort, Collect-reduce, Histogram, Sorting, Group-by, Parallel Algorithms, Shared-Memory Parallelism

ACM Reference Format:
Xiaojun Dong, Yunshu Wu, Zhongqi Wang, Laxman Dhulipala, Yan Gu, and Yihan Sun. 2023. High-Performance and Flexible Parallel Algorithms for Semisort and Related Problems. In *Proceedings of the 35th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '23), June 17–19, 2023, Orlando, FL, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3558481.3591071

SPAA '23, June 17–19, 2023, Orlando, FL, USA
© 2023 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-9545-8/23/06.
https://doi.org/10.1145/3558481.3591071

1 Introduction

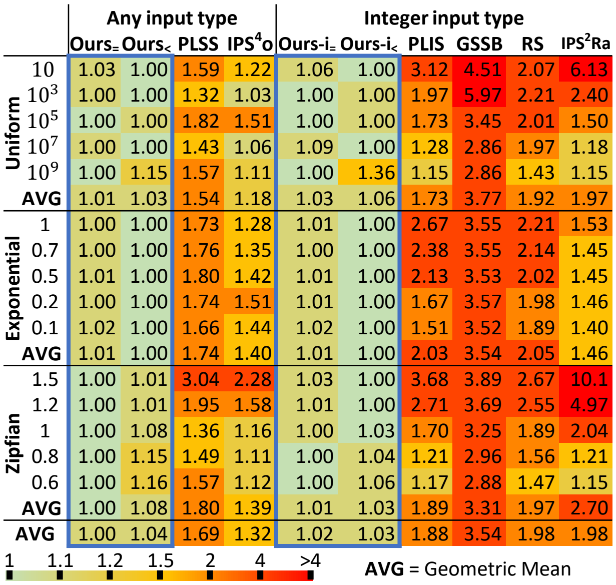
The ***semisort*** problem takes as input an array of records with associated keys, and returns a reordered array such that all records with identical keys are contiguous. Importantly, the problem does not require all keys to appear in sorted order in the output, nor all records with distinct keys to be sorted. Several other important and widely-applicable problems are closely related to semisort, such as the ***histogram*** problem that counts the number of occurrences of each key, and the more general ***collect-reduce*** problem that computes the aggregate "sum" for each key, based on all the records. The "sum" function can be defined based on any associative (sometimes also commutative) combine function (e.g., addition or maximum). Semisort, histogram, and collect-reduce are all widely used in different areas, but are often referred to using different names, e.g., `groupBy/aggregation` in databases [36, 58], `frequency` in data analytics applications, `reduceByKey/groupByKey` in RDD in Spark [79], the `shuffle` step in the MapReduce paradigm [27], and others [49]. As an example of the applicability of these problems, consider a database of sales receipts keeping the information of each sold lineitem. Useful operations to analyze trends in this data include quickly gathering lineitems from the same branch together (semisort), counting the number of sold items in each month (histogram), and obtaining the total sale of lineitems of each brand (collect-reduce).

Semisorting was first studied as a theoretical problem by Valiant to efficiently simulate parallel machine models (e.g., the PRAM) with other models [71]. Sequentially, it is easy to semisort in $O(n)$ time using a hash table, and theoretically-efficient parallel algorithms are also known [48]. Today, in contrast to its initial development as a theoretical tool for machine simulations, semisort is widely used in the design and analysis of efficient and practical parallel algorithms, for example for graph analytics [1, 2, 6, 14, 20, 22, 28, 30–32, 34, 35, 37, 38, 55, 56, 60, 63–65, 70], geometry problems [19, 44, 62, 73, 75], sequence algorithms and many others [13, 17, 41, 46, 51, 52, 68, 69, 78]. However, there is a *disconnect between theory and practice* in these parallel applications. In all of the above-mentioned papers, semisort is only used in theoretical analysis to obtain better bounds by the theoretically-efficient parallel semisort algorithm [48], but is not used in practical implementations of these algorithms. In particular, for the papers that implement and evaluate their parallel algorithms, a comparison sort (usually samplesort in [9, 13, 18]) is used. Although semisorting is asymptotically simpler than sorting, semisorting is avoided in practice in favor of sorting the data.

The only known parallel semisort algorithm and implementation is by Gu et al. [48] in 2015 (the GSSB algorithm), with $O(n)$ expected work (number of operations) and space, and $O(\log n)$ span (longest dependencies) *whp* [17]. Despite the asymptotic guarantees, the algorithm has not been widely used in practice for a few reasons. First, the algorithm uses many random accesses and is I/O-unfriendly since it heavily uses hash tables (see Tab. 4). Second, the algorithm *interface* also incurs performance overhead. Specifically, the algorithm assumes the records are associated with *hashed keys* with no duplicates rather than arbitrary keys (more details in Sec. 2.3). This assumption requires additional steps to hash the original keys and resolve collisions subsequently, which may take time comparable to semisort itself. Although none of these issues increase the asymptotic bounds, they both contribute to performance slowdowns that are hard to avoid in a faithful implementation. Hence, the semisort implementation in [48] is not faster than many recent sorting algorithms [10, 13, 59] in practice. Meanwhile, the GSSB algorithm is not stable or (internally-)deterministic (i.e., the result may depend on runtime scheduling) due to the use of parallel hash tables.

In this paper, we revisit the semisort problem, with the goal of achieving a high-performance parallel semisort implementation with a flexible interface. We propose new parallel semisort algorithms that are efficient regarding work, I/O and space usage. Our flexible interface for semisort can also be extended to support efficient and parallel histogram and collect-reduce. Our algorithm takes any key type K, and a user-defined hash function $h: K \mapsto [1, \ldots, n^\kappa]$ to map keys to integers. In principle, the only extra information we need is an *equality test* $=_K: K \times K \mapsto$ *Boolean*. We observe that in most use cases, the key type also supports a *less-than* test $<_K: K \times K \mapsto$ *Boolean* to determine a total ordering, which can be used to improve the performance. We refer to the general semisort algorithm (only $=_K$ supported) and the version with $<_K$ as semisort$_=$ and semisort$_<$, respectively.

Our algorithm builds on the strengths of GSSB [48], but substantially redesigns several components to overcome the existing performance issues of the GSSB algorithm. GSSB works in three steps (we review more details in Sec. 2.3). It first uses samples to determine the heavy (frequent) and light (infrequent) keys, and constructs buckets for them based on estimated sizes from the samples. Each heavy key will be in a separate bucket, while multiple light keys can be grouped into the same bucket. The algorithm then scatters all records to their buckets by placing each record to a random slot in their bucket (or linear probe when the slot is occupied). Lastly, the algorithm refines each light bucket by radix sorting (the hashed keys) in each light bucket. The main issue in GSSB is that the scatter phase is implemented using a parallel hash table, which causes excessive random memory access, some space overhead, instability, and non-determinism.



		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

Figure 1: Heatmap of the relative performance of implementations normalized to the fastest in each test (each row). $n = 10^9$. 64-bit keys and 64-bit values. The parameters in exponential distributions are multiplied by 10^4. The algorithm names and details are introduced in Tab. 2.

To avoid the use of a parallel hash table, we propose an idea inspired by the I/O-efficient parallel samplesort [18]: when constructing buckets and scattering records, we split the input into consecutive subarrays, use auxiliary arrays to count the appearance of each bucket in each subarray, and distribute the records in each subarray based on the counts. This approach enables a cache-friendly access pattern to the input, allows us to obtain the exact size of each bucket, and is stable and race-free (and thus deterministic). However, since samplesort and semisort are quite different, using the idea in [18] does not directly enable high-performance. The challenge lies in choosing the best parameters for the number of heavy and light buckets. At a high-level, the samplesort in [18] creates a bucket for every sampled key. However, using too many buckets increases the size of the auxiliary counting array, which can greatly increase memory accesses. On the other hand, having more buckets is useful to improve parallelism, since each bucket can be processed independently in parallel. Specifically for semisort, we also wish to create more heavy buckets because heavy keys do not need to be refined and are easier to process.

To create the heavy and light buckets in the best way, we propose novel algorithmic ideas for semisort. First, we control the parameters to keep the number of buckets small, so that the auxiliary arrays fit in cache. This avoids excessive memory access to the auxiliary arrays (see more details about the auxiliary arrays in Sec. 3.2 and Fig. 2). Second, we deal with each light bucket *recursively* in parallel. To enable efficient recursive calls, we carefully design optimizations to avoid extra space in recursive calls. Our new approach saves the main memory accesses for the auxiliary arrays, and more interestingly, identifies more heavy keys than GSSB with different degrees of "heaviness" using recursions. The "relatively heavy" keys in each light bucket can be identified and handled more efficiently and improve the overall performance.

In addition to algorithmic improvement for performance, we also redesigned the algorithm interface. Our algorithm directly takes the input with any key type, a user-defined hash function h, and an equality test (or less-than for semisort$_<$). This avoids the additional pre- and post-processing in GSSB. Due to the more flexible interface and algorithm design, our semisort algorithm can be easily extended to histogram and collect-reduce (see Sec. 3.5).

We tested our algorithms on a variety of benchmarks, with different core counts, input sizes, key lengths, and distribution patterns (uniform, exponential, and Zipfian). We summarize our results as a heatmap in Fig. 1. We also test two applications: graph transposing (reordering graph edge lists), and k-gram on English texts. Both our semisort$_=$ and semisort$_<$ algorithms achieve high performance on almost all tests. For example, on 10^9 input data with 64-bit keys and 64-bit values over 15 distributions, our algorithm is 3.4× faster than the GSSB algorithm, and at least 1.27× faster than the best of the previous algorithms, both on average (geometric mean). Our algorithms also consistently perform well on the two applications with real-world data. In all but four application-input combinations we tested in this paper, our algorithm is the fastest. Our code is publicly available [40]. We present more results and analyses in the full version of this paper [39].

2 Preliminaries

2.1 Problem Definitions

Given a sequence of ***records*** from a universe U, we define a key function $key : U \mapsto K$ to define the ***key*** for each record, where K is the key type. We define $=_K$ as the ***equality test*** on K. When applicable, we use $<_K$ as the ***less-than test*** on K. Given a sequence of records A, its key function key_A, and the equality test $=_K$ on K, the ***semisort*** problem is to reorder the records in A to A' such that all records with the same key are contiguous in A'. We also require the user to provide a family of hash functions $h : K \mapsto [1, \ldots, n^\kappa]$, for some constant $\kappa \geq 1$. We call $h(\cdot)$ the *user hash function.*

Given A, key_A, $=_K$, and h_A, the ***histogram*** problem is to emit an array of key-value pairs G consisting of the unique keys of A, with the value for each key equal to the number of times it appears in A. The ***collect-reduce*** function takes the same arguments as semisort and two additional functions: a *map* function $M : U \to E$, and a *reduce* monoid $(\oplus_E, I_E)$. The map function maps a record to a value of some type E. The reduce operation $\oplus_E : E \times E \to E$ combines values of type E with identity I_E. The collect-reduce function returns the array of key-value pairs $R \in K \times E$ consisting of the unique keys of A, with the value associated with each key k equal to $\oplus_{r \in S_k} M(r)$, where $S_k = \{r \in A \mid key_A(r) =_K k\}$. Note that histogram can be expressed as collect-reduce where the map function is the constant function 1, and the monoid is $(+, 0)$. With clear context, we drop the subscripts for these operations and functions.

2.2 Computational Models and Other Notations

We use the work-span (or work-depth) model for fork-join parallelism with binary forking to analyze parallel algorithms [17, 25], which is recently used in many papers on parallel algorithms [4, 5, 12, 16, 18, 19, 21, 23, 29, 33, 35, 43, 47, 72, 76, 77]. We assume a set of threads that share a common memory. A process can `fork` two child software threads to work in parallel. When both children complete, the parent process continues. The ***work*** of an algorithm is the total number of instructions and the ***span*** (depth) is the length of the longest sequence of dependent instructions in the computation. We can execute the computation using a randomized work-stealing scheduler [7, 24, 45] in practice.

To measure the memory access cost in an algorithm, we use the classic I/O model [3, 42]. We assume a two-level memory hierarchy. The processor is connected to the cache of size M, and the cache is connected to an infinite-size main memory. Both cache and main memory are divided into blocks (cachelines) of size B, so there are M/B cachelines in the cache. The CPU can only access the memory on blocks resident in the cache and it is free of charge. We assume an optimal offline cache replacement policy to transfer the data between the cache and the main memory, and a unit cost for each cacheline load and evict. The ***I/O cost*** of an algorithm is the total cost to execute this algorithm on this model. Usually the sequential I/O cost is sufficient to predict the parallel performance [18, 45].

We say that a sorting/semisorting algorithm is ***stable*** if the output preserves the relative order among equal keys from the input order, and otherwise we say that the algorithm is unstable.

We say an algorithm is ***race-free*** when no two concurrent operations in the algorithm can access the same memory access and at least one of them is a write [25]. A race-free algorithm is (internally) deterministic [15], and has many advantages including ease of reasoning about the code, verifying correctness, debugging, and analyzing the performance. In our algorithms, all operations in the algorithm are deterministic once we fix the random seed.

We use $O(f(n))$ *with high probability* (whp) in n to mean $O(cf(n))$ with probability at least $1 - n^{-c}$ for $c \geq 1$.

2.3 The GSSB Semisort Algorithm

We first review the existing GSSB semisort algorithm [48]. As mentioned, the practical performance of GSSB is limited due to its excessive random memory accesses and restrictive interface. Our algorithm builds on the strengths of the GSSB, while overcoming the aforementioned limitations. The GSSB algorithm assumes the input as a sequence of *hashed keys* in range $[0, \ldots, n^\kappa]$ for some constant $\kappa \geq 1$, and semisorts the hashed keys.

Sampling and Bucketing. This is a key technique in GSSB to handle heavily duplicate keys. GSSB first selects a sequence S of samples from the input sequence A with sample rate $p = O(1/\log n)$. The samples will be used to give an initial partition of the records into buckets, such that the same key always goes to the same bucket. Based on the samples, the keys are divided into ***heavy keys*** and ***light keys*** otherwise. We call the records with heavy (light) keys the ***heavy (light) records***. The theory behind this idea is that if sufficient ($\Omega(\log n)$) samples for a key k can be obtained, one can estimate the frequency of k (relatively) accurately. We call them the ***heavy keys*** or ***heavy records***. Let n_H be the number of heavy keys identified by the samples. The algorithm will construct n_H ***heavy buckets***, each for an individual heavy key. Meanwhile, a key k with few ($o(\log n)$) samples are unlikely to appear many times in the input, and we call them ***light keys*** or ***light records***. The light records are grouped into $n_L = \Theta(n/\log^2 n)$ ***light buckets*** by using the hashed value to randomly map to one of the n_L buckets. Our new algorithm will also use a similar technique to detect heavy (duplicate) keys, but with different parameters for better performance.

Size Estimation and Scattering. For a bucket with s samples, GSSB uses a ***size estimation*** function $f(s)$ to upper bound bucket size *whp*. The algorithm will allocate an array of size $(1+\epsilon)f(s)$ for this bucket for some constant $\epsilon > 0$. Then each record is scattered to

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.

a random position in the corresponding bucket. This is performed by using `compare_and_swap`, which atomically puts the record into the position, and re-picking another position using linear probing upon collisions or conflicts. Our new algorithms do not use this approach.

Local Sort and Pack. After scattering, all the heavy keys are collected in individual heavy buckets. Each light bucket can contain more than one light key type. The records in a bucket may not be contiguous due to the random scattering. GSSB then uses a radix sort (on the hashed keys) to refine light buckets (comparison sort is used in practice) and make them contiguous. A ***packing step*** is needed for heavy buckets to put records in contiguous slots. Our new algorithm also uses different approaches in this step.

The main performance issue in GSSB is the random access in the scatter phase—each record is assigned to a random location, and has to retry if necessary. GSSB hence needs $O(n)$ random writes for the scattering phase, which is I/O-inefficient. This also requires more space (and thus memory footprint) since we need to ensure a load factor $c < 1$. We will show how to overcome this issue, as well as to make our new algorithms stable and race-free in Sec. 3.

Another major issue of GSSB is its interface. GSSB assumes a collision-free hash function $h : K \mapsto [1, \ldots, n^\kappa]$ that maps arbitrary key types to random integers (hashed keys), and the algorithm (and implementation) directly semisort the hashed keys, which are random integers. When using more realistic and practical hash functions with possible collisions, one has to perform preprocessing and postprocessing to deal with collisions. While such pre/postprocessing do not asymptotically increase the cost of the algorithm in theory, they can in practice incur significant time overheads comparable to semisort itself ($O(n)$), and therefore make using semisort in applications prohibitively costly, relative to sorting.

3 Our New Algorithms

In this section, we present our algorithms for semisort and related problems. We present the useful notations in Tab. 1. We start by overviewing the high-level idea, and then present more details in Sec. 3.1 to 3.4. We discuss how to support histogram and collect-reduce in Sec. 3.5. In Sec. 3.6, we present the theoretical analysis of our algorithm, and discuss the choices of parameters in theory and in practice.

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.
            $\alpha$: base case threshold.
            $l$: subarray size.
if $|A| < \alpha$ then return BaseCase($A, h$, COMP)          // Base cases
Sampling and Bucketing:
$S \leftarrow n_L \log n'$ sampled keys from $A$
Count the occurrences of each key in $S$
Initialize the heavy table $H$
$id \leftarrow n_L$
// This for-loop can also be performed in parallel theoretically
for each distinct key $k \in S$ 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 $k$
        $id \leftarrow id + 1$
$n_H \leftarrow$ number of distinct keys in $H$
Blocked Distributing:
Initializing matrix $C[\,][\,]$ with size $(n_L + n_H) \times (n'/l)$
parallel_for $i : 0 \le i < n'/l$ do          // For each subarray
    for $j : i \cdot l \le j < (i+1) \cdot l$ do
        $id \leftarrow$ GetBucketId($key(A[j]), H, h, n_L$)
        // $C[i][id]$: #records falling into bucket $id$ in subarray $i$
        $C[i][id] \leftarrow C[i][id] + 1$
Initialize $T$ of size $n'$
// $X[i][j]$: offset in $T$ for record in subarray $i$ going to bucket $j$
Compute $X[i][j] \leftarrow \sum_{j'<j \text{ or } (j'=j, i'<i)} C[i'][j']$
parallel_for $i : 0 \le i \le n_L + n_H$ do
    $offsets[i] \leftarrow X[i][0]$
parallel_for $i : 0 \le i < n'/l$ do          // For each subarray
    for $j : i \cdot l \le j < (i+1) \cdot l$ do
        $id \leftarrow$ GetBucketId($key(A[j]), H, h, n_L$)
        $T[X[i][id]] \leftarrow A[j]$
        $X[i][id] \leftarrow X[i][id] + 1$
$A \leftarrow T$          // Avoided in implementation, see Sec. 3.4
Local Refining:
parallel_for $i : 0 \le i < n_L$ 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 $h(k) \mod n_L$          // $h(\cdot)$ is the hash function
```

Our semisort algorithm follows the same framework as GSSB, but employs novel techniques to improve the performance for *all the steps*. Our new algorithm is **I/O-friendly, stable**, and **race-free**. In contrast to GSSB, we do not require pre-hashing the keys. Our algorithm directly handles input records of any type, and extracts the hashed keys by applying the user hash function in the algorithm when needed. This generality in the interface also improves efficiency both in time and space—it avoids the pre- and post-processing, as well as the hash table to pre-hash keys and resolve collisions, which can incur another $O(n)$ random reads and $O(n)$ extra space. Our algorithm is *stable*—all records with the same key will be kept in the same order in the output. This feature is useful for collect-reduce and histogram and increases their generality, as discussed in Sec. 3.5. Our algorithm is also *race-free*, which means no concurrent writes are needed to any shared memory position. This also makes our algorithms simple, practical, and *internally-deterministic* (i.e., the output does not depend on runtime scheduler).

Our algorithm consists of three steps: *Sampling and Bucketing* (find heavy keys), *Blocked Distributing* (count the number records in each bucket), and *Local Refining* (refine the ordering of records in light buckets).

(1) ***Sampling and Bucketing.*** First, the algorithm performs sampling to find the heavy keys. Similar to GSSB, each heavy key uses an individual bucket, and multiple light keys share a bucket. However, we pick a smaller number of buckets for a better overall memory-access pattern (see discussions in Sec. 4.1).
(2) ***Blocked Distributing.*** Next, it counts the exact number of records in each bucket. Given the bucket sizes, the algorithm distributes input records to their associated buckets in an *I/O-friendly* manner. By performing *exact* counting, the temporary arrays used are only of size n, and no parallel hash tables are necessary (as in GSSB). This distribution step makes the algorithm stable and race-free.
(3) ***Local Refining.*** After Step 2, the heavy keys are at their final positions in the heavy buckets. For light buckets, unlike GSSB, our algorithm *recursively semisorts* them until the recursive input size is small enough (i.e., fitting in cache), at which point the keys are semisorted sequentially. This approach allows the algorithm to detect "medium-level" heavy keys in subsequent recursive rounds and also reduce the total number of I/Os.

The pseudocode of our algorithm is given in Alg. 1, and a running example is given in Fig. 2. Next, we introduce the details of each step and explain why our decisions improve the performance of our algorithm. Since our algorithm uses recursive calls, we use n as the input size of the original (top-level) problem, and use n' as the current subproblem size in the recursive call.

3.1 Step 1: *Sampling and Bucketing*

The goal of the sampling and bucketing step is similar to GSSB—we want to identify heavy and light keys, which decides the bucket id for each record. Instead of having $O(n/\log^2 n)$ light buckets as in GSSB, we use the number of heavy and light buckets (n_H and n_L) as parameters. We use n_L as a tunable parameter, and set the upper bound of n_H accordingly as $O(n_L)$. We will later discuss in Sec. 3.6 about how to pick n_L to achieve the best practical performance.

To determine heavy keys, we take a sequence of samples S of size $\Theta(n_L \log n)$ by selecting each record uniformly at random (Lines 2–10 in Alg. 1). The light keys are keys appearing fewer than $\log n$ times, which indicates that their actual number of occurrences is small. We group multiple light keys into one bucket based on the hash keys. We create n_L light buckets by evenly partitioning the range of hashed keys (given by the user hash function) into n_L buckets. For simplicity, we use $n_L = 2^b$ as a power of 2, and the light bucket id of a key k is obtained by taking the last (least significant) b bits in the hash value of k, i.e., the bucket id is $(h(k) \bmod n_L)$.

The heavy keys are those appearing at least $\log n$ times in the samples; as in the analysis of GSSB, their actual number of occurrences is large *whp*. Given the sample size $|S| = \Theta(n_L \log n)$, the number of heavy keys $n_H = O(n_L)$. We will create n_H buckets with ids in $[n_L, n_L + n_H)$ (the first n_L buckets are for the light keys). We use a sequential hash table H to store all heavy keys associated with their bucket ids, referred to as the *heavy table*, so that the later steps can look up whether a key is heavy in constant work.

3.2 Step 2: *Blocked Distributing*

Unlike GSSB, which uses a *scattering* step to place records to random positions in the buckets, our algorithm uses a more I/O-friendly and space-efficient approach. The goal of this step is to count the *exact* number of records in each bucket, and distribute all records to the associated buckets into contiguous slots. Since we have the exact count, we only need an array T of size n for all the buckets, making our algorithm space-efficient. This distribution step makes our algorithm stable and race-free.

Our idea is inspired by recent sorting algorithms [10, 18, 26, 59]. We first partition the sequence evenly into n'/l subarrays, each with l records (recall that n' is the current subproblem size). We then process all the subarrays in parallel (Line 12), but sequentially within each individual subarray (Line 13). We count the number of records in each bucket in a $(n'/l) \times (n_L + n_H)$ matrix C, which is referred to as the *counting matrix*. In particular, C_{ij} is the number of records in subarray i falling into bucket j. To do this, within each subarray i, our algorithm determines which bucket each key k belongs to using the GetBucketId function (Line 27). This function first looks up the heavy table H to check if k is a heavy key, and if so, it obtains the bucket id j from H. Otherwise, the bucket id of a light key k is simply given by $j = h(k) \bmod n_L$. We then increment the corresponding cell in C_{ij} by one (Line 14).

We then distribute all records in the input to their corresponding buckets, using the information in C. To do so, we compute the offset per subarray per bucket as a *prefix array* X that has the same size as C. Array X can be computed using the prefix sum of C, but in the column-major order (Line 15, see an illustration in Fig. 2). After the prefix array X is computed, we once again process each subarray and move each record to its corresponding bucket (Line 18–22) in the temporary array T. This step takes $O(1)$ work per record—we use $O(1)$ work to decide which bucket a record is in, and after that, we move the record and increment the offset counter in X.

We noticed that when picking the appropriate parameters, our approach is much faster than the corresponding step in GSSB in practice, mainly due to smaller memory footprint and fewer memory accesses. We will later show the analysis in Sec. 3.6.

3.3 Step 3: *Local Refining*

After the previous step, we have all heavy keys stored contiguously in their corresponding heavy buckets, which are also their final positions in T. Light keys are still unsorted. We work on each light bucket in parallel by recursively semisorting each of them. We stop recursing and switch to the base case when the bucket size is small enough and fits in cache, which is decided by the parameter α (Line 1). For our experiments with input sizes ranging from 10^8–10^9, we typically need one more level of recursion before reaching the

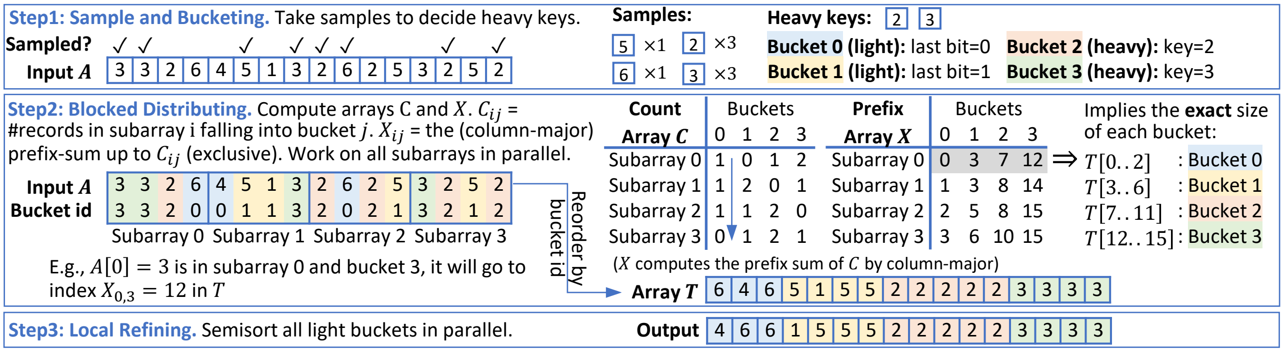

Step1: Sample and Bucketing. Take samples to decide heavy keys.
Sampled? ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓
Input A 3 3 2 6 4 5 1 3 2 6 2 5 3 2 5 2
Samples: 5 ×1 2 ×3 6 ×1 3 ×3
Heavy keys: 2 3
Bucket 0 (light): last bit=0
Bucket 1 (light): last bit=1
Bucket 2 (heavy): key=2
Bucket 3 (heavy): key=3
Step2: Blocked Distributing. Compute arrays C and X. C_ij = #records in subarray i falling into bucket j. X_ij = the (column-major) prefix-sum up to C_ij (exclusive). Work on all subarrays in parallel.
Input A 3 3 2 6 4 5 1 3 2 6 2 5 3 2 5 2
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
E.g., A[0] = 3 is in subarray 0 and bucket 3, it will go to index X_0,3 = 12 in T
Reorder by bucket id
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
Implies the exact size of each bucket:
T[0..2] : Bucket 0
T[3..6] : Bucket 1
T[7..11] : Bucket 2
T[12..15] : Bucket 3
(X computes the prefix sum of C by column-major)
Array T 6 4 6 5 1 5 5 2 2 2 2 2 2 3 3 3 3
Step3: Local Refining. Semisort all light buckets in parallel.
Output 4 6 6 1 5 5 5 2 2 2 2 2 2 3 3 3 3

Figure 2: Our algorithm with a running example. We consider an input with $n = 16$ records with keys given. $|S| = 8$ samples are taken, and keys with more than 2 samples are heavy keys. $n_L = n_H = 2$ in this example. We have $l = 4$ subarrays each with 4 records. We compute the counting matrix C and prefix array X as shown, and records can be distributed accordingly. Finally the local refining step recursively solves the 2 light buckets.

base case, if most of the keys are light keys. Since the base-case size fits in cache, the time to semisort the base cases is small. We provide two solutions for the base cases: semisort$_=$ and semisort$_<$.

semisort$_=$. In the base case, semisort$_=$ uses a sequential hash table with chaining. We first build a hash table of size $(1+\epsilon)n'$ for some constant $\epsilon > 0$. Then, we iterate over all keys and insert each key to the hash table with separate chaining. Finally, all records are packed to the output by looping over the hash table in order. Chaining allows the algorithm to maintain the order of the original input for records with the same key, and thus our algorithm is stable. Since each base case is small, the hash table can be maintained locally by each thread.

semisort$_<$. In the base case, semisort$_<$ uses a standard comparison sort. By using a stable comparison sort, we can also guarantee the stableness of our semisort.

3.4 In-place Optimization

Before the recursive call in Alg. 1, we copy the temporary array T back to A (Line 23). This copy accesses the whole arrays A and T, which is expensive in practice. We note that we can save this copying by swapping the two arrays A and T in the recursive call. Namely, we skip Line 23 and in Line 25 we sort the light buckets in T, and use the corresponding part in A as the other array to take the output. For the base cases and the heavy buckets, if they happen to reside in T, we copy them back to A. By doing this, we avoid the copying in Line 23. This reuses the auxiliary array T and also avoids allocating new memory in every recursive level. Since in most cases the recursion will reach the base case in two levels, the entire algorithm copies the data twice per record, first from A to T, and then from T back to A.

Here we use "in-place" to indicate that the input and output of semisort are in the same array. Our algorithm still uses $O(n)$ extra space. We will discuss how to reduce space usage in Sec. 6.

3.5 Supporting Histogram and Collect-Reduce

Using our semisort algorithm, the histogram and collect-reduce primitives can be supported with minor modifications. Here we will elaborate on collect-reduce since histogram can be considered as collect-reduce with values always equal to 1 for all records.

We still use the *Sampling and Bucketing* step to determine the heavy keys. Then in the *Blocked Distributing* step, it is unnecessary to distribute the heavy keys to their corresponding buckets. Instead, we first directly compute the reduced values (or counts for histogram) for the heavy records in each subarray (all the subarrays can be processed in parallel), and then reduce the results of all subarrays. In the base-case of the *Local Refining* step, we use the version based on hash tables. When any duplication is identified, we directly combine their values instead of chaining. Since the algorithm is stable, it works on any associative reduce functions (in particular, there is no need to be commutative).

Generally speaking, histogram and collect-reduce can be significantly faster than semisort when there are many heavy duplicate keys, as we do not need to distribute the heavy records and only need to distribute the "locally reduced value" for each heavy key in each subarray. When no or few duplicate keys are in the input, histogram and collect-reduce can perform slightly slower than semisort. This is because they perform almost identical computations as semisort to reorder records, but need an extra step to pack the keys and reduced values into the output.

3.6 Analysis and Parameter Choosing

Our new semisort algorithm has three parameters: l (subarray size), n_L (light bucket number), and α (base case size). Other parameters (e.g., the number of heavy buckets n_H) are set accordingly. The values of l and n_L are fixed for *all levels of recursions*. To ensure the space usage is $O(n)$, we will assume $n_L \leq l$ since the sizes of matrices C and X has size $\Theta(n_L \cdot n/l)$. We also assume the sample set size $|S| = n_L \log n = O(n)$. In the following, we will use n as the *original problem size*, and use n' as the current size of the recursion.

We will analyze the cost bounds and show that our semisort algorithm is efficient under *reasonable assumptions* of modern multicore architecture. Then we will show how to select the parameters in practice for the best practical performance.

Theoretical Analysis. We start with analyzing the number of recursion levels in our algorithm.

Lemma 3.1. *The number of recursion levels is $O(\log_{n_L}(n/\alpha))$* whp *for both semisort$_=$ and semisort$_<$.*

Proof. From the same analysis from GSSB [48], the number of records in each light bucket is $O(n/n_L)$ *whp*. Therefore, the light bucket size shrinks by a factor of $\Theta(n_L)$ *whp* in each level of recursion, and the number of recursive levels is $O(\log_{n_L}(n/\alpha))$ *whp*. □

For simplicity in stating the bounds, we use $\boldsymbol{r} = O(\log_{n_L}(n/\alpha))$ to denote the ***number of recursion levels***. We start with the work of the algorithms and present the result in Thm. 3.2.

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

Proof. We first show the work analysis for semisort$_=$. We start with considering the top level of recursion. As assumed above, the number of samples is $O(n_L \log n) = O(n)$, and thus the *Sampling and Bucketing* step has $O(n)$ work. In the *Blocked Distributing* step, it takes $O(1)$ work per record to find the bucket it belongs to. As mentioned above, we assume $n_L \le l$ so that the counting matrix C and prefix array X have sizes $O(n)$, and computing prefix sum also has $O(n)$ work. The step to distribute the records to array T (lines 18–22) is also $O(n)$ since each record is processed once. For each recursion level, this argument is still true, and the work of all the subproblems in one level adds up to $O(n)$. Assuming r recursion levels, the work before entering the base cases is $O(n)$ for both semisort$_=$ and semisort$_<$. For semisort$_=$, the work of each base case is $O(n')$, which gives $O(n)$ total work for all base cases. For semisort$_<$, the work of each base case is $O(n' \log n')$, where n' can be at most α. Therefore the total base-case work is $O(n\log\alpha)$ for semisort$_<$. Combining the results gives the bounds in Thm. 3.2. □

Although semisort$_<$ has a higher work, the overhead is caused by the comparison sort in base cases. However, the base cases fit in cache and are highly-optimized. In the experiments semisort$_<$ shows as good performance as semisort$_=$ in most cases.

We then analyze the span of semisort$_=$ and semisort$_<$, and show that they are highly parallel.

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.

Proof. The *Sampling and Bucketing* step is executed sequentially with $O(n_L \log n)$ span. We note that this step can be easily parallelized [48], but our implementation still performs it sequentially, since it is cheap anyway. For the distributing step, we have two sequential for-loops (Lines 13 and 19), leading to $O(l)$ span. Computing the prefix sum (X from C) has $O(\log n)$ span. In total, the span of one recursive level is $O(l + n_L \log n)$. Hence, considering r recursive levels, both algorithms have $O((l + n_L \log n)r)$ span before the base cases. semisort$_=$ uses sequential hash tables in base cases, which leads to $O(\alpha)$ span. semisort$_<$ uses a comparison sort in base case, which can achieve $O(\log n)$ span *whp* in theory [17] (our implementation coarsens the base case by using a sequential sort, since the base case size is small). Combining the results above gives the bounds in Thm. 3.3. □

Considering both work and span, the parallelism (defined by the ratio between work and span) for both algorithms is roughly $\Theta(n/l)$ (in practice we choose l much larger than n_L and α). Given the number of processors P in a machine, our semisort algorithm achieves sufficient parallelism if we can set $n/l = \Omega(P)$.

We analyze the I/O bound of the algorithms with our choices of parameters to make the bound optimal ($O(n/B)$). We make the assumption that $M/B = \Omega(n^{1/2})$ (recall that M and B are cache size and cacheline size, respectively). For reasonable values of $n \le 10^{12}$, this assumption is true for both commodity machines (e.g., laptops) as well as more powerful servers. We present our results in Thm. 3.4.

Theorem 3.4. *Assume $M/B = \Omega(n^{1/2})$, using parameters $n_L = \Theta(n^{1/4})$, $\alpha = \Theta(n^{1/2})$, and $l = \Theta(n^{3/4})$, both semisort$_=$ and semisort$_<$ have I/O cost of $O(n/B)$* whp.

Proof. Given the parameters in the theorem, the number of recursive levels is $r = O(1)$ *whp*. Therefore, we only analyze the top-level recursion. Since the sizes of C and X are $O(n_L \cdot (n/l)) = O(\sqrt{n}) = O(M)$, all memory accesses to arrays C and X fully fit into cache except for the first access. When $M/B = \Omega(n^{1/2})$ and $\alpha = \Theta(n^{1/2})$, we can choose α to fit the base cases in cache, such that the base cases can be solved without using additional main memory access after loading the data to the cache. The only cache misses are when accessing the input array A and the buckets T. The accesses to A are all serial accesses. For T, we are writing serially from $(n_H + n_L) \cdot (n/l)$ pointers as stored in the X matrix. Even when all the pointers are non-consecutive, only $(n_H + n_L) \cdot (n/l) = O(n^{1/2})$ cachelines are active at any time, and they all fit in cache. For every pointer, there is one cache miss every B accesses to the array T. Therefore, the total I/O cost to generate T is $O(n/B)$. Note that this analysis is true for both the root level (when input size is n), as well as the recursive levels (the total sizes of C and X for all subproblems in the same recursive level are still $(n_H + n_L) \cdot n/l = O(n^{1/2})$). In summary, both semisort$_=$ and semisort$_<$ have I/O cost $O(n/B)$ *whp* assuming $M/B = \Omega(n^{1/2})$, which improves the $O(n)$ I/O bound of GSSB by a factor of $O(B)$. The bound is optimal, since loading the input needs $\Omega(n/B)$ I/Os. □

Since I/O-efficiency is one of our main design goals, we use the parameters in Thm. 3.4 to present the work and span bounds below.

Theorem 3.5. *Assume $M/B = \Omega(n^{1/2})$, and parameters $n_L = \Theta(n^{1/4})$, $\alpha = \Theta(n^{1/2})$, and $l = \Theta(n^{3/4})$. semisort$_=$ has $O(n)$ work, $O(n^{3/4})$ span, and $O(n/B)$ I/O cost. semisort$_<$ has $O(n\log n)$ work, $O(n^{3/4})$ span, and $O(n/B)$ I/O cost. All bounds are* whp *in n.*

Parameters in our Implementations. The performance of our semisort algorithm is reasonably consistent for a large parameter range. The best parameters of each input instance can be different, decided by input size, heavy record ratio, etc. In our implementation and all experiments, we pick $n_L = 2^{10}$, $l = n/5000$ (at most 5000 subarrays in all subproblems in one recursive level), and $\alpha = 2^{14}$. These numbers satisfy the conditions in the theoretical analysis in Thm. 3.5 when $n = 10^8$ to 10^9. We set the number of samples $|S| = 500 \log n$, so we can have at most $n_H = 500$ heavy keys. We set up these parameters to ensure that the matrices C and X and the base cases are small enough to fit in the last-level cache for modern multicore machines.

4 Comparisons with Existing Algorithms

4.1 Improvements over GSSB

In this section, we compare and discuss the improvements of our algorithm(s) over the existing semisort algorithm GSSB.

Flexible Interface. Recall that GSSB requires hashed keys (integers) as input, which needs a pre- and post-processing to resolve collisions. Our algorithm supports *arbitrary key types K with $=_K$* or $<_K$, with a user hash function. For integer keys, we provide the option to use the identity function, resulting in ***semisort-i$_=$*** and ***semisort-i$_<$***, which can be much faster in many cases, although we note that these versions do not admit as good theoretical bounds. Our interface also supports histogram and collect-reduce only with minor changes.

Low Space Usage. In the *Blocked Distributing* step, we compute the exact counts for the buckets, so when distributing the keys, the total size of the buckets is n, instead of $\Theta(n)$ as in GSSB (their buckets need to have a load factor smaller than 1 because of random scatter). Other than space overhead, GSSB also needs a costly packing step.

I/O-Efficiency. Our algorithm also uses several techniques to enable a better memory access pattern. We pick a small number of buckets ($n_L = 2^{10}$), as opposed to $O(n/\log^2 n)$ of them in GSSB, such that the counting matrix C and its prefix sum X in our algorithms fit in cache (recall that we access them in column-major). As such, the *Blocked Distributing* step incurs no random accesses to the main memory.

Stability and Determinism. Due to avoiding using parallel hash tables, our semisort algorithms (both semisort$_=$ and semisort$_<$) are stable and race-free. GSSB is not race-free (due to using parallel hash tables), and is unstable (heavy keys are in random order), and thus cannot support non-commutative operations in collect-reduce.

4.2 Relationship to Sample Sort and Integer Sort

Many ideas in our semisort algorithm are closely related to ideas in sorting algorithms, as we will discuss in this section.

Samplesort. Samplesort is the general idea of using multiple pivots in quicksort; clearly this algorithm can be used to solve the semisort$_<$ problem. The algorithm selects p pivots, uses them to partition the input into $p + 1$ buckets, and sorts all of the buckets in parallel. We refer the audience to [10] for a detailed literature review on samplesort. We compare to the state-of-the-art samplesorts from ParlayLib [13] and IPS^{4}o [10] in our experiments.

Similar to samplesort, our algorithm also partitions the input into buckets and processes them in parallel. However, samplesort is a comparison sort that requires the $<_k$ operation, and has an $\Omega(n \log n)$ lower bound in work, whereas our semisort$_=$ algorithm only requires $O(n)$ work. The ParlayLib samplesort [13] uses one level of partition. IPS^{4}o [10] (preliminary version as [61]) also uses a small number of samples and sort recursively. They use an implicit search tree (breadth-first traversing the tree that stores the sorted pivots) to find the bucket for each record, which is not required in our algorithm. They also use a smart approach for the distribution step, and we discuss this in Sec. 6.

Integer sort. Integer sorting can be used to semisort integer keys for semisort$_=$, or to semisort the hash value of any key types with an extra step to resolve collisions. Unlike the sequential radix sort that starts from the least-significant bits, all parallel integer sort algorithms are top-down and first look at the most-significant bits. We refer to [59] for a detailed literature review for parallel integer sort. We compare to the state-of-the-art integer sorts from ParlayLib [13], RegionsSort [59], and IPS^{4}Ra [10] in our experiments.

The major advantage of our semisort algorithm over integer sorting is that our algorithm can identify heavy keys. Consider a heavy key x and a light key $x + 1$. Our algorithm will put x in a separate heavy bucket, and only deal with $x + 1$ in a light bucket in the next steps. For existing integer sorts [10, 13, 59], both keys are likely kept in the same bucket for all levels and separated only in the last round, which can result in significant wasted work and load imbalance.

Name	Stable	Det.	K	COMP	Notes
Ours$_=$	Yes	Yes	Any	=	Our semisort$_=$ algorithm
Ours$_<$	Yes	Yes	Any	<	Our semisort$_<$ algorithm
Ours-i$_=$	Yes	Yes	Int	=	Our integer semisort$_=$ algorithm
Ours-i$_<$	Yes	Yes	Int	<	Our integer semisort$_<$ algorithm
Ours$_\oplus$	Yes	Yes	Any	=	Our collect-reduce algorithm
PLSS	Y/N	Yes	Any	<	ParlayLib sample sort [13]
PLIS	Yes	Yes	Int	<	ParlayLib integer sort [13]
IPS^{4}o	No	No	Any	<	IPS^{4}o sample sort [10]
IPS^{2}Ra	No	No	Int	<	IPS^{2}Ra integer sort [10]
GSSB	No	No	Int	=	GSSB semisort [48]
RS	No	No	Int	<	RegionsSort [59]
PLCR	Yes	Yes	Any	<	Collect-reduce from ParlayLib [13]

Table 2: Algorithms tested in our experiments. "Det." = determinism. "K" = key type. "Any" = any input key type. "Int" = only allows for integer keys. "COMP" = required comparison function. PLSS has two implementations. We use the faster but unstable version in our experiments.

5 Experiments

Experimental Setup. We run our experiments on a 96-core machine (with two-way hyper-threading) with 4 × 2.1 GHz Intel Xeon Gold 6252 CPUs processors (with 36MB L3 cache) and 1.5TB of main memory. We implement our algorithms in C++ using ParlayLib [13] for fork-join parallelism and some parallel primitives. We compile our code using `clang` version 14.0.6 with `-O3` flag. We always use `numactl -i all` to interleave the memory on all CPUs except for sequential tests. We run each test six times and report the median of the last five runs. All running times are given in **seconds**.

Baseline Algorithms. We compare our algorithms to the state-of-the-art comparison and integer sorting algorithms and collect-reduce algorithms. We provide the list of the baseline algorithms we compare our algorithms against in Tab. 2. For fairness and consistency, we require the output to be written to the input array (i.e., in-place). We note that this is beneficial for PLSS, IPS^{4}o, IPS^{2}Ra, and RS as they are originally designed for the in-place setting. Some of the baselines only work for integer types (***integer-only***), including PLIS, GSSB, RS, and IPS^{2}Ra. IPS^{4}o and PLSS work on any input types (***any-type***). For the any-type algorithms, semisort$_<$, PLSS and IPS^{4}o require the less-than test $<_K$, while our semisort$_=$ only needs the equality-test $=_K$. GSSB assumes the input keys are already hashed and does not resolve collisions, so we also categorize it as integer-only. Among all implementations, all our algorithms and PLIS are *stable*. This also means that they can be applied to collect-reduce with arbitrary (associative) reduce operations, while the others also require the reduce operation to be commutative. We note that there are two versions of samplesort in ParlayLib. The stable one is slower and the unstable one is faster. Our experiments use the unstable but faster version. When comparing the *average* performance, we always use the ***geometric mean***.

We also tested classic sequential sorting algorithm such as STL sort and a sequential hash table. However, since they are not I/O efficient, even their sequential performance is not as fast as the sequential execution of the aforementioned parallel algorithms. Hence, we do not include their running time in our experiment.

	Para-	Dist.	Max	Heavy	Any Type				Integer Only					
	meter	Keys	Freq.	Freq.	Ours$_=$	Ours$_<$	PLSS	IPS^{4}o	Ours-i$_=$	Ours-i$_<$	PLIS	GSSB	RS	IPS^{2}Ra
Uniform	10	10	100M	100%	0.730	$\underline{0.707}$	1.12	0.863	0.650	$\underline{0.615}$	1.92	2.77	1.27	3.77
	10^3	1K	1M	100%	0.743	$\underline{0.740}$	0.975	0.762	0.693	$\underline{0.693}$	1.37	4.13	1.53	1.66
	10^5	100K	10K	0%	$\underline{0.725}$	0.727	1.32	1.10	$\underline{0.684}$	0.685	1.18	2.36	1.37	1.02
	10^7	10M	100	0%	0.986	$\underline{0.984}$	1.41	1.05	0.950	$\underline{0.873}$	1.11	2.49	1.72	1.03
	10^9	1B	1	0%	$\underline{1.00}$	1.16	1.57	1.11	$\underline{0.958}$	1.30	1.10	2.74	1.37	1.10
	Avg.	-	-	-	$\underline{0.828}$	0.846	1.26	0.966	$\underline{0.775}$	0.802	1.31	2.84	1.45	1.49
Exponential	1×10^{-4}	182K	100K	89.6%	$\underline{0.723}$	0.726	1.25	0.928	0.690	$\underline{0.686}$	1.83	2.44	1.52	1.05
	7×10^{-5}	252K	70.0K	85.2%	0.730	$\underline{0.729}$	1.28	0.985	0.694	$\underline{0.691}$	1.64	2.46	1.48	1.01
	5×10^{-5}	343K	50.0K	79.3%	0.740	$\underline{0.733}$	1.32	1.04	0.699	$\underline{0.695}$	1.48	2.45	1.40	1.01
	2×10^{-5}	789K	20.0K	48.2%	$\underline{0.765}$	0.765	1.33	1.15	0.719	$\underline{0.709}$	1.18	2.53	1.41	1.04
	1×10^{-5}	1.47M	10.0K	0.00%	0.822	$\underline{0.808}$	1.34	1.16	0.750	$\underline{0.738}$	1.12	2.60	1.40	1.03
	Avg.	-	-	-	0.755	$\underline{0.752}$	1.30	1.05	0.710	$\underline{0.703}$	1.43	2.49	1.44	1.03
Zipfian	1.5	1.79M	383M	97.7%	$\underline{0.682}$	0.686	2.07	1.56	0.663	$\underline{0.643}$	2.37	2.50	1.71	6.50
	1.2	34.7M	181M	83.6%	$\underline{0.767}$	0.773	1.50	1.21	0.676	$\underline{0.667}$	1.81	2.46	1.70	3.31
	1	210M	46.9M	42.2%	$\underline{0.925}$	0.997	1.25	1.08	$\underline{0.815}$	0.841	1.39	2.65	1.54	1.67
	0.8	525M	3.22M	5.32%	$\underline{1.00}$	1.16	1.50	1.12	$\underline{0.930}$	0.971	1.13	2.75	1.45	1.12
	0.6	756M	100K	0.10%	$\underline{1.00}$	1.16	1.57	1.12	$\underline{0.949}$	1.00	1.11	2.73	1.39	1.10
	Avg.	-	-	-	$\underline{0.866}$	0.934	1.56	1.21	$\underline{0.797}$	0.811	1.49	2.62	1.55	2.13
Overall Geometric Mean					$\underline{0.815}$	0.841	1.37	1.07	$\underline{0.760}$	0.770	1.41	2.65	1.48	1.48

Table 3: Running times with different input distribution with $n = 10^9$, 64-bit keys and 64-bit values. Underlined numbers are the fastest running time in each distribution-input type instance. "parameter" = distribution parameters (i.e., μ in uniform, λ in exponential, and s in zipfian distribution). "Distinct keys", "maximum frequency", and "heavy frequency" are statistics for each test (see details in Sec. 5). The algorithm names are described in Tab. 2. "Avg." means geometric mean numbers across multiple tests.

Our Algorithms. We use the two versions of our algorithm semisort$_<$ and semisort$_=$ that work on ***any-type***. In tables and figures, we also use "Ours$_=$" and "Ours$_<$" to refer to them, and use "Ours$_\oplus$" to refer to our collect-reduce implementation. When comparing with the integer-only implementations, we use simplified versions without hashing (see Sec. 4.1), and call them semisort-i$_=$ and semisort-i$_<$ (or "Ours-i$_=$" and "Ours-i$_<$"), where the hash function is an identity function. The choices of parameters in our algorithms are in Sec. 3.6.

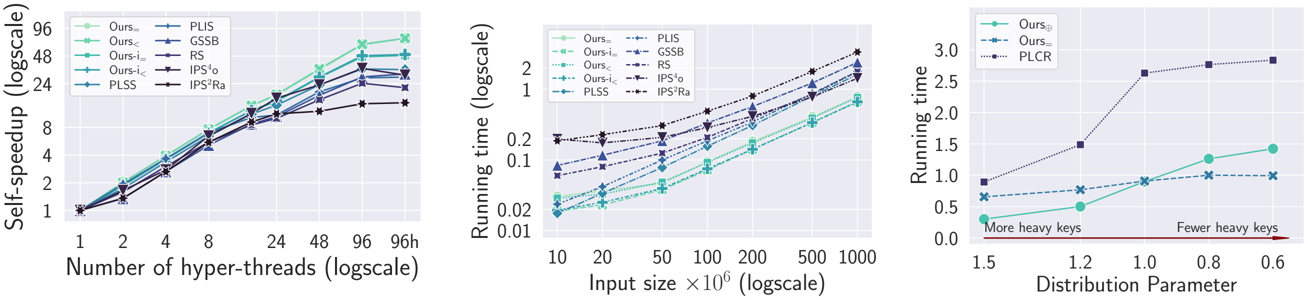
Input Distributions. We use three distributions for evaluating our algorithms: uniform(μ), exponential(λ), and Zipfian(s). If not specified, the default setting is $n = 10^9$ with 64-bit keys and 64-bit values. We also include tests of our algorithm on varying input sizes and key lengths (Figs. 3b and 4). For uniform distribution, we test $\mu = 10^1, 10^3, 10^5, 10^7, 10^9$. For exponential distribution, we test $\lambda = 1 \times 10^{-5}, 2 \times 10^{-5}, 5 \times 10^{-5}, 7 \times 10^{-5}, 1 \times 10^{-4}$. For Zipfian distribution, we test $s = 0.6, 0.8, 1, 1.2, 1.5$. We use *distribution-param* to denote the input distribution with parameter *param* (e.g., *uniform*-10^9). We show relevant statistics of the inputs along with our results in Tab. 3. We present the number of distinct keys, the maximum frequency, and the ratio of keys with more than $500 \log n$ occurrences, which is noted as "Heavy Freq." in Tab. 3. They are measured for each distribution to indicate skewness of the data. For the synthetic data, we always set the value type the same as the key type. For most of the tests, we provide figures on one representative distribution, and provide more results in the full version [39].

5.1 Overall Performance

We present the running time of all tested implementations with $n = 10^9$ 64-bit keys with different distributions in Tab. 3, and a heatmap (normalizing all running times to the fastest on each test) in Fig. 1. On all but four tests, our algorithms are always the ***best two*** implementations. Among any-type algorithms, our semisort$_=$ and semisort$_<$ are 1.02–2.28× and up to 2.27× faster (respectively) over the best of the other algorithms. For integer-only algorithms, our semisort-i$_=$ is 1.08–2.59× faster than the other algorithms. Our semisort-i$_<$ is about 15% slower than PLIS in one test, and is up to 2.67× faster than all baselines on all other tests.

Overall, our algorithms are always faster than the baseline algorithms using geometric mean. Note that some of the baselines are competitive on some individual tests, such as IPS^{4}o on *uniform*-10^3 and *uniform*-10^9, PLIS on *uniform*-10^9 and *Zipfian*-0.6, and IPS^{2}Ra on *Zipfian*-0.6. However, their performance can be unstable over different distributions. IPS^{4}o is relatively fast on uniform distributions but performs worse on skewed distributions. We also compute the geometric means in Tab. 3 and Fig. 1 to compare the performance on each distribution. Based on these numbers, semisort$_=$ and semisort$_<$ have very close performance (within 5%). All the other algorithms are at least 25% slower than both of our implementations on average. We also show relative performance for 32-bit and 128-bit keys in the full version of this paper [39]. On average, our algorithms are consistently the fastest. We note that not all comparisons are apple-to-apple comparisons. PLSS and IPS^{4}o work for general sorting which is asymptotically harder than semisort. PLIS, RS, and IPS^{2}Ra are for integer sorting, which is also slightly different than semisorting. Also, PLIS and all our implementations are stable while others are not (see Tab. 2).

Interestingly, the integer sort algorithms can be slower than comparison sorts on 64-bit keys. We tested on 32-bit and 128-bit keys and show the running time in the full version of this paper. Unsurprisingly, integer sort algorithms are usually faster than comparison sort algorithms on 32-bit keys, and get worse on 128-bit



(a) **(b)** **(c)**

Figure 3: (a). Self-speedup of all tested implementations with increasing hyper-thread counts on Zipfian-1.2. $n = 10^9$. (b). Scalability with increasing input size (n) of all tested implementations on Zipfian-1.2. (c). Performance of collect-reduce with various Zipfian distributions. $n = 10^9$. Ours$_\oplus$ is our collect-reduce algorithm. Ours$_=$ is our semisort$_=$ algorithm. PLCR is the collect-reduce in ParlayLib [13]. All three cases are on 64-bit keys and 64-bit values.

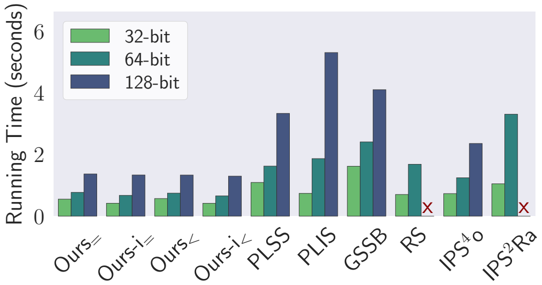


Figure 4: Running time of our semisort implementations and other implementations with different key-lengths on Zipfian-1.2. $n = 10^9$. We put crosses on RS and IPS^{2}Ra because they do not support 128-bit keys.

keys (PLIS is the only integer sort in Tab. 2 that supports 128-bit keys). On average, our algorithms are still the fastest on 32- and 128-bit keys, and the gap is smaller for 32-bit keys and larger for 128-bit keys.

One advantage of our algorithms is that they can identify heavy keys and use little further work (no local refining needed) on them. Thus, the running time of our algorithms decreases with more heavy keys (see Tab. 3). Many baseline algorithms also use optimizations on the heavy keys (e.g., PLSS), and they show a similar trend.

Parallel Scalability. We present the scalability curves using different number of threads in Fig. 3a on one representative distribution (*Zipfian*-1.2, $n = 10^9$), and for other distributions in the full version of this paper. All of our semisort algorithms, as well as PLSS, generally achieve the top-tier (almost linear) speedup, while other algorithms also scale well with increasing core counts. The self-speedup of semisort$_=$ and semisort$_<$ are 50–80×, The speedup numbers are slightly worse for semisort-i$_=$ and semisort-i$_<$(30–50× speedups), as they save the work for the hashing step but can lead to unbalanced subproblem partitioning (light buckets).

Input Size Scalability. We test all algorithms on input sizes from 10^7 to 10^9 on different distributions. A representative one (*Zipfian*-1.2 is given in Fig. 3b), and others are given in the full version of this paper. For very small test cases $n \leq 2 \times 10^7$, PLSS is the fastest on certain tests. However, in those cases, the running time is below 0.05s. For $n \geq 5 \times 10^7$, our algorithms are consistently faster than all baselines. These results indicate that our algorithms perform well on reasonably small size and scale favorably well to large inputs.

Varying Key Lengths. In addition to 64-bit keys, we also tested 32-bit and 128-bit keys for $n = 10^9$. We always set the value to be the same type as the key. Full results are given in the full version of this paper. Firstly, integer sort algorithms are sensitive to key lengths. RS and IPS^{2}Ra do not support 128-bit keys, and PLIS's performance on 128-bit keys is usually the slowest based on our testing. On 32-bit keys, integer sort algorithms can achieve much better (relative) performance than on 64-bit keys. Also, integer sort algorithms generally perform poorly on highly-skewed data (see discussion in Sec. 4.2). Other algorithms, including semisort (ours and GSSB) and comparison sort (PLSS and IPS^{4}o), are less sensitive to key lengths. Hence, the trends on 32- and 128-bit are similar to that on 64-bit. Our new algorithms generally perform well since semisort is simpler than sorting, and we can apply special optimizations (e.g., for heavy keys). In certain cases when not many special optimizations can be used (e.g., *uniform*-10^9), PLSS and IPS^{4}o perform similarly to our algorithms.

Cache Performance. To study the cache performance of our algorithm, we measured the number of cache misses of each algorithm on two representative input distributions: *uniform*-10^9 (mostly heavy keys) and *Zipfian*-1.2 (with heavy duplicates), and showed them in Tab. 4. Overall, the number of cache misses has a strong correlation of the running time, given that all these algorithms have $O(n \log n)$ or similar work. In both cases, our algorithms incur the fewest cache misses, while GSSB has the most. Between semisort$_<$ and semisort$_=$, semisort$_=$ generally has more cache misses but offset by linear work, so generally their performances are comparable.

5.2 Collect-Reduce

We test our collect-reduce algorithm (histogram is a special case for collect-reduce) and show the results on Zipfian distribution in Fig. 3c. The full results for other distributions are given in the full version of this paper. Recall that our collect-reduce algorithm is similar to semisort$_=$, but directly combines values for keys. The values of the heavy keys are combined in the *Blocked Distributing* step (no need to distribute), and the values of the light keys are combined in the *Local Refining* step. The only existing parallel implementation of collect-reduce that we know of is in ParlayLib [13] (PLCR), and we compare with it. We also show the performance

Dist.	**Param.**		**Any Type**				**Integer Only**					
			Ours$_=$	**Ours$_<$**	**PLSS**	**IPS^{4}o**	**Ours-i$_=$**	**Ours-i$_<$**	**PLIS**	**GSSB**	**RS**	**IPS^{2}Ra**
Uniform	10^9	#misses	27.1B	24.8B	28.2B	36.1B	26.9B	24.8B	30.2B	63.1B	44.6B	27.7B
		Rel. time	1.00	1.15	1.57	1.11	1.00	1.36	1.15	2.86	1.43	1.15
Zipfian	1.2	#misses	28.8B	26.7B	41.4B	52.8B	28.6B	26.7B	33.7B	76.6B	55.7B	46.6B
		Rel. time	1.00	1.01	1.95	1.58	1.01	1.00	2.71	3.69	2.55	4.97

Table 4: Number of cache misses and running time on two representative input distributions with $n = 10^9$, 64-bit keys and 64-bit values. "Dist." = distribution. "Param." = distribution parameters. "Rel. time" = relative running time normalized to the fastest one. The number of cache misses is in billion.

	n	m	n_{dist}	f_{max}	r_{heavy}	**Ours-i$_=$**	**Ours-i$_<$**	**PLSS**	**IPS^{4}o**	**PLIS**	**GSSB**	**RS**	**IPS^{2}Ra**
LJ [11]	4.85M	69.0M	4.49M	13.9K	62.8K	0.042	0.045	0.075	0.101	0.039	4.56	0.062	s.g.
TW [53]	41.7M	1.47B	35.7M	770K	74.8M	0.714	0.834	1.57	0.814	0.900	t.o.	1.06	2.94
CM [54, 74]	321M	1.61B	320M	17	0	0.791	1.04	1.84	1.10	0.903	3.58	1.09	1.44
SD [57]	89.2M	2.04B	72.8M	2.34M	456M	0.916	1.08	2.10	1.16	1.24	s.g.	1.37	2.82
Overall geometric mean						0.385	0.452	0.821	0.569	0.446	-	0.559	-

Table 5: Running time on graph transposing (in seconds). n = number of vertices. m = number of edges. n_{dist} = number of distinct keys. f_{max} = maximum frequency. r_{heavy} = ratio of keys with more than $500 \log n$ occurrences. "t.o." = did not finish in one minute. "s.g." = segmentation fault.

	n	n_{dist}	f_{max}	r_{heavy}	**Ours$_=$**	**Ours$_<$**	**PLSS**	**IPS^{4}o**
2-gram	68.0M	3.12M	2.18M	28.0%	0.312	0.332	0.346	0.753
3-gram	224M	47.5M	319K	4.43%	1.44	1.80	2.00	3.26
Overall geometric mean					0.671	0.772	0.832	1.57

Table 6: Running time on semisorting n-grams [8] (in seconds). n = number of records. n_{dist} = number of distinct keys. f_{max} = maximum frequency. r_{heavy} = ratio of keys with more than $500 \log n$ occurrences.

of semisort$_=$ as a baseline in Fig. 3c. The operator that we test for the reduce (on the values) is addition. We use Zipfian distributions with varying parameters as it smoothly covers different amounts of skew in the input. First, our collect-reduce is consistently faster than ParlayLib's implementation, and the gap is larger when the distribution is more skewed. Furthermore, when heavy keys occur more, collect-reduce is significantly faster than semisort$_=$. This is because we reduce the values for each bucket in the *Blocked Distributing* step, and then combine them without moving them. However, when few heavy keys exist, collect-reduce incurs more work than semisort, because some additional work is needed in the *Local Refining* step to pack the output since some keys are combined, while in semisort the input size equals to output size and no packing is needed. In conclusion, when the input is more skewed (more heavy keys), collect-reduce is faster than semisort$_=$, and vice versa on more evenly-distributed data (more light keys).

5.3 Applications

We integrate our algorithms into two real-world applications—graph transposing, where the input is edges, and n-grams, where the input is strings—to test our algorithm in more realistic settings. Unlike our previous experiments with synthetic distributions for performance study, here we benchmark these applications on real-world datasets and derive a more realistic understanding of semisorting performance in practice.

Graph transposing. Our first application is to transpose a directed graph $G = (V, E)$, i.e., to generate $G^\top = (V, E^\top)$, where $E^\top = \{(u, v) : (v, u) \in E\}$. This is a widely used primitive in graph algorithms. For example, parallel algorithms for strongly connected components [20, 32, 50, 67] require running reachability searches both "forwards" and "backwards". The backward reachability searches can be performed by running forward reachability query on $G^\top$. In many existing graph libraries, the edges are stored in the Compressed Sparse Row (CSR) format, where for each vertex v, the other endpoints of edges from v are stored contiguously. Thus, transposing the graph is exactly semisorting the CSR input using the other endpoint. In some existing parallel graph libraries such as Ligra [66] and GBBS [32], *stable* comparison sorts are used for graph transposing to preserve the ordering of the first endpoint.

We evaluate transpose on four real-world directed graphs, soc-LiveJournal (LJ) [11], twitter (TW) [53], Cosmo50 (CM) [54, 74], and sd_arc (SD) [57], where the largest input has 2.04 billion directed edges. For the social networks (LJ, TW) and web graph (SD), the degree distributions are more skewed. For the k-NN graph CM, the degrees are more evenly-distributed. We give more details about these datasets in Tab. 5. We use the initial CSR versions of these graphs and use our semisort$_<$ and semisort$_=$ algorithms to transpose the graphs. We compare with all the baseline algorithms and show the relative performance in Tab. 5. On all the graphs, the keys (vertex id) are 32-bit. Since the input data are integers, we use our integer version (identity hashing function).

Our semisort-i$_=$ is the fastest on three graphs (TW, CM, and SD), and is within 15% slower than the fastest on the other graph (LJ). Our semisort-i$_<$ is competitive, and is within 20% slower than the fastest on the other graphs. PLIS has relatively good performance on all graphs; it is the fastest on LJ (the smallest graph) and within 35% on the others. On the average performance across the four graphs, semisort-i$_=$ is significantly better than the others (1.15–2.13× faster). semisort-i$_<$ and PLIS have similar performance on average (within 1%). They are at least 25% faster than other implementations.

N-Gram. Our second application is to process n-grams, where an n-gram is a consecutive sequence of n items from a given sequence (e.g., text or speech). We use the 2-gram and 3-gram datasets from Wikipedia [8], and clean the data by only keeping alphabetical characters and converting them to lowercase. Each n-gram record consists of n consecutive words in the document. We use the first $n-1$ words of a record as the key, and use the last word as the value. We note that in our algorithms, we compute the hash values of the words on the fly. Semisorting n-grams can be used to identify all possible words after a given context, and to provide recommendations for text inputs, and to learn the pattern of the input sequences. Our results are shown in Tab. 6. On both 2-gram and 3-gram, our semisort$_=$ is the fastest, while semisort$_<$ (within 25% slower) is competitive. The average performance of semisort$_=$ is 15% faster than semisort$_<$, 24% faster than PLSS, and 2.3× faster than IPS^{4}o.

6 Conclusions and Future Work

In this paper, we designed flexible and high-performance algorithms for semisort and related problems. We presented two implementations, semisort$_=$ (only the equality-test is required), and semisort$_<$ (the less-than-test is also available). Compared to previous semisort algorithms, our new algorithms yield improvements in terms of space-efficiency and I/O-friendliness, ensure stability and determinism, and importantly, increase the flexibility of the interface. On different input distributions, input sizes and key lengths, our implementations achieve high performance, and outperform existing sorting and semisorting algorithms in most of the tests. For example, on 10^9 64-bit keys, on all the tested distributions, (one of) our algorithms are always the fastest among all tested algorithms, and the other one always performs similarly.

Based on our experiments, in-place versions of the sorting algorithms (e.g., IPS^{4}o) are competitive and sometimes more efficient than the non-in-place versions (e.g., PLSS). The good performance for the in-place algorithms is due to the I/O savings in the distributing step—they use the same array for both the input and the buckets (A and T in Alg. 1). We note that the new techniques proposed in this paper are independent of this distribution step. An interesting future direction is to redesign this step (e.g., borrowing ideas from IPS^{4}o) to improve the overall performance and reduce the extra space usage.

Acknowledgement

This work is supported by NSF grants CCF-2103483, IIS-2227669, NSF CAREER award CCF-2238358, and UCR Regents Faculty Fellowships. We thank anonymous reviewers for the useful feedbacks.

References

[1] U. A. Acar, D. Anderson, G. E. Blelloch, and L. Dhulipala. Parallel batch-dynamic graph connectivity. In *ACM Symposium on Parallelism in Algorithms and Architectures*, pages 381–392, 2019.

[2] U. A. Acar, D. Anderson, G. E. Blelloch, L. Dhulipala, and S. Westrick. Parallel batch-dynamic trees via change propagation. In *European Symposium on Algorithms (ESA)*, 2020.

[3] A. Aggarwal and J. S. Vitter. The Input/Output complexity of sorting and related problems. *Commun. ACM*, 31(9), 1988.

[4] K. Agrawal, J. T. Fineman, K. Lu, B. Sheridan, J. Sukha, and R. Utterback. Provably good scheduling for parallel programs that use data structures through implicit batching. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2014.

[5] Z. Ahmad, R. Chowdhury, R. Das, P. Ganapathi, A. Gregory, and M. M. Javanmard. Low-span parallel algorithms for the binary-forking model. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 22–34, 2021.

[6] D. Anderson, G. E. Blelloch, and K. Tangwongsan. Work-efficient batch-incremental minimum spanning trees with applications to the sliding-window model. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2020.

[7] N. S. Arora, R. D. Blumofe, and C. G. Plaxton. Thread scheduling for multiprogrammed multiprocessors. *Theory of Computing Systems (TOCS)*, 34(2), Apr 2001.

[8] J. Artiles and S. Sekine. Tagged and cleaned wikipedia (tc wikipedia) and its ngram. https://nlp.cs.nyu.edu/sekine/, 2008.

[9] M. Axtmann, S. Witt, D. Ferizovic, and P. Sanders. In-place parallel super scalar samplesort (ipsssso). In *European Symposium on Algorithms (ESA)*, 2017.

[10] M. Axtmann, S. Witt, D. Ferizovic, and P. Sanders. Engineering in-place (shared-memory) sorting algorithms. *ACM Transactions on Parallel Computing (TOPC)*, 9(1):1–62, 2022.

[11] L. Backstrom, D. Huttenlocher, J. Kleinberg, and X. Lan. Group formation in large social networks: membership, growth, and evolution. In *ACM International Conference on Knowledge Discovery and Data Mining (SIGKDD)*, pages 44–54, 2006.

[12] N. Ben-David, G. E. Blelloch, J. T. Fineman, P. B. Gibbons, Y. Gu, C. McGuffey, and J. Shun. Implicit decomposition for write-efficient connectivity algorithms. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2018.

[13] G. E. Blelloch, D. Anderson, and L. Dhulipala. Parlaylib — a toolkit for parallel algorithms on shared-memory multicore machines. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 507–509, 2020.

[14] G. E. Blelloch, L. Dhulipala, P. B. Gibbons, Y. Gu, C. McGuffey, and J. Shun. The read-only semi-external model. In *SIAM Symposium on Algorithmic Principles of Computer Systems (APOCS)*, pages 70–84. SIAM, 2021.

[15] G. E. Blelloch, J. T. Fineman, P. B. Gibbons, and J. Shun. Internally deterministic parallel algorithms can be fast. In *ACM Symposium on Principles and Practice of Parallel Programming (PPOPP)*, pages 181–192, 2012.

[16] G. E. Blelloch, J. T. Fineman, P. B. Gibbons, and H. V. Simhadri. Scheduling irregular parallel computations on hierarchical caches. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 355–366, 2011.

[17] G. E. Blelloch, J. T. Fineman, Y. Gu, and Y. Sun. Optimal parallel algorithms in the binary-forking model. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 89–102, 2020.

[18] G. E. Blelloch, P. B. Gibbons, and H. V. Simhadri. Low depth cache-oblivious algorithms. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2010.

[19] G. E. Blelloch, Y. Gu, J. Shun, and Y. Sun. Parallel write-efficient algorithms and data structures for computational geometry. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2018.

[20] G. E. Blelloch, Y. Gu, J. Shun, and Y. Sun. Parallelism in randomized incremental algorithms. *J. ACM*, 67(5):1–27, 2020.

[21] G. E. Blelloch, Y. Gu, J. Shun, and Y. Sun. Randomized incremental convex hull is highly parallel. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2020.

[22] G. E. Blelloch, Y. Gu, Y. Sun, and K. Tangwongsan. Parallel shortest paths using radius stepping. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 443–454, 2016.

[23] G. E. Blelloch and M. Reid-Miller. Pipelining with futures. *Theory of Computing Systems (TOCS)*, 32(3):213–239, 1999.

[24] R. D. Blumofe and C. E. Leiserson. Space-efficient scheduling of multithreaded computations. *SIAM J. on Computing*, 27(1), 1998.

[25] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms (3rd edition)*. MIT Press, 2009.

[26] NVIDIA CUB library. https://nvlabs.github.io/cub/.

[27] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. *Commun. ACM*, 2008.

[28] L. Dhulipala. *Provably Efficient and Scalable Shared-Memory Graph Processing*. PhD thesis, Carnegie Mellon University, 2020.

[29] L. Dhulipala, G. E. Blelloch, Y. Gu, and Y. Sun. PaC-trees: Supporting parallel and compressed purely-functional collections. In *ACM Conference on Programming Language Design and Implementation (PLDI)*, 2022.

[30] L. Dhulipala, G. E. Blelloch, and J. Shun. Julienne: A framework for parallel graph algorithms using work-efficient bucketing. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 293–304, 2017.

[31] L. Dhulipala, G. E. Blelloch, and J. Shun. Low-latency graph streaming using compressed purely-functional trees. In *ACM Conference on Programming Language Design and Implementation (PLDI)*, pages 918–934, 2019.

[32] L. Dhulipala, G. E. Blelloch, and J. Shun. Theoretically efficient parallel graph algorithms can be fast and scalable. *ACM Transactions on Parallel Computing (TOPC)*, 8(1):1–70, 2021.

[33] L. Dhulipala, D. Eisenstat, J. Łącki, V. Mirronki, and J. Shi. Hierarchical agglomerative graph clustering in poly-logarithmic depth. *arXiv preprint:2206.11654*, 2022.

[34] L. Dhulipala, C. Hong, and J. Shun. Connectit: a framework for static and incremental parallel graph connectivity algorithms. *Proceedings of the VLDB Endowment (PVLDB)*, 14(4):653–667, 2020.

[35] L. Dhulipala, C. McGuffey, H. Kang, Y. Gu, G. E. Blelloch, P. B. Gibbons, and J. Shun. Semi-asymmetric parallel graph algorithms for NVRAMs. *Proceedings of the VLDB Endowment (PVLDB)*, 13(9), 2020.

[36] T. Do, G. Graefe, and J. Naughton. Efficient sorting, duplicate removal, grouping, and aggregation. *ACM Transactions on Database Systems (TODS)*, 47(4):1–35, 2023.

[37] X. Dong, Y. Gu, Y. Sun, and Y. Zhang. Efficient stepping algorithms and implementations for parallel shortest paths. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 184–197, 2021.

[38] X. Dong, L. Wang, Y. Gu, and Y. Sun. Provably fast and space-efficient parallel biconnectivity. *ACM Symposium on Principles and Practice of Parallel Programming (PPOPP)*, pages 52–65, 2023.

[39] X. Dong, Y. Wu, Z. Wang, L. Dhulipala, Y. Gu, and Y. Sun. High-performance and flexible parallel algorithms for semisort and related problems. *arXiv preprint:2304.10078*, 2023.

[40] X. Dong, Y. Wu, Z. Wang, L. Dhulipala, Y. Gu, and Y. Sun. Parallel semisort and related problems implementations. https://github.com/ucrparlay/Parallel-Semisort, 2023.

[41] J. Ellert, J. Fischer, and N. Sitchinava. Lcp-aware parallel string sorting. In *European Conference on Parallel Processing (Euro-Par)*, pages 329–342. Springer, 2020.

[42] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms. In *IEEE Symposium on Foundations of Computer Science (FOCS)*, 1999.

[43] M. Goodrich, R. Jacob, and N. Sitchinava. Atomic power in forks: A super-logarithmic lower bound for implementing butterfly networks in the nonatomic binary fork-join model. In *ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 2141–2153. SIAM, 2021.

[44] Y. Gu. *Write-Efficient Algorithms*. PhD thesis, Carnegie Mellon University, 2018.

[45] Y. Gu, Z. Napier, and Y. Sun. Analysis of work-stealing and parallel cache complexity. In *SIAM Symposium on Algorithmic Principles of Computer Systems (APOCS)*, pages 46–60. SIAM, 2022.

[46] Y. Gu, Z. Napier, Y. Sun, and L. Wang. Parallel cover trees and their applications. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 259–272, 2022.

[47] Y. Gu, O. Obeya, and J. Shun. Parallel in-place algorithms: Theory and practice. In *SIAM Symposium on Algorithmic Principles of Computer Systems (APOCS)*, pages 114–128, 2021.

[48] Y. Gu, J. Shun, Y. Sun, and G. E. Blelloch. A top-down parallel semisort. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 24–34, 2015.

[49] T. Henriksen, S. Hellfritzsch, P. Sadayappan, and C. Oancea. Compiling generalized histograms for gpu. In *International Conference for High Performance Computing, Networking, Storage, and Analysis (SC)*, pages 1–14. IEEE, 2020.

[50] Y. Ji, H. Liu, and H. H. Huang. ispan: Parallel identification of strongly connected components with spanning trees. In *International Conference for High Performance Computing, Networking, Storage, and Analysis (SC)*, pages 731–742. IEEE, 2018.

[51] T. Kaler, T. B. Schardl, B. Xie, C. E. Leiserson, J. Chen, A. Pareja, and G. Kollias. Parad: A work-efficient parallel algorithm for reverse-mode automatic differentiation. In *SIAM Symposium on Algorithmic Principles of Computer Systems (APOCS)*, pages 144–158. SIAM, 2021.

[52] H. Kang, P. B. Gibbons, G. E. Blelloch, L. Dhulipala, Y. Gu, and C. McGuffey. The processing-in-memory model. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 295–306, 2021.

[53] H. Kwak, C. Lee, H. Park, and S. Moon. What is twitter, a social network or a news media? In *International World Wide Web Conference (WWW)*, pages 591–600, 2010.

[54] Y. Kwon, D. Nunley, J. P. Gardner, M. Balazinska, B. Howe, and S. Loebman. Scalable clustering algorithm for n-body simulations in a shared-nothing cluster. In *International Conference on Scientific and Statistical Database Management*, pages 132–150. Springer, 2010.

[55] Q. Liu, J. Shi, S. Yu, L. Dhulipala, and J. Shun. Parallel batch-dynamic k-core decomposition and related graph problems. *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2022.

[56] Q. C. Liu. *Scalable and Efficient Graph Algorithms and Analysis Techniques for Modern Machines*. PhD thesis, Massachusetts Institute of Technology, 2021.

[57] R. Meusel, O. Lehmberg, C. Bizer, and S. Vigna. Web data commons — hyperlink graphs. http://webdatacommons.org/hyperlinkgraph, 2014.

[58] I. Müller, P. Sanders, A. Lacurie, W. Lehner, and F. Färber. Cache-efficient aggregation: Hashing is sorting. In *ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 1123–1136, 2015.

[59] O. Obeya, E. Kahssay, E. Fan, and J. Shun. Theoretically-efficient and practical parallel in-place radix sorting. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 213–224, 2019.

[60] J. Qiu, L. Dhulipala, J. Tang, R. Peng, and C. Wang. Lightne: A lightweight graph processing system for network embedding. In *ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 2281–2289, 2021.

[61] P. Sanders and S. Winkel. Super scalar sample sort. In *European Symposium on Algorithms (ESA)*, pages 784–796. Springer, 2004.

[62] Z. Shen, Z. Wan, Y. Gu, and Y. Sun. Many sequential iterative algorithms can be parallel and (nearly) work-efficient. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2022.

[63] J. Shi, L. Dhulipala, and J. Shun. Parallel clique counting and peeling algorithms. pages 135–146. SIAM, 2021.

[64] J. Shi and J. Shun. Parallel algorithms for butterfly computations. In *SIAM Symposium on Algorithmic Principles of Computer Systems (APOCS)*, pages 16–30. SIAM, 2020.

[65] J. Shun. Practical parallel hypergraph algorithms. In *ACM Symposium on Principles and Practice of Parallel Programming (PPOPP)*, pages 232–249, 2020.

[66] J. Shun and G. E. Blelloch. Ligra: A lightweight graph processing framework for shared memory. In *ACM Symposium on Principles and Practice of Parallel Programming (PPOPP)*, pages 135–146, 2013.

[67] G. M. Slota, S. Rajamanickam, and K. Madduri. Bfs and coloring-based parallel algorithms for strongly connected components and related problems. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 550–559. IEEE, 2014.

[68] Y. Sun. *Join-based Parallel Balanced Binary Trees*. PhD thesis, Carnegie Mellon University, 2019.

[69] K. Tangwongsan and S. Tirthapura. Parallel streaming random sampling. In *European Conference on Parallel Processing (Euro-Par)*, pages 451–465. Springer, 2019.

[70] T. Tseng, L. Dhulipala, and G. Blelloch. Batch-parallel euler tour trees. In *2019 Proceedings of the Twenty-First Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 92–106. SIAM, 2019.

[71] L. G. Valiant. General purpose parallel architectures. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science (Vol. A)*, pages 943–973. MIT Press, 1990.

[72] L. Wang, X. Dong, Y. Gu, and Y. Sun. Parallel strong connectivity based on faster reachability. In *ACM SIGMOD International Conference on Management of Data (SIGMOD)*, 2023.

[73] Y. Wang, Y. Gu, and J. Shun. Theoretically-efficient and practical parallel dbscan. In *ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 2555–2571, 2020.

[74] Y. Wang, S. Yu, L. Dhulipala, Y. Gu, and J. Shun. Geograph: A framework for graph processing on geometric data. *ACM SIGOPS Operating Systems Review*, 55(1):38–46, 2021.

[75] Y. Wang, S. Yu, Y. Gu, and J. Shun. Fast parallel algorithms for euclidean minimum spanning tree and hierarchical spatial clustering. In *ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 1982–1995, 2021.

[76] Y. Xu, K. Singer, and I.-T. A. Lee. Parallel determinacy race detection for futures. In *ACM Symposium on Principles and Practice of Parallel Programming (PPOPP)*, pages 217–231, 2020.

[77] Y. Xu, A. Zhou, G. Q. Yin, K. Agrawal, I.-T. A. Lee, and T. B. Schardl. Efficient access history for race detection. In *Algorithm Engineering and Experiments (ALENEX)*, pages 117–130. SIAM, 2022.

[78] W. Yang, V. Harsh, and E. Solomonik. Optimal round and sample-size complexity for partitioning in parallel sorting. *arXiv preprint:2204.04599*, 2022.

[79] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster Computing with Working Sets. In *USENIX Workshop on Hot Topics in Cloud Computing (HotCloud)*, 2010.