Parallel Integer Sort: Theory and Practice

Xiaojun Dong
UC Riverside
xdong038@ucr.edu

Laxman Dhulipala
University of Maryland
laxman@umd.edu

Yan Gu
UC Riverside
ygu@cs.ucr.edu

Yihan Sun
UC Riverside
yihans@cs.ucr.edu

Abstract

Integer sorting is a fundamental problem in computer science. This paper studies parallel integer sort both in theory and in practice. In theory, we show tighter bounds for a class of existing practical integer sort algorithms, which provides a solid theoretical foundation for their widespread usage in practice and strong performance. In practice, we design a new integer sorting algorithm, DovetailSort, that is theoretically-efficient and has good practical performance.

In particular, DovetailSort overcomes a common challenge in existing parallel integer sorting algorithms, which is the difficulty of detecting and taking advantage of duplicate keys. The key insight in DovetailSort is to combine algorithmic ideas from both integer- and comparison-sorting algorithms. In our experiments, DovetailSort achieves competitive or better performance than existing state-of-the-art parallel integer and comparison sorting algorithms on various synthetic and real-world datasets.

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

Keywords: Integer Sort, Radix Sort, Parallel Algorithms

1 Introduction

Sorting is one of the most widely-used primitives in algorithm design, and has been extensively studied. For many if not most cases, the keys to be sorted are fixed-length integers. Sorting integer keys is referred to as the ***integer sort*** problem. An integer sorting algorithm takes as input n records with integer keys in the range 0 to $r - 1$, and outputs the records with keys in non-decreasing order. Despite decades of effort in studying integer sorting algorithms, however, obtaining *parallel integer sorting (IS) algorithms that are efficient both in theory and in practice* has remained elusive.

Theoretical Challenges. *As a special type of sorting, integer sorting algorithms **can** outperform comparison sorting algorithms by using the integer encoding of keys.* This claim

is verified in many existing studies [5, 43] as well as our experiments (see Fig. 1). As a result, in real-world applications (e.g., [19, 51, 54]), integer sort is usually preferred (instead of comparison sort) when the keys are integers. While it is not surprising that IS algorithms can outperform comparison sorts, we observe a *significant gap in connecting the high performance with theory*. Theoretical parallel IS algorithms with good bounds [2, 3, 7, 31, 41] are quite complicated, and we are unaware of any implementations of them. On the other hand, for the practical parallel IS solutions [5, 9, 43], we are unaware of "meaningful" analysis to explain their good performance for general r. The best-known bounds for them, as discussed in [43], are $O(n \log r)$ work (number of operations) and polylog(rn) span (longest dependence chain). However, note that the main use case for integer sort is when $r = \Omega(n)$, since otherwise the simpler counting sort [45, 53] can be used. In this case, the bounds for practical parallel IS algorithms are no better than comparison sorts ($O(n \log n)$ work and polylogarithmic span [9, 12, 13, 24]). This leads to the following open question in theory: do the practical parallel IS algorithms indeed have lower asymptotic costs than comparison sort (and if so, under what circumstances)?

Practical Challenges. *As a special type of sorting, integer sorting algorithms **should** outperform comparison sorting algorithms by using the integer encoding of keys.* Since integers are comparable, comparison sort can be considered as a baseline for sorting integers. Unfortunately, SOTA parallel IS algorithms do not consistently outperform comparison-based sorting algorithms. One key reason is the inherent difficulty of dealing with *duplicate keys* in integer sorts. In principle, duplicate keys are beneficial for sorting algorithms. For example, *samplesort* can skip a recursive subproblem between two equal pivots; similarly, quicksort can separate keys equal to the pivot to avoid further processing them. Interestingly, such a case does not apply to integer sort. Existing parallel IS implementations follow the *most-significant digit (MSD)* framework that partitions all keys into buckets based on the *integer encoding* (i.e., 8–12 highest bits), and recurses within each bucket. As such, equal keys cannot be detected until the last recursion. Although some techniques can be used to detect special distributions (e.g., all keys are the same), to the best of our knowledge, no existing parallel IS implementation can benefit from duplicate keys in a general (provable) and non-trivial manner[1]. Fig. 1 and Tab. 3

---

[1] Some techniques in existing IS implementations [5, 43] have a side-effect to benefit from duplicates in some cases, at a cost of making the algorithms *unstable*. We want to overcome this issue without sacrificing stability.
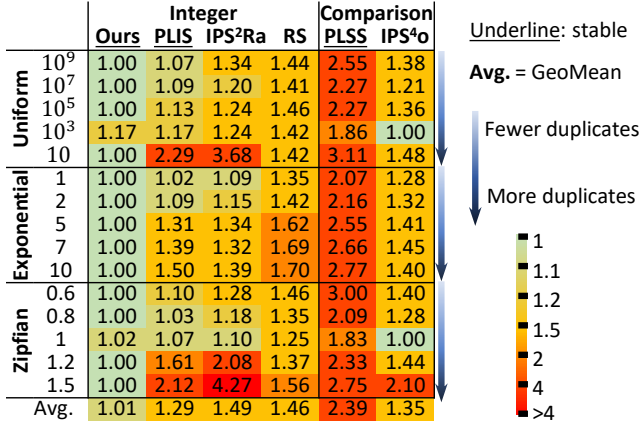
Figure 1. Heatmap to compare sorting algorithms on 10^9 **records with 32-bit keys and 32-bit values.** All numbers are **running times relative to the best** for each input. Raw data are in Tab. 3. The baseline algorithms are described in Tab. 2.

show that on inputs with heavy duplicates, SOTA parallel IS algorithms can be slower than comparison sorts.

Our Contributions. In this paper, we study parallel integer sort to overcome the aforementioned challenges both in theory and in practice. In theory, *we answer the open question by proving that **a class of** parallel IS algorithms, including DovetailSort proposed in this paper, have* $O(n\sqrt{\log r})$ *work for input size* n *and key range* 0 *to* $r - 1$. This is asymptotically better than comparison sorts for a large key range $r = n^{o(\log n)}$. This explains why the existing parallel IS algorithms outperform comparison sorts in practice in many cases. We also show that we can achieve $\text{polylog}(nr)$ span *whp* for an unstable integer sort, and $\tilde{O}(2^{\sqrt{\log r}})$ span for a class of practical stable parallel IS algorithms, such as our new DovetailSort and the integer sort in ParlayLib [9]. We also prove that DovetailSort has $\Theta(n)$ work for some special input distributions with heavy duplicates. In addition to the new bounds, our theoretical results (Sec. 4) also explain the choices of parameters for many (if not all) practical parallel IS implementations. We believe that our theoretical contribution is important both for analyzing our own algorithm and for understanding parallel integer sort as a general algorithmic problem.

In practice, *we propose DovetailSort (DTSort), a parallel integer sorting algorithm that is **efficient and stable***. DTSort combines the algorithmic ideas from both integer- and comparison-sorting algorithms. The goal is to detect and take advantage of heavily duplicate keys (**heavy keys**) as in samplesort (a commonly-used comparison sort), but preserve the low asymptotic work bound as in integer sort. The key idea are to 1) use sampling to identify heavy keys, separate them from the other keys (called **light keys**), and skip them in the recursive calls, and 2) exploit the bitwise encoding of the keys to split light keys into subproblems and deal with them recursively. Although the high-level idea of *separating* heavy and light keys is natural, an interesting challenge is that all keys need to be put in increasing order in the output,

	$A[1..n]$	Original input array with size n
	$[r]$	The range of the integer keys $0, \ldots, r - 1$
	γ	Number of bits in a "digit" (sorted by each level)
	$b = 2^\gamma$	The "radix" size
	θ	Base case size threshold
	n'	Current (recursive) problem size
	d	Number of remaining "digits" to be sorted

Table 1. Notations and parameters in this paper.

and thus the heavy and light keys have to be finally *interleaved* with each other. Therefore, the heavy-light separation idea requires ***dovetailing*** the keys at a low cost. To do this, DTSort first carefully puts the heavy keys close to their final destinations when they are identified. After the light keys have been sorted from the recursive calls, DTSort uses a dedicated algorithm to combine them into the final sorted order. We present more details in Sec. 3.

Due to the algorithmic improvements, DTSort achieves consistently strong performance on various synthetic and real-world input instances. We evaluate DTSort using SOTA parallel sorting algorithms as baselines. We illustrate the results on 32-bit keys in Fig. 1, and present more in Sec. 6. Compared to the best baseline on 32-bit keys (PLIS), our algorithm is competitive (1.02× faster) for no or light duplicate keys, and can be up to 2.3× faster with more duplicates. The advantage of DTSort is larger on 64-bit keys (see Tab. 3). We also evaluate DTSort in several applications with real-world input data. In tens of the input instances we tested, DTSort is the fastest in most cases. We note that sorting is one of the most well-studied problems, and it is almost impossible for one algorithm to be the best for *all* input instances. The goal is to design algorithms that are always close to best in any use case. We believe DTSort achieves very good overall performance on a variety of input patterns. Our code is publicly available [20]. We present more results and analyses in the full version of this paper [21].

2 Preliminaries

2.1 Notations. We use $O(f(n))$ *with high probability* (*whp*) in n to mean $O(cf(n))$ with probability at least $1 - n^{-c}$ for $c \geq 1$. Throughout the paper, we consider the high probability bound in n for input size n. We omit "in n" with clear context. We use $[r]$ to refer to integers in the range 0 to $r - 1$. In our analysis, we use the term $2^{\sqrt{\log r}}$, which satisfies $\omega(\log^{c_1} r) = 2^{\sqrt{\log r}} = o(r^{c_2})$ for any positive constants c_1 and c_2. We use $\tilde{O}(f(nr))$ to denote $O(f(nr) \cdot \text{polylog}(nr))$. We use $\log n$ to represent $\log_2 n$. More notations are presented in Tab. 1.

2.2 Computational Models. We use the work-span (or work-depth) model for fork-join parallelism with binary forking to analyze parallel algorithms [12, 18]. 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 in practice in $W/P + O(D)$ time *whp* on P processors [4, 16, 28].

2.3 Sorting and Integer Sorting.
Integer sort takes as input an array A of n records with a key function that maps each record to an integer key in range $[r]$. The goal is to reorder the records to be in non-decreasing order. As mentioned in Sec. 1, the main and most general use scenario of integer sorts is when $r = \Omega(n)$, so we assume this in our theoretical analysis unless otherwise specified. Many existing integer sort algorithms consider the integer keys as radix-b (base-b) numbers, and work on each "digit" in a round. Usually each digit refers to every γ bits in the integer and $b = 2^\gamma$. These algorithms are also referred to as radix sort. The classic sequential radix sort uses $b = \Theta(n)$, starts from the least significant digits (LSD), and reorders all the records on every digit using a stable sort, until all bits are used up [18]. It takes $O(n \log_n r)$ work.

Unlike the sequential setting, existing parallel integer sort implementations start from the most significant digit (MSD). The algorithms (see Alg. 1) have two steps: 1) *distributing* all records into $b = 2^\gamma$ buckets by their MSD, and 2) *recursing* to sort all buckets in parallel. This top-down approach allows all subproblems to be solved independently in parallel, which is more I/O friendly. The distribution step can be performed by a counting sort (see Sec. 2.4) with the bucket id as the MSD. Most practical parallel integer sorting algorithms [5, 9, 15, 17, 35, 39, 43, 44, 50, 57, 59] follow the framework in Alg. 1 and have good performance. However, if a tight bound exists for these parallel MSD algorithms remains open.

For simplicity, we assume to store the sorted result in the input array ("in-place"). Here "in-place" refers to the algorithm *interface*. Our algorithm still uses $O(n)$ extra space. A sorting algorithm is **stable** if the output preserves the relative order among equal keys as in the input, and unstable otherwise. Stability is required in many sorting applications.

2.4 Counting Sort (aka. Distribution).
Counting sort is a commonly-used primitive in parallel sorting algorithms to distribute records into (a small number of) buckets. It takes an input of n' records with a function to map each record to an integer (the **bucket id**) in $[r']$ ($r' \leq n'$). The goal is to sort the records by the bucket ids. The classic stable counting sort [53], which is widely used in practice, has $O(n')$ work and $O(r' + \log n')$ span. Due to page limits, we put more discussions about this algorithm in the full paper [21], but the audience can treat the counting sort as a black box.

There exists a randomized algorithm for unstable counting sort on $n' = \Omega(\log^2 n)$ records with $r' = O(n' \log^{O(1)} n)$ buckets in $O(n' + r')$ work and $O(\log n)$ span *whp* in n [45]. However, this approach is complicated and is less frequently used in practice (discussions also given in the full paper [21]).

Algorithm 1: Parallel-MSD-Sort($A[0..n-1], d$)

Input: A: input array with keys in $[r]$. d: number of digits remained to be sorted. Each digit refers to γ bits in the key. On the root-level $d = (\log r)/\gamma$. θ: base case threshold.

Notes: We use the d-th digit as the d-th digit from low to high.

1 **if** $d = 0$ **then** **return** A *// All bits are sorted*

2 **if** $|A| \leq \theta$ **then** **return** *ComparisonSort*(A)   *// Base case*

3 *Distributing:* Apply counting sort on A using the d-th digit of the key as bucket id

4 *Recursing:* For $i \in [2^\gamma]$, let A_i be the chunk in A where the records have the d-th digit as i

5 **parallel_for** $i \leftarrow 0$ *to* $2^\gamma - 1$ **do** Parallel-MSD-Sort($A_i, d-1$)

2.5 Comparison Sort, Semisort, and Sampling.
Efficient and parallel sorting is widely studied [8, 32, 34, 46–48, 52]. Comparison sort and semisort are the other two types of sorting problems. Comparison sort takes an input of n records and a comparison function "$<$" to order two given records, and outputs the records in non-decreasing order. Semisort takes an input of n records and an equality-test "$=$" on two records, and reorders the records such that records with equal keys are adjacent in the output [22]. SOTA parallel algorithms for comparison sort [5, 9] and semisort [22, 30] all use sampling that can detect heavy duplicate keys.

The sampling scheme used in these papers was first proposed in [45]. Given a parameter p and input size n, the sampling scheme first selects $p \log n$ samples uniformly at random. It then sorts the samples, subsampling every $(\log n)$-th element into a list S'. Based on Chernoff bounds, any key that appears more than once in S' must have $\Omega(n/p)$ occurrences in the input *whp*. In existing sorting/semisorting algorithms [5, 9, 22, 30], each such key will be placed in its own bucket. Since all records in this bucket have the same key, no further sorting is needed. As such, these algorithms can be significantly faster when the input contains many duplicate keys. Our DTSort is inspired by this technique for detecting heavy duplicate keys. In Sec. 3, we will discuss how to apply this sampling scheme to an integer sort algorithm.

3 The DovetailSort Algorithm

We now present DovetailSort (DTSort), which follows the MSD sort framework in Alg. 1. By a careful design, DTSort can efficiently handle heavy duplicate keys, and is **stable**.

To detect and handle duplicate keys, we use the sampling idea in existing comparison-based sorts. However, we need to carefully combine this approach with the MSD sort to avoid blowing up the work bound to $O(n \log n)$. As mentioned, we want our algorithm to separate heavy and light keys, with the goal of avoiding sorting heavy keys in all recursive levels. The idea of identifying heavy/light buckets has been used in parallel semisort [22, 30]. However, integrating it into a parallel MSD sort is highly non-trivial. Unlike in semisort where heavy and light keys are completely independent and can be placed in any order in the output, all buckets in an
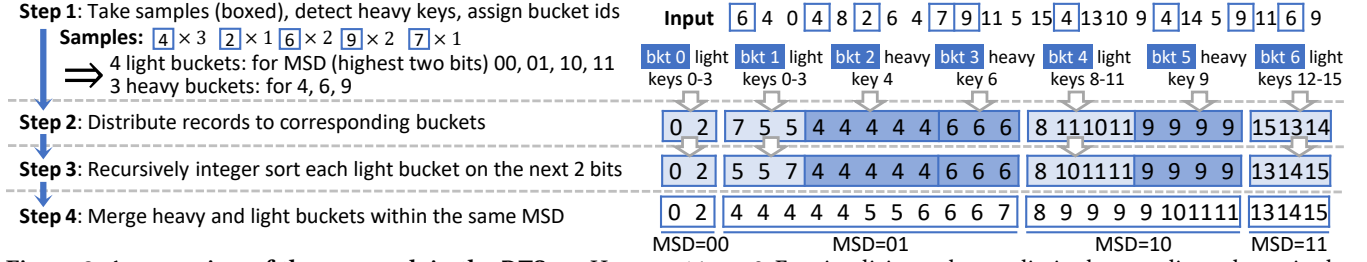
Xiaojun Dong, Laxman Dhulipala, Yan Gu, and Yihan Sun

Figure 2. An overview of the approach in the DTSort. Here $r = 16$, $\gamma = 2$. For simplicity and space limit, the sampling scheme in the figure is not exactly accurate as described in the algorithm. Here we simply set keys with 2 or more samples as heavy keys.

Algorithm 2: DTSort(A, d)

Input: A: input array with keys in $[r]$. d: number of digits remained to be sorted. Each digit refers to $\gamma = \sqrt{\log r}$ bits in the key. On the root-level $d = (\log r)/\gamma = \sqrt{\log r}$.

Notes: We use "the d-th digit" as the d-th digit from low to high.

1 **if** $d = 0$ **then return** A *// All bits are sorted*
2 **if** $|A| \le 2^{2\gamma}$ **then return** *ComparisonSort*(A)   *// Base cases*

Sampling (find heavy keys & assign buckets):

3 $S \leftarrow \Theta(2^\gamma \log n)$ sampled keys from A
4 Sort S, subsample every $(\log n)$-th key, and store the keys that have more than one subsamples into S'
5 **for** $i \leftarrow 0$ *to* $|S'| - 1$ **do**   *// each heavy keys' MSD and its key*
6    $h[i] \leftarrow \langle$the d-th digit in $S'[i]$, $S'[i]\rangle$
7 **for** $i \leftarrow 0$ *to* $2^\gamma - 1$ **do** *// each light bucket's MSD and a dummy key*
8    $l[i] \leftarrow \langle i, -1 \rangle$
9 Merge h and l into an array *buckets*
10 Initialize the hash table H and lookup table L
11 **for** $i \leftarrow 0$ *to* $|buckets|$ **do**   *// assign bucket ids in order*
12    $\langle x, y \rangle \leftarrow buckets[i]$
13   **if** $y = -1$ **then** $L[x] \leftarrow i$ *// light bucket and its id*
14   **else** Insert key y with value i to H *// heavy bucket and its id*

Distributing (Reorder A by bucket ids):

15 Use counting sort on A with key function GetBucketID (line 21)

Recursing (sort light buckets):

16 **parallel_for_each** *light bucket B* **do** DTSort($B, d - 1$)

Dovetail Merging (interleave light and heavy buckets):

17 **parallel_for** $i \in [2^\gamma]$ **do**   *// merge buckets for all MSD zones*
18   Let $B_0, B_1, \ldots B_m$ be all buckets in MSD zone i
19   DTMerge($B_0, B_1, \ldots B_m$)
20 **return** A

21 **Function** GetBucketId(k)
22   **if** k *is found in H* **then return** $H[k]$ **else return** $L[k]$

---

integer sort must be finally placed in an increasing order. We refer to this step as the "*dovetail merging*" step, and discuss an efficient solution both in theory and in practice in Sec. 3.4.

We present DTSort in Alg. 2 and an illustration in Fig. 2. It consists of four steps, explained in the subsections below. It is an MSD sort using $\gamma = \sqrt{\log r}$ bits in each digit. We explain how this parameter enables a good work bound in Sec. 4. We use n as the original input size, and n' as the current recursive subproblem size. Separating them is needed to

rigorously analyze the algorithm and give the correct high probability bounds. We use *heavy/light bucket* to mean a bucket for heavy/light keys, and *heavy/light record* to mean a record with a heavy/light key. We use "MSD" to refer to the MSD in the current subproblem (see the d-th digit in Alg. 2).

3.1 Step 1: *Sampling*

As mentioned earlier, the first step is to use sampling to determine heavy keys, and assign a bucket id to each heavy and light bucket. The theory of this idea is developed in previous work on samplesort and semisort [30, 45].

In this step, $\Theta(2^\gamma \log n)$ samples are picked into an array S. Recall that γ is the number of bits to be sorted in a level. We then subsample every $(\log n)$-th key in S, and output keys with more than one subsample to S'. As discussed in Sec. 2.5, we define all keys in S' as heavy keys, and all other keys are light keys. According to the MSD (the highest γ bits) of the keys, the key range is divided into $b = 2^\gamma$ subranges. We call each key range sharing the same MSD an **MSD zone**. In a regular integer sort, each MSD zone forms a bucket. In our case, all *light keys* within the same MSD zone will share a bucket, but each *heavy key* has its own bucket.

Unlike samplesort and semisort that do not need to further handle heavy keys once they are recognized and distributed, in integer sort, we need to finally combine them with light keys in the same MSD zone. Hence, we wish to keep the heavy keys close to their final destination in the output from this very first step, in order to perform an efficient merge at the end of the algorithm. To do so, we will set the bucket id of a heavy bucket immediately after the light bucket which it should finally be integrated into. Namely, for each MSD zone, there is exactly one light bucket followed by some (possible zero) heavy buckets, which are all the heavy keys with the same MSD as this zone. An illustration is given in Fig. 2. A unique id is assigned to each bucket in a serial order. Multiple heavy buckets in the same MSD zone will be totally ordered by their full keys. In this way, the light buckets and heavy buckets in the same MSD zone are consecutive, which makes it easier to combine them in Step 4.

We explain this step in detail on line 3–14 in Alg. 2. To assign the bucket ids, we first obtain an array h of all heavy keys in a sorted order (line 6), which is a pair (z, k), where k is the key itself and z is the MSD zone of k. We then use an array l (line 8) to maintain a pair (z, k) for each light bucket

(similar to h), where $z \in [2^\gamma]$ is the MSD of the light bucket, and $k = -1$ is a dummy key to mark this is a light bucket. By merging h and l, all buckets will be re-organized in the desired order—all buckets will first be ordered by their MSD; within the same MSD, the first bucket is always the light one, followed by all heavy buckets sorted by their key. We can then assign ids to them serially.

After assigning the bucket ids, we build a hash map H to maintain the bucket id of the heavy keys, and use a lookup array L to map every MSD to its associated light bucket id. As mentioned, the total number of buckets is small, so the cost to maintain them is small both in time and space. With H and L, we can determine the bucket id for any key k in constant time (the GetBucketID function in Alg. 2): we first look up whether k is in H; if so, k is a heavy key, and its bucket id can be found in H. Otherwise, k is a light key, and we can look up its bucket id in L by using the MSD of k.

3.2 Step 2: *Distributing*

After the first step, we know the bucket id of all keys. In this step, we use the stable counting sort in Sec. 2.4 with their bucket id as the key, similar to many existing parallel sorting algorithms [9, 22], After this step, all records are stably reordered into the corresponding buckets.

3.3 Step 3: *Recursing*

After Step 2, keys are sorted across MSD zones, but keys in the same MSD zone are not necessarily sorted. In this step, we first sort each light bucket recursively.

As with a regular integer sort, the recursive call deals with the subproblem using the same integer sort routine on the next "digit". In DTSort, since all records in a heavy bucket have the same key, no further sorting is needed. Hence, we only recursively sort the light buckets (line 16).

Note that by using a recursive approach, we can potentially detect much more than 2^γ heavy keys if we consider all levels of recursion. Namely, the heaviest keys will be identified at the topmost level and bypass all recursive calls, and some medium-level heavy keys will be identified in the middle levels but still bypass a few recursive calls. This enables a balance in finding more heavy keys *in total* to save work, as well as to use a limited number of heavy keys *in each subproblem* to avoid a large amount of extra space in the counting sort in Step 2 (see a discussion in [22]).

3.4 Step 4: *Dovetail Merging*

After Step 2, all keys are sorted across MSD zones, and all heavy keys are ordered by their keys. Step 3 further sorts all light keys within each MSD zone. To compute the final output, we finally must merge the two sorted sequences (light records and heavy records) in each MSD zone. We refer to this step as the *dovetail merging* step, which is the DTMerge function on line 19 in Alg. 2.

Each DTMerge call deals with an MSD zone with $m + 1$ buckets $B_{0..m}$ stored consecutively in array A. The first one B_0
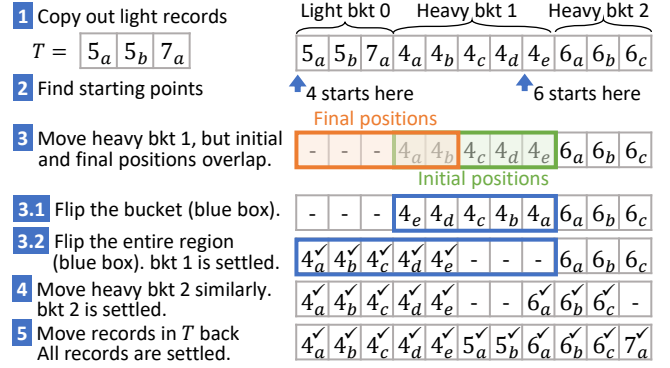

Figure 3. Illustration of the dovetail merging step. The example merges the buckets in MSD zone 01 in Fig. 2. We use a letter as subscription to distinguish different records with the same key.

is the light bucket in this zone. The rest of the m buckets are heavy buckets. Given these special properties, the problem is equivalent to viewing the m heavy buckets as partitioning the light bucket into $m+1$ blocks, and then to interleave them. Hence, we call this process *dovetail merging*. A naïve solution is to call a standard parallel merge to combine B_0 with the rest $B_{1..m}$, with linear work and polylogarithmic span. Despite the efficiency in theory, in practice, it requires two rounds of out-of-place global data movements—*all* records have to be merged into a temporary array, and then be copied back to the input array. Such data movements will cause more cache misses and may make the algorithm inefficient. There exists an in-place parallel merge algorithm in theory [29], but it incurs a large constant hidden in the asymptotic cost (moving each record at most four times), which may also be expensive in practice. To enable good performance, our goal is to reduce the data movements, with the additional consideration of moving less data out of the input array.

Our idea is based on two observations. First, being strictly in-place is unnecessary, since $O(n)$ temporary space is required by the counting sort in Step 2. Therefore, we can reuse the space while minimizing the number of out-of-place memory access. The second useful observation is that the number of heavy keys is guaranteed to be small (at most 2^γ of them). As a result, to achieve a practical solution, our idea is to process buckets in the same MSD zone one by one *sequentially* (note that all MSD zones are still processed in parallel), and aim to achieve *high parallelism within each bucket*.

With these insights, we present our DTMerge algorithm with an illustration in Fig. 3 and pseudocode in Alg. 3. The high-level idea is to only copy the smaller one of the light bucket (B_0) or the heavy buckets ($B_{1..m}$) out to a temporary array, so that we copy at most half the records. The larger half will be handled within the input array. WLOG, assume the current subproblem has more heavy records than light ones (line 2), and the other case can be handled symmetrically (line 13). In this case, we first move the light records to a temporary array T (line 3), and then move all heavy buckets to their final positions. To obtain the final positions

of each heavy bucket, we first binary search all heavy keys (in parallel) in the light bucket B_0 (line 1). Combining this information with the sizes of all buckets, we can infer the starting point of each heavy key. The next step is to (stably) move the entire bucket B_i to this starting point, and do so for all buckets one by one from B_1.

Ideally, we wish to copy all records in B_i to their final positions in parallel, but have to be careful to ensure that their final positions are "safe" to be overwritten. Since all heavy buckets are initially at the end of this MSD zone, their final destinations must be ahead of their initial positions. There are three cases for a bucket B_i's destination. The first case is when it contains light records, which is safe since all light records have been backed up in T. The second is when it overlaps with an earlier heavy bucket $B_{i'}$ where $i' < i$. It is also safe—since $i' < i$, bucket i' must have been moved to its final destination. In the first two cases, we can safely move the records directly into the final destination in parallel (line 10). The last and the most interesting case is when the destination of the bucket overlaps with its own initial position (line 5). Our idea is inspired by the in-place *circular shift* algorithm [25, 58] (see Phase 3 in Fig. 3): we can first flip (reverse the order) the bucket itself (line 7), and then flip the entire associated region in the array (line 8). Flipping an array can be performed in parallel and in-place by swapping the corresponding records. As such, the entire chunk of B_i is moved to an earlier position in a *stable* way. After dealing with all heavy buckets, we copy all records in T back to A, all in parallel, directly to their final destination (line 12). In this way, our idea incurs out-of-place copies for at most half the records, and requires moving the remaining records at most twice, but entirely within A. Within each bucket, all operations (direct copy or swapping records) are highly parallel.

Note that although buckets within the same MSD zone are processed sequentially, this only increases the span of the entire algorithm by a factor of $\log n$ because there are only $O(2^\gamma)$ heavy buckets. Interestingly, although a parallel merge gives better span bound than our DTMerge, even in theory it is not necessary, because the counting sort in Step 2 also requires $O(2^\gamma)$ span. We present more theoretical analysis in Thm. 4.5.

In Sec. 6.3, we experimentally study DTMerge and show it accelerates Step 4 compared to using a parallel merge. We believe the "dovetail" step is a unique challenge arising from detecting heavy keys in integer sorts, and our new technique solves it in an effective yet simple way.

3.5 Base Cases

We use two base case conditions. The first is when all digits have been sorted (line 1). The second is when the problem size is smaller than $2^{2\gamma}$ (line 2), where we apply a comparison sort. Using comparison sort as a base case is standard for parallel integer sort implementations. Interestingly, instead

---

Algorithm 3: DTMerge($A, B_0, B_1, \ldots B_m$)

Input: $m + 1$ buckets consecutive in array A. The first one B_0 is a sorted light bucket. The other m buckets are heavy buckets sorted by their keys.

1   Binary search all heavy keys in B_0, accordingly get $p_{1..m}$, where p_m is the starting index for heavy bucket B_i when the array is fully sorted.

2   **if** $|B_0| \leq \sum_{i=1}^m |B_i|$ **then**   *// more heavy keys than light*

3   copy B_0 to array T

4   **for** $i \leftarrow 1$ *to* m **do**   *// starting from the first heavy bucket*
        // Move B_i to the final positions starting from $A[p_i]$

5   **if** *The final positions for B_i overlap with the current positions of B_i* **then**

6   Let s be the current start point of B_i

7   flip($A, s, s + |B_i|$)   *// Flip the entire region*

8   flip($A, p_i, s + |B_i|$)   *// Flip the destination region*

9   **else**   *// No overlap. Directly copy*

10   **ParallelForEach** $j \leftarrow 0$ *to* $|B_i| - 1$ **do**

11   Copy the j-th record in B_i to $A[p_i + j]$

12   Move light records from T to their final positions in A

13   **else** Symmetric case: copy heavy keys out and merge

---

of viewing it as an empirical optimization, our theoretical analysis indicates that this base case, as well as the threshold to use this base case, are very important to guarantee theoretical efficiency. We present more details in Sec. 4.

4 The Theory of Parallel MSD Sort

In this section, we provide an analysis for *a class of MSD integer sorting algorithms* based on the framework in Alg. 1, including DTSort. Recall that the algorithm sorts a "digit" of γ bits in each round by 1) distributing all records to the corresponding buckets by the current digit being sorted, and 2) sorting each bucket recursively. Finally, a base case of using comparison sort is applied to subproblems with sizes smaller than a parameter θ. As discussed in Sec. 1 and 2, although this parallel MSD sort is widely used in practice, we are unaware of any tight analysis of it. To the best of our knowledge, the best work bound is $O(n \log r)$ [43], which is no better than a comparison sort with $O(n \log n)$ work for the realistic setting when $r = \Omega(n)$. This is inconsistent with the excellent performance of existing parallel IS algorithms in practice, which are much faster than SOTA comparison-based sorts for many inputs. In this paper, we bridge this gap between theory and practice by showing Thm. 4.1, 4.4 and 4.5. Recall that the main use case for integer sorts is when $r = \Omega(n)$ (otherwise a counting sort has a better cost bound). We hence make this assumption on r in our analysis.

4.1 The Analysis of the General MSD Sort

As a warmup, and to illustrate some of the key theoretical ideas, we start by showing that we can achieve an *unstable* MSD sort with low work and span.

Theorem 4.1. *There exists an unstable parallel MSD sorting algorithm with $O(n\sqrt{\log r})$ work and $O(\log r + \sqrt{\log r}\log n)$ span whp.*

Under the assumption of the standard RAM model that can handle integers with $O(\log n)$ bits (i.e., $r = n^{O(1)}$), the theorem implies $O(n\sqrt{\log n})$ work and $O(\log^{1.5} n)$ span *whp*.

The key in our analysis is to properly set up the parameters in Alg. 1. Once we do so, the analysis becomes relatively clean. The parameters to use are digit width $\gamma = \sqrt{\log r}$ and base case threshold $\theta = 2^{c\gamma}$ for some constant $c \geq 1$. This indicates $(\log r)/\gamma = O(\sqrt{\log r}) = O(\gamma)$ recursive levels in the entire algorithm. We use the *unstable counting sort* [45] (see Sec. 2.4) for the distribution step, and use comparison sort with $O(n'\log n')$ work and $O(\log n')$ span [24] to handle base cases with size n'. The two main parts in Alg. 1 are the base cases (line 2) and the distribution step (line 3). We first analyze their work with the parameters shown above.

Lemma 4.2. *With the chosen parameters, the total work for base cases (line 2) in Alg. 1 is $O(n\sqrt{\log r})$.*

Proof. Let $n_1, \cdots, n_k$ be the base case sizes, where $\sum n_i = n$. Due to the base case condition, $n_i \leq 2^{c\gamma}$, so the work to sort it is $O(n_i \log n_i) = O(n_i c\gamma)$. The total work for all base cases is therefore $O(nc\gamma) = O(n\sqrt{\log r})$ (c is a constant). □

Lemma 4.3. *With the chosen parameters, the distribution step (line 3) Alg. 1 in all recursive calls has work $O(n\sqrt{\log r})$ whp.*

Proof. We start with the total work in each recursive level (i.e., those with the same parameter d). Assume all subproblems in this level have sizes $n_1, \cdots, n_k$, where $\sum n_i \leq n$. Using the unstable counting sort [45], the distribution work is $\sum O(n_i + 2^\gamma)$ *whp* for each subproblem. Due to the base case condition, $n_i > 2^{c\gamma} \geq 2^\gamma$ since $c \geq 1$. Hence the total work in each recursive level is $\sum O(n_i) = O(n)$. Consider all $O(\sqrt{\log r})$ recursive levels, the total work is $O(n\sqrt{\log r})$. □

With the two lemmas, we can now prove Thm. 4.1.
Proof of Thm. 4.1. Combining Thm. 4.2 and 4.3 gives the work bound in Thm. 4.1. We now prove the span. The span for the base case is $O(\log 2^{c\gamma}) = O(\gamma)$ [24]. The total span for one recursive call is the sum of $O(\log n)$ *whp* (for the distribution step) [12] and $O(\gamma)$ (line 5). Since there are $O(\sqrt{\log r})$ recursive levels, the total span bound is as stated. □

We note that Thm. 4.1 holds for $\gamma = \Theta(\sqrt{\log r})$, but for simplicity we omit the $\Theta(\cdot)$ notation here.
Practical and Stable MSD Sorting Algorithms. Thm. 4.1 shows the theoretical efficiency of parallel MSD algorithms. However, the counting sort [45] used in the analysis is not stable, and is unlikely to be practical due to too many rounds of data movements [30]. Many practical algorithms choose to use the stable counting sort [53]. We analyze it as follows.

Theorem 4.4. *There exists a stable parallel MSD sorting algorithm with $O(n\sqrt{\log r})$ work and $O(2^{\sqrt{\log r}}\sqrt{\log r})$ span.*

Proof. First, note that the work analysis remains the same as Thm. 4.1, since the stable counting sort has the same work as the unstable one. The only difference is that the stable version has span related to the number of buckets, which is $O(\log n + 2^\gamma)$. Assuming $r = \Omega(n)$, this is $O(2^\gamma)$. The total span of the entire MSD sorting is therefore $O(2^{\sqrt{\log r}}\sqrt{\log r})$. □

This indicates the same work bound as the unstable version. Although the span is slightly higher than that in Thm. 4.1, this version enables stability, and is more practical. In fact, many existing practical parallel IS algorithms, such as PLIS [9], use a similar approach as in Thm. 4.4.
Theory-Guided Practice. Thm. 4.4 shows $O(n\sqrt{\log r})$ work for practical parallel MSD sort. This explains why the integer sort algorithms can outperform comparison sorts with $O(n\log n)$ work for realistic range of $r = n^{O(1)}$. Our parameter $\gamma = \sqrt{\log r}$ used in the analysis is also the choice for most practical MSD algorithms. When $r = 2^{64}$, we have $\gamma = \sqrt{\log r} = 8$ and $\theta = 2^{c\gamma} = 2^{8c}$, which roughly matches the existing algorithms that empirically sort 8–12 bits in each recursive level, and use base case sizes from 2^8 to 2^{16} [5, 9, 43].

4.2 The Analysis of DTSort

In this section, we show that our DTSort with the additional steps of sampling and dovetail merging is theoretically efficient. We first prove the following theorem.

Theorem 4.5. *The DTSort algorithm (Alg. 2) is a stable integer sort with $O(n\sqrt{\log r})$ work and $\tilde{O}(2^{\sqrt{\log r}})$ span.*

Proof. DTSort can be viewed as an extension of the generic stable integer sort which we proved work/depth bounds for in Thm. 4.4. Compared to the algorithm analyzed in Thm. 4.4, DTSort has two additional steps: the sampling step and the dovetail merging step. We first show that the asymptotic work bound remains the same. Note that without sampling and merging, a subproblem with size n' has $O(n')$ work. In DTSort, the total number of samples is $|S| = 2^\gamma \log n$. We use base case threshold $\theta = 2^{2\gamma}$, and assume $r = \Omega(n)$, so the cost to sort the samples is $O(|S|\log|S|) = O(2^\gamma \log n \cdot \gamma) = o(2^{2\gamma}) = o(n')$. Other parts in the sampling step have work proportional to $|S|$, so the overall work in Step 1 is $o(n')$.

In the dovetail merging step, we need $O(2^\gamma)$ binary searches on n' records, so the total cost is $O(2^\gamma \log n')$, which is $O(n')$ considering $n' \geq 2^{2\gamma}$ and $r = \Omega(n)$. When moving a bucket of size t, the total work is $O(t)$ (either a direct copy or two rounds of flipping on at most $2t$ records). We also copy out and back at most $n'/2$ elements. Combining both parts, the work for the merging step is also $O(n')$. Therefore, compared to the algorithm we used in Thm. 4.4, DTSort uses asymptotically the same amount of work, $O(n\sqrt{\log r})$.

For the span, note that the algorithm used in Thm. 4.4 has $O(2^\gamma)$ span in each recursive call. Again consider the two additional steps. Taking samples in parallel and sorting them requires $O(\gamma)$ span. We process L and H sequentially

since there are at most 2^γ heavy buckets and exactly 2^γ light buckets. This gives $O(2^\gamma)$ span in each recursive call.

For the dovetail merging step, we need to move at most 2^γ heavy buckets. Moving each of them is either a direct copy, or two rounds of simple flipping. Both can be done by a parallel-for on all affected records, giving $O(\log n)$ span assuming binary forking. This gives $O(2^\gamma \log n) = \tilde{O}(2^\gamma)$ span in one recursive call. Therefore the two additional steps both have $\tilde{O}(2^\gamma)$ span in each recursive call. Considering all $O(\sqrt{\log r})$ levels gives the stated span bound. □

We also note that for the dovetail merging step, if we use the simple parallel merge (i.e., our baseline solution), we can get the same span bound of $O(2^{\sqrt{\log r}}\sqrt{\log r})$ as in Thm. 4.4. Also, if we assume arbitrary forking (e.g., PRAM), we can also obtain the same span bound as in Thm. 4.4. To avoid the minor subtlety here, we hide the logarithmic terms in the theorem by the $\tilde{O}(\cdot)$ notation. Our main claim is that DTSort can achieve $O(n\sqrt{\log r})$ work and is highly parallel.

In addition to the worst-case bound, we can also show that DTSort can achieve asymptotically better work bound on some real-world distributions with heavy duplicates. We show two special cases that DTSort achieves linear work in Thm. 4.6 and 4.7. Here the exponential distribution means the frequencies of keys following the probability density function $f(x; \lambda) = \lambda e^{-\lambda x}$ for $x > 0$ and a parameter $\lambda > 0$.

Theorem 4.6. *DTSort has $O(n)$ work whp if the input key frequency exhibits an exponential distribution with $\lambda e^\lambda \geq \bar{c}/2^\gamma$ for some $\bar{c} > 1$. Here $\lambda > 0$ is the parameter of the exponential distribution, which gives probability density function $f(x; \lambda) = \lambda e^{-\lambda x}$ for $x > 0$.*

Proof. We first consider the exponential distribution, where the input follows the probability density function $f(x; \lambda) = \lambda e^{-\lambda x}$ for $x > 0$ and a parameter $\lambda > 0$. Since we are considering integer sort, the randomly drawn real number is rounded to an integer, and randomly mapped to an integer in $[r]$. DTSort can detect a large fraction of the records as heavy keys when the input is drawn from an exponential distribution. Particularly, the sampling scheme in Alg. 2 can detect keys with more than $\bar{c}n/2^\gamma$ occurrences as heavy keys *whp* for some constant $\bar{c} > 1$ [30], or equivalently, keys with frequencies higher than $\bar{c}/2^\gamma$. This indicates that an integer x with $\lambda e^{-\lambda x} \geq \bar{c}/2^\gamma$ will be identified as heavy. The cumulative distribution function of the exponential distribution is $F(x; \lambda) = 1 - e^{-\lambda x}$ for $x \geq 0$. Thus, the total number of light records is no more than $n \cdot e^{-\lambda x} = \bar{c}n\gamma/2^\gamma = \bar{c}n\sqrt{\log r}/2^{\sqrt{\log r}}$ when plugging in x from the above threshold for heavy keys. However, since we are considering integer sort (keys are rounded to integers), we need to make sure that $x \geq 1$, so we need the assumption that $\lambda e^\lambda \geq \bar{c}/2^\gamma$. Even though we assume the worst case that the rest of the records will be processed for all the rest $\sqrt{\log r} - 1$ levels, the cost to process them

is $O(\bar{c}n\sqrt{\log r}/2^{\sqrt{\log r}} \cdot \sqrt{\log r}) = O(n\log r/2^{\sqrt{\log r}}) = o(n)$. Hence, the total work for DTSort on exponential distribution under the assumption is $O(n)$. □

Theorem 4.7. *DTSort has $O(n)$ work whp if there are no more than $c'2^\gamma$ distinct keys, for some constant $c' < 1$.*

Proof. For simplicity, we use the same notation $\bar{c}$ as the previous proof. Similarly, keys with frequencies higher than $\bar{c}/2^\gamma$ will be detected as heavy. The maximum number of light records is no more than $c'2^\gamma \cdot \bar{c}n'/2^\gamma < c'\bar{c}n'$ for a subproblem of size n'. By setting $c' < 1/\bar{c}$, the number of light records will shrink by a constant fraction. The total cost in a recursive level is linearly proportional to the total size of all subproblems. Since the number of light records decreases geometrically across levels, the total work is asymptotically bounded by the root level, which is $O(n)$. □

5 Implementation Optimizations

We now discuss some optimizations we find useful when implementing DTSort.

Overflow Bucket. Note that integer sort takes the input keys from a range from 0 up to r, but the actual input keys may be in a smaller range. For example, although the key can be 32-bit integers, the actual range can be much smaller than 2^{32}. To take advantage of this, one possibility is to compute the maximal key by a parallel reduce, as in many existing implementations [9, 35]. Here we discuss a simple yet effective optimization to avoid this computation.

Similar to the heavy/light separation of the keys, here we also use sampling to detect the key range. In each subproblem, DTSort sets the upper key range r' to be the largest value in the sample set S. As such, DTSort skips leading zeros in r' since creating buckets for these bits are wasteful. Since the upper bound is computed by a sample set, DTSort creates an *overflow bucket* to hold records whose keys have at least a "1" for the skipped bits. The size of the overflow bucket is usually very small. We put the overflow bucket at the end, and comparison sort records in this bucket. As such, we do not need to compute the maximal key in the input.

Minimizing Data Movement. In Alg. 2, for simplicity, in our description we assume we use the stable distribution mentioned in Sec. 2.4 and write back the bucket to A (see line 15). The second copy-back is not necessary and should be avoided, since the performance of sorting algorithms can suffer due to excessive data movement. In our implementation, we actually maintain two arrays A and T. After the distribution, we use array T for future execution, until we see another distribution or a dovetail merging, and we then distribute or merge the results back to A (or the other array in the recursion). This approach will reduce the total data movement. Finally, if the data remains in T at the end of the algorithm, we need to move them back to A before finishing.

Name	Stable	In-place	Type	Notes
DTSort	Yes	No	Integer	Our integer sort algorithm
PLIS	Yes	No	Integer	ParlayLib integer sort [9]
IPS²Ra	No	Yes	Integer	IPS²Ra integer sort [5]
RS	No	Yes	Integer	RegionsSort [43]
RD	No	No	Integer	RADULS [36]
PLSS	Y/N	Y/N	Comparison	ParlayLib sample sort [9]
IPS⁴o	No	Yes	Comparison	IPS⁴o sample sort [5]

Table 2. Algorithms tested in our experiments. There are two versions of PLSS. Here we use the unstable but faster one. In-place means fully in-place ($o(n)$ extra memory).

6 Experiments

We implemented DTSort in C++ and tested its performance. DTSort uses ParlayLib [9] to support fork-join parallelism.

Experimental Setup. We run our experiments on a 96-core machine (192 hyper-threads) with 4×2.1 GHz Intel Xeon Gold 6252 CPUs (each with 36MB L3 cache) and 1.5TB of main memory. 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 in seconds.

Parameter Selection. DTSort mostly follows the theoretical analysis in Sec. 4, with the minor observation that using a variable radix width γ (based on n) slightly improved the overall performance. Hence, DTSort uses $\gamma = \log_2 \sqrt[3]{n}$ and capped by 8 and 12 (guided by $\Theta(\sqrt{\log r})$, leading to the same asymptotic work bound), similar to some existing IS algorithms such as PLIS. The base case threshold is $\theta = 2^{14}$.

Tested Algorithms. We compare our DTSort with six *baseline* parallel algorithms, including four MSD integer sorting algorithms and two comparison-based algorithms (both are samplesorts). We present their information in Tab. 2.

Synthetic Distributions. We used three ***standard*** real-world distributions to test all algorithms: *Unif-μ* (uniform distribution with μ distinct keys), *Exp-λ* (exponential distribution with parameter $10^{-5}\lambda$), and *Zipf-s* (Zipfian distribution with parameter s), We also designed a special distribution *Bit-Exponential* (*BExp-t*), which aims to simulate "worst-case" inputs for DTSort. For *BExp-t*, every bit of a key is a 0 with probability $1/t$, and 1 otherwise. BExp controls the distribution of the bitwise encoding (rather than frequency) of keys. With a large t, most subproblems will have a mix of light and heavy records, which is an adversarial case for DTSort due to the necessary dovetail merging for almost every subproblem. In fact, BExp is generally hard for all MSD algorithms because it makes subproblem sizes uneven. For simplicity, we say a distribution is *heavy* (or *heavier* than another) if it contains more duplicates, and *light* (*lighter*) otherwise.

We choose five parameters for each distribution to give various ratios of heavy keys (see Tab. 3). For uniform distributions, we test $\mu = 10, 10^3, 10^5, 10^7, 10^9$. For exponential distributions, we test $\lambda = 1, 2, 5, 7, 10$. For Zipfian distributions, we test $s = 0.6, 0.8, 1, 1.2, 1.5$. For BExp distribution,

we test $t = 10, 30, 50, 100, 300$. We round keys to the closest integer for exponential and Zipfian distributions. For the standard distributions, we map the keys to larger ranges (up to 2^{32} or 2^{64}) to test the algorithms on the full range of $[r]$. We use the geometric mean to compare the *average* performance across multiple tests.

6.1 Overall Performance on Synthetic Data

We present the running time of all tested algorithms on synthetic distributions in Tab. 3 with different key lengths.

Standard Distributions. With 32-bit keys, DTSort has the best overall performance, which is 1.27–1.47× faster than the baselines. A heatmap for 32-bit results is also presented in Fig. 1. As expected, the other integer sorting algorithms are usually competitive to DTSort on light distributions, but can be much slower than DTSort on heavy distributions. Take the best baseline on 32-bit keys, PLIS, as an example. On light distributions, It is only 2–8% slower than DTSort, and is faster than comparison sorts. However, on the heaviest several distributions, PLIS is almost always slower than the best comparison sort IPS⁴o, making its overall advantage to IPS⁴o small (only 4%). On the other hand, DTSort can take advantage of the heavy keys and outperforms IPS⁴o in all but two tests (which we study in more details below). Across all tests, DTSort can be up to 2.3× faster than PLIS.

With 64-bit keys, all algorithms perform slower than with 32-bit keys due to the increase in data size, but the integer sorting algorithms are affected more significantly. This matches the theory that the work bounds of these algorithms increase with the value of r (see Sec. 4). In this case, *all* baseline integer sorting algorithms are slower than the fastest comparison sort IPS⁴o on average, and many of them cannot outperform IPS⁴o even on some light distributions. On the contrary, DTSort is still faster than both tested comparison sorts in almost all tests, and is 1.2–1.5× faster on average.

Among all tests on the standard distributions, DTSort *performs the best on all but three tests*, where samplesort IPS⁴o is better. Two of them are *Unif-10³* for both 32- and 64-bits. The reason should be that, based on our parameter γ, the average frequency of *Unif-10³* is the threshold to distinguish heavy and light keys on the root level (about 10^6). We tested this on purpose to show the bad case for DTSort, since all "light keys" are reasonably heavy (right around the threshold), but they have to go through another level of recursion and have to be processed in the dovetail merging step. This also illustrates the inherent difficulty of dealing with duplicates for parallel IS algorithms. The last case where DTSort is slower than IPS⁴o is *Zipf-1*, and DTSort is within 5% slower.

In general, DTSort effectively takes advantage of different levels of heavy duplicates: it exhibits a similar trend as samplesort algorithms where the running time decreases with more duplicates. Meanwhile, as an integer sort, DTSort also has the benefit of being asymptotically more efficient than comparison sorts with realistic input key range $r = n^{O(1)}$.

Instances		32-bit key and 32-bit value pairs						64-bit key and 64-bit value pairs						
		Integer				Comparison		Integer					Comparison	
		Ours	PLIS	IPS²Ra	RS	PLSS	IPS⁴o	Ours	PLIS	IPS²Ra	RS	RD	PLSS	IPS⁴o
Standard distributions:														
Uniform	10^9	.500	.537	.671	.718	1.27	.690	.994	1.14	1.09	1.43	1.86	1.65	1.11
	10^7	.501	.549	.600	.705	1.14	.604	1.03	1.15	1.06	1.71	1.86	1.47	1.05
	10^5	.478	.542	.595	.696	1.08	.653	.859	1.22	1.01	1.36	2.66	1.35	1.10
	10^3	.506	.505	.538	.613	.805	.432	.795	1.41	1.66	1.54	3.23	1.01	.759
	10	.308	.707	1.13	.438	.959	.456	.581	1.93	3.78	1.26	8.25	1.12	.850
Exponential	1	.526	.536	.574	.711	1.11	.671	.976	1.16	1.04	1.39	2.28	1.33	1.16
	2	.502	.546	.577	.711	1.12	.661	.919	1.22	1.05	1.39	2.45	1.35	1.19
	5	.435	.567	.583	.705	1.11	.612	.819	1.52	1.03	1.41	2.44	1.35	1.03
	7	.419	.582	.554	.708	1.08	.609	.782	1.69	1.01	1.48	2.53	1.32	.972
	10	.402	.603	.560	.682	1.09	.561	.763	1.87	1.05	1.53	2.61	1.28	.930
Zipfian	0.6	.493	.543	.630	.720	1.23	.691	1.00	1.14	1.11	1.43	1.82	1.63	1.12
	0.8	.524	.542	.619	.710	1.20	.670	1.00	1.18	1.09	1.46	1.92	1.56	1.08
	1	.601	.631	.648	.735	1.08	.590	1.04	1.44	1.71	1.53	3.10	1.30	1.08
	1.2	.516	.832	1.07	.709	1.10	.743	.918	1.95	3.29	1.72	5.85	1.45	1.22
	1.5	.446	.946	1.90	.695	1.48	.939	.883	2.56	6.57	1.74	6.78	1.99	1.65
Avg.		.472	.601	.698	.679	1.11	.629	.882	1.46	1.49	1.49	2.92	1.39	1.07
Adversarial distributions for integer sort with mixed heavy and light keys:														
Bit-Exponential	10	1.11	.833	1.38	.841	.857	.610	3.30	2.70	3.31	1.89	8.03	1.57	1.08
	30	.643	1.08	3.18	.775	1.27	.908	2.75	4.04	7.90	2.30	12.8	1.36	1.26
	50	.550	1.20	4.29	.717	1.60	1.00	1.85	4.57	11.9	2.27	9.47	1.74	1.45
	100	.512	1.31	5.89	.664	1.99	1.48	1.42	4.92	17.8	2.20	4.96	2.44	2.03
	300	.616	1.40	8.22	.606	2.32	2.02	1.32	5.25	27.9	2.12	4.15	3.26	3.31
	Avg.	.659	1.15	3.91	.716	1.52	1.11	1.99	4.19	10.9	2.15	7.25	1.97	1.68

Table 3. Running time (in seconds) on synthetic data with $n = 10^9$. The instances of one distribution is ordered by increasing number of heavy records from top to bottom. The fastest running time on each input instance is underlined. "Avg." = geometric mean. The keys of RD need to be padded to multiples of 64 bits. We remove it from the 32-bit experiments because it is too slow after padding to 64 bits.

The Adversarial Distribution (BExp). As mentioned above, we design the *BExp* distribution to study an adversarial case of DTSort and other MSD algorithms. The distribution incurs very different sizes of MSD zones for *all* subproblems, which incurs load imbalance for all MSD algorithms. It also generates a mix of light and heavy buckets in almost all subproblems, which greatly increases the cost of the dovetail merging and offsets the benefit of identifying heavy keys.

Almost all existing IS algorithms have poor performance on this distribution. Interestingly, even the comparison-based sorting algorithms perform much worse when the distribution is very skewed. Among the MSD algorithms, IPS²Ra and RD are significantly affected and can be up to 10× slower than the others. DTSort, PLIS, and RS are lightly affected on 32-bit keys, but in general can still be within 2× slower compared to their performance on standard distributions. A possible explanation is that all the three algorithms use similar work-stealing scheduler (cilk for RS, ParlayLib scheduler for DTSort and PLIS), which supports dynamic load balancing. This matches our scalability tests (reported in the full paper [21]), where IPS²Ra and RD do not scale beyond 24 threads on this distribution. RS seems to be particularly resistant to the BExp distribution. One reason is that RS is *unstable*, and records with the same key do not need to be put in a specific order. This sacrifice on stability has a side-effect to benefit from the highly skewed distributions.

Overall, DTSort still achieves the best performance on 32-bit keys, and is still faster than comparison sorts in most cases. We believe this advantage comes from the removal of heavy keys from recursive problems, and thus alleviates the load-imbalance issue. In addition, the performance of DTSort is also significantly improved by using our proposed DTMerge algorithm. As we will show in Sec. 6.3, as expected, on BExp distributions, the merging step is very expensive, which can take 50% of the entire running time if we simply use parallel merge. Our new algorithm accelerates this step by 1.7–2.8×, making the overall running time much lower.

6.2 Applications

Integer sort is widely used in practice. We choose two representative applications on graph and geometric processing to evaluate the performance of the tested algorithms.

Graph Transpose. Given a directed graph $G = (V, E)$, the graph transpose problem is to generate $G^\top = (V, E^\top)$ where $E^\top = \{(v, u) : (u, v) \in E\}$. Here we assume the edges are

Instances		n	Integer				Comparison	
			Ours	PLIS	IPS^{2}Ra	RS	PLSS	IPS^{4}o
Graph transpose								
	LJ	**69.0M**	<u>.043</u>	.043	s.g.	.065	.080	.159
	TW	**1.47B**	<u>.888</u>	.942	3.24	1.05	1.57	.891
	CM	**1.61B**	<u>.782</u>	.945	1.41	1.07	1.84	1.12
	SD	**2.04B**	<u>1.10</u>	1.29	2.87	1.34	2.08	1.23
	CW	**42.6B**	28.5	37.0	32.5	s.g.	60.1	<u>24.6</u>
	Avg.		<u>.985</u>	1.13	-	-	1.96	1.37
Morton order								
RealWorld	**GL**	**24.9M**	.026	.028	.259	<u>.024</u>	.028	.171
	CM	**321M**	.184	<u>.178</u>	.343	.209	.327	.338
	OSM	**2.77B**	2.32	2.39	3.65	s.g.	2.73	<u>1.53</u>
	Avg.		<u>.223</u>	.227	.687	-	.293	.445
Varden [23]	**SS2d**	**1B**	<u>.498</u>	.557	.662	.634	1.27	.775
	SS3d	**1B**	<u>.512</u>	.568	.778	.611	1.12	.754
	SS2d'	**2B**	<u>.973</u>	1.16	1.27	1.17	2.44	1.39
	SS2d'	**2B**	<u>.990</u>	1.60	2.86	1.17	2.30	1.96
	Avg.		<u>.704</u>	.875	1.17	.854	1.68	1.12

Table 4. Running time (in seconds) on multiple applications. We use 32-bit keys and 32-bit values. See Sec. 6.2 for the detailed explanation on the keys and values. The fastest running time on each instance is underlined. "n" = input sizes. "s.g." = segmentation fault. "Avg." = the geometric mean on instances of the same application. "-" = not applicable.

stored using the standard format of compressed sparse row. To compute the transposed graph, we only need to stably integer sort all edges using the key as the second vertex v; here the vertices with larger degrees are the "heavy keys". Graph transpose is widely used as a subroutine in many graph algorithms [14, 19, 33, 49].

We tested five real-world directed graphs, including soc-LiveJournal (LJ) [6], twitter (TW) [37], Cosmo50 (CM) [38, 56], sd_arc (SD) [42], and clueweb (CW) [42]. The largest graph has 42.6 billion directed edges. We save each edge as a (32-bit, 32-bit) integer pair since all vertex ids are within 32 bits. On social networks (LJ and TW) and web graphs (SD and CW), the degree distributions are more skewed (i.e., more duplicate keys). CM is a k-NN graph, and the degrees are more evenly-distributed. We present our results in Tab. 4.

Compared to the best baseline (PLIS), DTSort is always faster by up to 1.3×. The samplesort IPS^{4}o performs the best on CW, but performs much slower on small graphs. Overall, DTSort has the best performance across all graphs.

Morton Sort. For a d-dimensional point, a z-value is calculated by interleaving the binary representations of each coordinate. The process to sort the z-values of a set of points is called Morton sort, which orders multidimensional data in one dimension while preserving the locality of data points. It is widely used in geometry, graphics, and machine learning applications [10, 11, 26, 27, 40, 55].

We tested seven datasets for Morton sort and list the result in Tab. 4. Three are real-world datasets, including GeoLife (GL) [56, 60], Cosmo50 (CM) [38, 56] and Open-StreetMap (OSM) [1]. The other four are synthetic datasets by the widely-used generator Varden [23] that creates points with varying densities. On the synthetic distribution with varying densities, DTSort outperforms all other baselines by 1.2–3.4×. On the three datasets with real-world points, existing integer sorts (RS or PLIS) achieved the best performance on GL and CM, while DTSort is competitive (3–17% slower). On OSM, none of the integer sorts outperform the samplesort IPS^{4}o. Interestingly, DTSort still achieved the best average performance on the three datasets (PLIS is also close). This may be due to the reason that DTSort *combines the benefit for both comparison and integer sorts*, and hence *does not have a very bad worst-case performance*.

6.3 Performance Study

Heavy Keys Detection Evaluation. To understand the performance gain by detecting heavy keys and processing them separately, we compare our implementation with (DTSort) and without (Plain) heavy keys detection in Fig. 4 (a) and (b). We select eight representative distributions, which are the lightest and heaviest cases for each distribution respectively.

As discussed in Sec. 3, the overhead of DTSort compared to Plain includes sampling to detect heavy keys, and merging heavy keys with light keys. On 32-bit inputs, when there are no or fewer heavy keys, DTSort is comparable with Plain, with at most 12% performance loss on *BExp-10*. It indicates that the overhead of sampling is negligible compared to other steps. With heavy duplicate keys, since heavy keys are gathered in separate buckets and do not need to be sorted in later recursions, DTSort outperforms Plain by up to 2.12× (on *BExp-300*). Overall, DTSort outperforms Plain by 25% on average. On 64-bit input, the performance improvement by using heavy keys detection becomes larger. DTSort consistently outperforms Plain, with a 1.50× speedup on average.

Dovetail Merge Evaluation. We present an in-depth study of the dovetail merging step. As mentioned in Sec. 3.4, our baseline solution (parallel merge), is correct and even asymptotically better, but it requires to move all records between the input array and a temporary array. To achieve better performance, we utilized the special property of this problem (i.e., very few heavy buckets) and proposed our DTMerge algorithm in Sec. 3.4. We select seven representative distributions to evaluate the merge performance. For uniform distribution, we only use *Unif-10³*, since no merge is needed in the lightest and heaviest cases. For other distributions, we select the lightest and heaviest cases.

To illustrate the effectiveness of our DTMerge approach, we compare it against a standard parallel merge in Parlay-Lib [9] (referred to as PLMerge) as the baseline. When reporting the running time, we also run the algorithms *without* the merging step in all recursive calls. This will not give
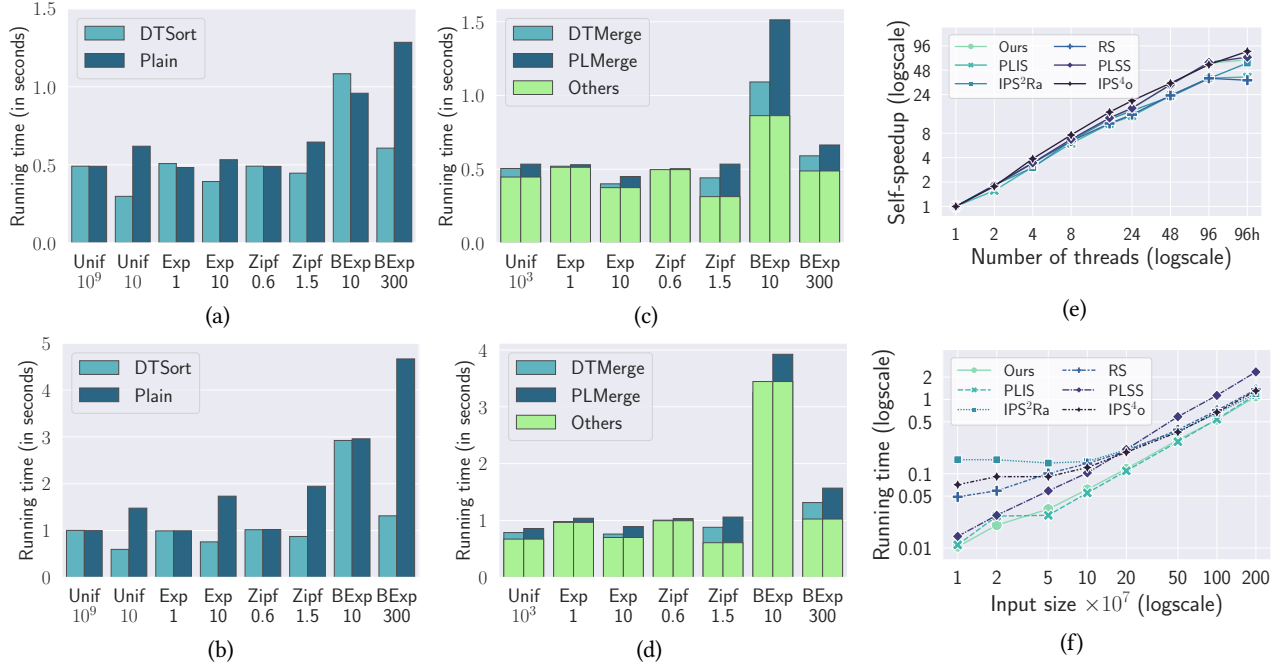
Figure 4. (a) and (b): Analysis for the performance of heavy-key detection. Numbers are running time (lower is better) with or without heavy-key detection. (a) is for 32-bit keys and (b) is for 64-bit keys. (c) and (d): Analysis for the performance of dovetail merging. Numbers are running time (lower is better) using our dovetail merging algorithm or a baseline merging algorithm. (c) is for 32-bit keys and (d) is for 64-bit keys. (e) and (f): Scalability (higher is better) with varying number of threads and running time (lower is better) with varying input sizes on 32-bit key and 32-bit value pairs on one instance: *Zipf-0.8*. Full analysis is given in the full paper [21]. Discussions are in Sec. 6.3.

a correct sorting result, but indicates the running time of the other steps in DTSort. We present the result for both 32 bits and 64 bits in Fig. 4 (c) and (d). Since they generally have similar trends, we discuss the experiments based on 32 bits.

On lighter distributions (e.g., *Exp-1* and *Zipf-0.6*), the overall improvement is marginal given that the dovetail merging step is fast anyway. However, on heavier (more skewed) distributions, this step can take a significant portion of the overall time if we use PLMerge, which is 17% on *Exp-10* and 42% on *Zipf-1.5*. With our DTMerge, this step only takes 6% in *Exp-10* and 28% for *Zipf-1.5*, improving the overall performance for both by about 20%.

For BExp, since almost all subproblems have a mix of heavy and light keys, the merging time is significant. Using PLMerge, the merging step takes 27–43% running time. Our DTMerge improved the performance of the merging step by up to 2.8×, leading to 1.1–1.38× faster sorting time.

Scalability Tests. DTSort has good scalability to both number of processors and input sizes. Due to page limit, we provide results on one distribution as an example in Fig. 4 (e) and (f), and give the full discussions in the full paper [21].

7　Conclusion

We study the theory and practice of parallel integer sort. In theory, we show that a class of widely-used practical MSD integer sorting algorithms have $O(n\sqrt{\log r})$ work and low span. The bounds are asymptotically lower than comparison

sort for a large key range $r = n^{o(\log n)}$, and provide a solid theoretical foundation for explaining the high performance of parallel integer sort. In practice, we propose DTSort to overcome a common challenge in existing MSD algorithms: the difficulty to detect and take advantage of duplicate keys. DTSort combines algorithmic insights from both integer and comparison sorting algorithms, and uses a special dovetail merging step to make it more efficient. DTSort achieves the same $O(n\sqrt{\log r})$ work bound as other MSD integer sorting algorithms, and has the same advantage of processing heavy duplicates efficiently as samplesort algorithms. In our experiments, DTSort achieves competitive or better performance than state-of-the-art parallel integer and comparison sorting algorithms on various synthetic and real-world datasets. As a future work, we consider further improving the performance of parallel sorting algorithm by accelerating the distributing step, inspired by some existing work [5, 29, 43].

References

[1] 2010. OpenStreetMap © OpenStreetMap contributors. https://www.openstreetmap.org/.

[2] Susanne Albers and Torben Hagerup. 1997. Improved parallel integer sorting without concurrent writing. *Information and Computation* 136, 1 (1997), 25–51.

[3] Arne Andersson, Torben Hagerup, Stefan Nilsson, and Rajeev Raman. 1998. Sorting in Linear Time? *J. Computer and System Sciences* 57, 1 (1998), 74–93.

[4] N. S. Arora, R. D. Blumofe, and C. G. Plaxton. 2001. Thread Scheduling for Multiprogrammed Multiprocessors. *Theory of Computing Systems (TOCS)* 34, 2 (01 Apr 2001).

[5] Michael Axtmann, Sascha Witt, Daniel Ferizovic, and Peter Sanders. 2022. Engineering in-place (shared-memory) sorting algorithms. *ACM Transactions on Parallel Computing (TOPC)* 9, 1 (2022), 1–62.

[6] Lars Backstrom, Dan Huttenlocher, Jon Kleinberg, and Xiangyang Lan. 2006. Group formation in large social networks: membership, growth, and evolution. In *ACM International Conference on Knowledge Discovery and Data Mining (SIGKDD)*. 44–54.

[7] Pramod Chandra P Bhatt, Krzysztof Diks, Torben Hagerup, VC Prasad, Tomasz Radzik, and Sanjeev Saxena. 1991. Improved deterministic parallel integer sorting. *Information and Computation* 94, 1 (1991), 29–47.

[8] Arghya Bhattacharya, Abiyaz Chowdhury, Helen Xu, Rathish Das, Rezaul A Chowdhury, Rob Johnson, Rishab Nithyanand, and Michael A Bender. 2022. When Are Cache-Oblivious Algorithms Cache Adaptive? A Case Study of Matrix Multiplication and Sorting. In *European Symposium on Algorithms (ESA)*.

[9] Guy E. Blelloch, Daniel Anderson, and Laxman Dhulipala. 2020. ParlayLib — a toolkit for parallel algorithms on shared-memory multicore machines. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. 507–509.

[10] Guy E Blelloch and Magdalen Dobson. 2022. Parallel Nearest Neighbors in Low Dimensions with Batch Updates. In *Algorithm Engineering and Experiments (ALENEX)*. SIAM, 195–208.

[11] Guy E. Blelloch, Daniel Ferizovic, and Yihan Sun. 2016. Just Join for Parallel Ordered Sets. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*.

[12] Guy E. Blelloch, Jeremy T. Fineman, Yan Gu, and Yihan Sun. 2020. Optimal parallel algorithms in the binary-forking model. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. 89–102.

[13] Guy E. Blelloch, Phillip B. Gibbons, and Harsha Vardhan Simhadri. 2010. Low depth cache-oblivious algorithms. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*.

[14] Guy E. Blelloch, Yan Gu, Julian Shun, and Yihan Sun. 2020. Parallelism in Randomized Incremental Algorithms. *J. ACM* 67, 5 (2020), 1–27.

[15] Guy E. Blelloch, Charles E. Leiserson, Bruce M. Maggs, C. Greg Plaxton, Stephen J. Smith, and Marco Zagha. 1991. A Comparison of Sorting Algorithms for the Connection Machine CM-2. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*.

[16] Robert D. Blumofe and Charles E. Leiserson. 1998. Space-Efficient Scheduling of Multithreaded Computations. *SIAM J. on Computing* 27, 1 (1998).

[17] Minsik Cho, Daniel Brand, Rajesh Bordawekar, Ulrich Finkler, Vincent Kulandaisamy, and Ruchir Puri. 2015. PARADIS: an efficient parallel algorithm for in-place radix sort. *Proceedings of the VLDB Endowment (PVLDB)* 8, 12 (2015), 1518–1529.

[18] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2009. *Introduction to Algorithms (3rd edition)*. MIT Press.

[19] Laxman Dhulipala, Guy E. Blelloch, and Julian Shun. 2021. Theoretically efficient parallel graph algorithms can be fast and scalable. *ACM Transactions on Parallel Computing (TOPC)* 8, 1 (2021), 1–70.

[20] Xiaojun Dong, Laxman Dhulipala, Yan Gu, and Yihan Sun. 2023. DovetailSort: A Parallel Integer Sort Algorithm. https://github.com/ucrparlay/DovetailSort.

[21] Xiaojun Dong, Laxman Dhulipala, Yan Gu, and Yihan Sun. 2024. Parallel Integer Sort: Theory and Practice. *arXiv preprint:2401.00710* (2024).

[22] 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 *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*.

[23] Junhao Gan and Yufei Tao. 2017. On the hardness and approximation of Euclidean DBSCAN. *ACM Transactions on Database Systems (TODS)* 42, 3 (2017), 1–45.

[24] Michael T Goodrich and Riko Jacob. 2023. Optimal Parallel Sorting with Comparison Errors. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. 355–365.

[25] David Gries and Harlan Mills. 1981. *Swapping sections*. Technical Report. Cornell University.

[26] Yan Gu, Yong He, and Guy E Blelloch. 2015. Ray specialized contraction on bounding volume hierarchies. In *Computer Graphics Forum*, Vol. 34. 309–318.

[27] Yan Gu, Yong He, Kayvon Fatahalian, and Guy Blelloch. 2013. Efficient BVH construction via approximate agglomerative clustering. In *High-Performance Graphics (HPG)*.

[28] Yan Gu, Zachary Napier, and Yihan Sun. 2022. Analysis of Work-Stealing and Parallel Cache Complexity. In *SIAM Symposium on Algorithmic Principles of Computer Systems (APOCS)*. SIAM, 46–60.

[29] Yan Gu, Omar Obeya, and Julian Shun. 2021. Parallel In-Place Algorithms: Theory and Practice. In *SIAM Symposium on Algorithmic Principles of Computer Systems (APOCS)*. 114–128.

[30] Yan Gu, Julian Shun, Yihan Sun, and Guy E. Blelloch. 2015. A Top-Down Parallel Semisort. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. 24–34.

[31] Torben Hagerup. 1991. Constant-time parallel integer sorting. In *ACM Symposium on Theory of Computing (STOC)*. 299–306.

[32] Hiroshi Inoue and Kenjiro Taura. 2015. SIMD-and cache-friendly algorithm for sorting an array of structures. *Proceedings of the VLDB Endowment* 8, 11 (2015), 1274–1285.

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

[34] Jyrki Katajainen and Jesper Larsson Träff. 1997. A meticulous analysis of mergesort programs. In *Italian Conference on Algorithms and Complexity (ICAC)*. Springer, 217–228.

[35] Marek Kokot, Sebastian Deorowicz, and Agnieszka Debudaj-Grabysz. 2017. Sorting data on ultra-large scale with RADULS: New incarnation of radix sort. In *Beyond Databases, Architectures and Structures (BDAS)*. Springer, 235–245.

[36] Marek Kokot, Sebastian Deorowicz, and Maciej Długosz. 2018. Even faster sorting of (not only) integers. In *Man-Machine Interactions 5: 5th International Conference on Man-Machine Interactions, ICMMI 2017 Held at Kraków, Poland, October 3-6, 2017*. Springer, 481–491.

[37] Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue Moon. 2010. What is Twitter, a social network or a news media?. In *International World Wide Web Conference (WWW)*. 591–600.

[38] YongChul Kwon, Dylan Nunley, Jeffrey P Gardner, Magdalena Balazinska, Bill Howe, and Sarah Loebman. 2010. Scalable clustering algorithm for N-body simulations in a shared-nothing cluster. In *International Conference on Scientific and Statistical Database Management*. Springer, 132–150.

[39] Shin-Jae Lee, Minsoo Jeon, Dongseung Kim, and Andrew Sohn. 2002. Partitioned parallel radix sort. *J. Parallel Distrib. Comput.* 62, 4 (2002), 656–668.

[40] Shengren Li, Lance C Simons, Jagaseesh Bhaskar Pakaravoor, Fatemeh Abbasinejad, John D Owens, and Nina Amenta. 2012. kANN on the GPU with shifted sorting. In *High-Performance Graphics (HPG)*.

[41] Yossi Matias and Uzi Vishkin. 1991. On parallel hashing and integer sorting. *J. Algorithms* 12, 4 (1991), 573–606.

[42] Robert Meusel, Oliver Lehmberg, Christian Bizer, and Sebastiano Vigna. 2014. Web Data Commons — Hyperlink Graphs. http://webdatacommons.org/hyperlinkgraph.

[43] Omar Obeya, Endrias Kahssay, Edward Fan, and Julian Shun. 2019. Theoretically-Efficient and Practical Parallel In-Place Radix Sorting. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. 213–224.

[44] Orestis Polychroniou and Kenneth A Ross. 2014. A comprehensive study of main-memory partitioning and its application to large-scale comparison-and radix-sort. In *ACM SIGMOD International Conference on Management of Data (SIGMOD)*. 755–766.

[45] Sanguthevar Rajasekaran and John H. Reif. 1989. Optimal and sublogarithmic time randomized parallel sorting algorithms. *SIAM J. on Computing* 18, 3 (1989), 594–607.

[46] Hitoshi Sato, Ryo Mizote, Satoshi Matsuoka, and Hirotaka Ogawa. 2016. I/O chunking and latency hiding approach for out-of-core sorting acceleration using GPU and flash NVM. In *IEEE International Conference on Big Data (Big Data)*. 398–403.

[47] Hideyuki Shamoto, Koichi Shirahata, Aleksandr Drozd, Hitoshi Sato, and Satoshi Matsuoka. 2016. GPU-accelerated large-scale distributed sorting coping with device memory capacity. *IEEE Trans. on Big Data* 2, 1 (2016), 57–69.

[48] Xipeng Shen and Chen Ding. 2004. Adaptive data partition for sorting using probability distribution. In *International Conference on Parallel Processing (ICPP)*. IEEE, 250–257.

[49] George M. Slota, Sivasankaran Rajamanickam, and Kamesh Madduri. 2014. BFS and coloring-based parallel algorithms for strongly connected components and related problems. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 550–559.

[50] Edgar Solomonik and Laxmikant V Kale. 2010. Highly scalable parallel sorting. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 1–12.

[51] Yihan Sun, Guy E Blelloch, Wan Shen Lim, and Andrew Pavlo. 2019. On supporting efficient snapshot isolation for hybrid workloads with multi-versioned indexes. *Proceedings of the VLDB Endowment (PVLDB)* 13, 2 (2019), 211–225.

[52] Jesper Larsson Träff. 2018. Parallel quicksort without pairwise element exchange. *arXiv preprint:1804.07494* (2018).

[53] Uzi Vishkin. 2010. Thinking in parallel: Some basic data-parallel algorithms and techniques. (2010).

[54] Letong Wang, Xiaojun Dong, Yan Gu, and Yihan Sun. 2023. Parallel Strong Connectivity Based on Faster Reachability. In *ACM SIGMOD International Conference on Management of Data (SIGMOD)*.

[55] Yiqiu Wang, Yan Gu, and Julian Shun. 2020. Theoretically-Efficient and Practical Parallel DBSCAN. In *ACM SIGMOD International Conference on Management of Data (SIGMOD)*. 2555–2571.

[56] Yiqiu Wang, Shangdi Yu, Laxman Dhulipala, Yan Gu, and Julian Shun. 2021. GeoGraph: A Framework for Graph Processing on Geometric Data. *ACM SIGOPS Operating Systems Review* 55, 1 (2021), 38–46.

[57] Jan Wassenberg and Peter Sanders. 2011. Engineering a multi-core radix sort. In *European Conference on Parallel Processing (Euro-Par)*. Springer, 160–169.

[58] Qingxuan Yang, John Ellis, Khalegh Mamakani, and Frank Ruskey. 2013. In-place permuting and perfect shuffling using involutions. *Inform. Process. Lett.* 113, 10-11 (2013), 386–391.

[59] Keliang Zhang and Baifeng Wu. 2012. A novel parallel approach of radix sort with bucket partition preprocess. In *International Conference on High Performance Computing (HPCC)*. IEEE, 989–994.

[60] Yu Zheng, Like Liu, Longhao Wang, and Xing Xie. 2008. Learning transportation mode from raw gps data for geographic applications on the web. In *International World Wide Web Conference (WWW)*. 247–256.

A   Artifact Description

A.1   Availability

Our artifact is available on Zenodo: https://doi.org/10.5281/zenodo.10252150. Our code is also released on GitHub: https://github.com/ucrparlay/DovetailSort. A detailed README file on how to compile and run the code is provided on the GitHub repository.

A.2   Requirements

- Hardware: Any modern multi-core Intel machines (ideally more than 32 physical cores). 128GB memory is required for the synthetic dataset. Running the complete dataset (including applications) requires about 500GB memory.

- Software: Linux machines with g++ or clang++ supporting C++17 features.

Our implementation is based on ParlayLib [9], which provides fork-join parallelism support and some basic parallel primitives. We suggest using numactl -i all to interleave the memory on all CPUs except for the sequential tests. We use it by default, but all our code can also run without it.

A.3   Getting the Artifact

Our code can be acquired from GitHub under the ppopp24-artifact branch:

```
git clone -b ppopp24-artifact https://github.com/
    ucrparlay/DovetailSort.git
cd DovetailSort/
```

A.4   Running the Artifact

We prepare scripts to reproduce the main tables and figures in our paper.

Synthetic dataset.. This script can reproduce our experiments on synthetic data (Tab. 3).

```
./scripts/run_synthetic.sh
```

It generates a table in TSV format in results/synthetic.tsv.

Applications.. The dataset for the two applications are available on Google Drive: https://tinyurl.com/DoveTailSort. Users can download the dataset into data/ and uncomment the following two lines in include/parlay/integer_sort.cpp

```
run_all_graphs(id);
run_all_points(id);
```

Input sizes and parallel scalability.. The raw data can be generated by the following scripts. It runs on the representative distribution in Fig. 4.

```
./scripts/run_sizes_scalability.sh
./scripts/run_parallel_scalability.sh
```

The results will be saved to results/sizes_scalability.tsv and results/parallel_scalability.tsv.

A.5   Optional: Customize the Experiments

The users can compile our code using the Makefile in the include/parlay/ directory. The make command compiles the executable using clang++ by default.

```
cd include/parlay/
make
```

It can also compile with g++ by:

```
make GCC=1
```

To compile the serial version, run:

```
make SERIAL=1
```

The default executable will run all synthetic distributions (uniform, exponential, Zipfian, and Bit-Exponential) with input sizes 10^9 in parallel.

```
numactl -i all ./integer_sort [n]
```

$n = 10^9$ if not specified. The users can change n to any number.