External-Memory Sorting with Comparison Errors

Michael T. Goodrich and Evrim Ozel(✉)

University of California, Irvine, USA
{goodrich,eozel}@uci.edu

Abstract. We provide several algorithms for sorting an array of n comparable distinct elements subject to probabilistic comparison errors in external memory. In this model, which has been extensively studied in internal-memory settings, the comparison of two elements returns the wrong answer according to a fixed probability, $p_e < 1/2$, and otherwise returns the correct answer. The ***dislocation*** of an element is the distance between its position in a given (current or output) array and its position in a sorted array. There are various existing algorithms that can be utilized for sorting or near-sorting elements subject to probabilistic comparison errors, but these algorithms do not translate into efficient external-memory algorithms, because they all make heavy use of noisy binary searching. In this paper, we provide new efficient methods that are in the external-memory model for sorting with comparison errors. Our algorithms achieve an optimal number of I/Os, in both cache-aware and cache-oblivious settings.

1 Introduction

Given n distinct comparable elements, we study the problem of efficiently sorting them subject to noisy probabilistic comparisons. In this framework, which has been extensively studied in internal-memory settings [2,5,7–9,11–16,21], the comparison of two elements, x and y, results in a true and accurate result independently according to a fixed probability, $p < 1/2$, and otherwise returns the opposite (false) result. In the case of ***persistent*** errors [2,7–9,13], the result of a comparison of two given elements, x and y, always returns the same result. In the case of ***non-persistent*** errors [5,11,12,14,16,21], however, the probabilistic determination of correctness is determined independently for each comparison, even if it is for a pair of elements, (x, y), that were previously compared.

Motivation for sorting with comparison errors comes from multiple sources, including ranking objects online via A/B testing [22], which evaluates the impact of a new technology or technological choice by executing a system in a real production environment and testing two instances of its performance (an "A" and "B") on a random subset of the users of the platform. Such systems can involve many users and choices to compare via A/B testing, e.g., see [10,20]; hence, we feel that managing such implementations could benefit from external-memory solutions.

P. Morin and S. Suri (Eds.): WADS 2023, LNCS 14079, pp. 493–506, 2023.
https://doi.org/10.1007/978-3-031-38906-1_32

Since one cannot always correctly sort an array, A, subject to persistent comparison errors, we follow Geissmann *et al.* [7–9], and define the ***dislocation*** of an element, x, in an array, A, as the absolute value of the difference between x's index in A and its index in the correctly sorted permutation of A. Further, define the ***maximum dislocation*** of A as the maximum dislocation for the elements in A, and define the ***total dislocation*** of A is the sum of the dislocations of the elements in A. By known lower bounds [7–9], the best a sorting algorithm can achieve under persistent comparison errors is a maximum dislocation of $O(\log n)$ and a total dislocation of $O(n)$.

In this paper, we are interested in sorting algorithms that are in the ***external-memory*** model. Unfortunately, the existing algorithms for sorting with noisy comparisons are not easily converted into efficient external-memory algorithms, because they all make use of noisy binary search, which involves a random walk in a binary search tree [5,8,14]. Instead, we desire efficient sorting algorithms that tolerate noisy comparisons and have an efficient number of input/output operations, primarily for the persistent model, since we can sort an array with maximum dislocation of $O(\log n)$ in the non-persistent model by a single scan where we repeat each comparison in internal memory $O(\log n)$ times.

Intuitively, the main disadvantage of relying on noisy binary search is that it is cache inefficient, in that it requires performing memory accesses for widely-distributed storage locations. Large-scale applications need to minimize the number of input/output (I/O) operations to external memory. Thus, we also desire sorting algorithms that tolerate noisy comparisons and minimize the number of I/Os. In external-memory applications, I/Os occur in terms of memory blocks. In this context, we use M to denote the size of internal memory and B to denote the size of a block of memory, and we note that the best I/O bound that is possible for sorting an array of size N in external-memory is $\Theta((N/B)\log_{M/B}(N/B))$, see, e.g., [19]. Thus, we also desire sorting algorithm that tolerate noisy comparisons and have this bound on their number of I/Os. Moreover, we desire solutions that are either cache-aware (taking advantage of knowledge of the parameters M and B) or cache-oblivious (which don't know the parameters M and B).

Related Prior Results. The non-persistent error model traces back to a classic problem by Rényi [18] of playing a game involving posing questions to someone who lies with a given probability; see, e.g., the survey by Pelc [17]. Braverman and Mossel [2] introduced the persistent-error model, where comparison errors are persistently wrong with a fixed probability, $p < 1/2 - \varepsilon$, and they achieved a running time of $O(n^{3+f(p)})$ time with maximum expected dislocation $O(\log n)$ and total dislocation $O(n)$. Klein, Penninger, Sohler, and Woodruff [13] improve the running time to $O(n^2)$, but with $O(n \log n)$ total dislocation w.h.p. The internal-memory running time for sorting in the persistent-error model optimally with respect to maximum and total dislocation was subsequently improved to $O(n^2)$, $O(n^{3/2})$, and ultimately to $O(n \log n)$, in a sequence of papers by Geissmann, Leucci, Liu, and Penna [7–9].

Feige, Raghavan, Peleg, and Upfal [5] provide a parallel algorithm for sorting with non-persistent errors that, with high probability, runs in $O(\log n)$ time and

$O(n \log n)$ work in the CRCW PRAM model, and Leighton, Ma, and Plaxton [14] show how to achieve these bounds in the EREW PRAM model.

None of these prior algorithms translate into efficient external-memory algorithms, however, where we focus on optimizing the number of input/output (I/O) operations. The main reason is that they all use noisy binary searching, which is a random walk in a binary search tree, where each step involves a noisy comparison. As an external-memory algorithm, this search algorithm unfortunately involves far-flung comparisons; hence, it causes a lot of I/Os.

Frigo, Leiserson, Prokop and Ramachandran [6] introduced the notion of cache-oblivious algorithms, which are algorithms that do not have any variables dependent on hardware parameters such as cache or block size that need to be tuned for it to perform optimally. The authors also introduced the ***(M,B) ideal-cache model*** to analyze cache oblivious algorithms, and defined the ***cache complexity*** $Q(n)$ and ***work complexity*** $W(n)$ of an algorithm with input size n, which respectively measure the number of cache misses the algorithm incurs in the ideal-cache model, and the conventional running time of the algorithm in a RAM model. The authors then introduced a cache-oblivious sorting algorithm, Funnelsort, and showed, assuming $M = \Omega(B^2)$ (also known as the ***tall-cache assumption***), that Funnelsort is cache-oblivious, has work complexity $O(n \log n)$ and cache complexity $O(1 + \frac{n}{B}(1 + \log_M n))$, which matches the $\Omega(\frac{n}{B} \log_{M/B} \frac{n}{B})$ lower bound for sorting in the external-memory model.

Our Results. In this paper, we provide efficient sorting algorithms in the external-memory model that tolerate noisy comparisons. All our algorithms utilize an optimal number of I/Os. In particular, we provide solutions for either the persistent or non-persistent error models, and for the cache-aware and cache-oblivious external-memory models. Our algorithms avoid using noisy binary searching by instead utilizing a generalized subroutine that is an external-memory version of window-sort. This allows us to then design windowed versions of external-memory merge-sort and funnel-sort. Both algorithms run in time $O(n \log^2 n)$ in internal memory, or in external memory with an optimal $O(n/B) \log_{M/B}(n/B)$ I/O's, subject to comparison errors with probability $p_e < 1/2$ so as to have a maximum dislocation of $O(\log n)$ w.h.p. For both algorithms, we assume that the block size is at least logarithmic in the problem size, i.e., $B = \Omega(\log n)$. Our windowed version of funnel-sort will also use the tall-cache assumption, i.e., $M = \Omega(B^2)$. In the sections that follow, we describe our algorithms for sorting with comparison errors.

2 Window-Sort

We begin with a version of window-sort [8], which will be useful as a subroutine in our algorithms. We provide the pseudo-code at a high level in Algorithm 1, for approximately sorting an array of size n that has maximum dislocation at most $d_1 \leq n$ so that it will have maximum dislocation at most $d_2 = d_1/2^k$, for some integer $k \geq 1$, with high probability as a function of d_2.

Algorithm 1: Window-Sort($A = \{a_0, a_1, \ldots, a_{n-1}\}, d_1, d_2$)

```
1 for w ← 2d_1, d_1, d_1/2, ..., 2d_2 do
2     foreach i ← 0, 1, 2, ..., n − 1 do
3         r_i ← max{0, i − w} + |{a_j < a_i : |j − i| ≤ w}|
4     Sort A (deterministically) by nondecreasing r_i values (i.e., using r_i as the
      comparison key for a_i)
5 return A
```

We note that determining the r_i values can be done by scans; hence, that step is I/O efficient in either cache-aware or cache-oblivious settings. Moreover, the sorting step can be done with an I/O efficient algorithm, in either the cache-aware or cache-oblivious settings, e.g., see [3,4,6]. For completeness, we provide below an analysis of window-sort. We note that to simplify our presentation, we assume $p_e \leq 1/16$, however this constraint can be relaxed to any $p_e < 1/2$ to obtain the same asymptotic results.

Lemma 1. *Suppose the comparison error probability, p_e, is at most $1/16$. If an array, A, has maximum dislocation at most d' prior to an iteration of window-sort for $w = 2d'$ (line 1 of Algorithm 1), then after this iteration, A will have maximum dislocation at most $d'/2$ with probability at least $1 - n2^{-d'/8}$.*

Proof. Let a_i be an element in A. Let W denote the window of elements in A for which we perform comparisons with a_i in this iteration; hence, $2d' \leq |W| \leq 4d'$. Because A has maximum dislocation d', by assumption, there are no elements to the left (resp., right) of W that are greater than a_i (resp., less than a_i). Thus, a_i's dislocation after this iteration depends only on the comparisons between a_i and elements in its window. Let X be a random variable that represents a_i's dislocation after this iteration, and note that $X \leq Y$, where Y is the number of incorrect comparisons with a_i performed in this iteration. Note further that we can write Y as the sum of $|W|$ independent indicator random variables and that $\mu = E[Y] = p_e|W| \leq d'/4$. Thus, if we let $R = d'/2$, then $R \geq 2\mu$; hence, we can use a Chernoff bound as follows:

$$\Pr(X > d'/2) \leq \Pr(Y > d'/2) = \Pr(Y > R) \leq 2^{-R/4} = 2^{-d'/8}.$$

Thus, with the claimed probability, the maximum dislocation for all of A will be at most $d'/2$, by a union bound. □

This allows us to implement window-sort in external memory, as follows.

Theorem 1. *Suppose the comparison error probability, p_e, is at most $1/16$. If an array, A, of size n has maximum dislocation at most $d_1 \geq \log n$, then executing Window-Sort(A, d_1, d_2) runs in $O(d_1 n)$ time in internal memory. It can be implemented in external memory with $O(n/B)$ I/Os if $n \leq M$; otherwise, it*

can be implemented with $O((nd_1/B) + (\log(d_1/d_2))(n/B)\log_{M/B}(n/B))$ *I/Os. Executing Window-Sort*(A, d_1, d_2) *results in* A *having maximum dislocation of* $d_2/2$ *with probability at least* $1 - 2n2^{-d_2/8}$*, where* $d_2 = d_1/2^k$*, for some integer* $k \geq 1$.

Proof. For the internal-memory running time, note that we can perform the deterministic sorting step using any efficient sorting algorithm in $O(n \log n)$ time. The running times for the windowed comparison steps (step 3 of Algorithm 1) form a geometric sum adding up to $O(d_1 n)$ and the total time for all the deterministic sorting steps (step 4 of Algorithm 1) is $O((\log(d_1/d_2))n \log n)$, which is at most $O(d_1 n)$ for $d_1 \geq \log n$. For the external-memory model in both the cache-aware and cache-oblivious settings, a cache-efficient sorting algorithm can be used, requiring at most $O(\log(d_1/d_2))(n/B)\log_{M/B}(n/B))$ I/Os for all the sorting steps. The scanning step can also be done in an cache efficient way, requiring at most $O(nd_1/B)$ I/Os.

For the maximum dislocation bound, note once $w = 2d_2$ and the array A prior to this iteration has maximum dislocation at most d_2, then it will result in having maximum dislocation at most $d_2/2$ with probability at least $1 - n2^{-d_2/8}$, by Lemma 1. Thus, by a union bound, the overall failure probability is at most

$$n\left(2^{-d_2/8} + 2^{-2d_2/8} + 2^{-4d_2/8} + \cdots + 2^{-d_1/8}\right) < n2^{-d_2/8}\sum_{i=0}^{\infty} 2^{-i} = 2n2^{-d_2/8}.$$

□

3 Window-Merge-Sort

In this section, we describe a simple external-memory algorithm for sorting with noisy comparisons, which achieves a maximum dislocation of $O(\log n)$. The number of I/Os for this algorithm is optimal. As is common (see, e.g., [1]), we assume that the block size is at least logarithmic in the problem size, i.e., $B \geq \log n$.

Our window-merge-sort method is a windowed version of merge sort; hence, it is deterministic. Suppose we are given an array, A, of n elements (to keep track of the original input size, we use n to denote the original size of A, and N to denote the size of the subproblem we are currently working on recursively). We take as input another parameter d, which determines the resulting maximum dislocation after running the algorithm.

For expository reasons, we first describe an internal-memory method that runs in $O(n \log^2 n)$ time and then we show how to generalize this method to an efficient external-memory method that uses an optimal number of I/Os. We give the pseudo-code for this method in Algorithm 2, with $d = c \log n$ for a constant $c \geq 1$ set in the analysis.

Algorithm 2: Window-Merge-Sort($A = \{a_0, a_1, \ldots, a_{N-1}\}, n, d$)

```
if N ≤ 6d then
    return Window-Sort(A, 4d, d)
Divide A into two subarrays, A_1 and A_2, of roughly equal size
Window-Merge-Sort(A_1, n, d)
Window-Merge-Sort(A_2, n, d)
Let B be an initially empty output list
while |A_1| + |A_2| > 6d do
    Let S_1 be the first min{3d, |A_1|} elements of A_1
    Let S_2 be the first min{3d, |A_2|} elements of A_2
    Let S ← S_1 ∪ S_2
    Window-Sort(S, 4d, d)
    Let B' be the first d elements of (the near-sorted) S
    Add B' to the end of B and remove the elements of B' from A_1 and A_2
Call Window-Sort(A_1 ∪ A_2, 4d, d) and add the output to the end of B
return B
```

Our method begins by checking if the current problem size, N, satisfies $N \leq 6d$, in which case we're done. Otherwise, if $N > 6d$, then we divide A into 2 subarrays, A_1 and A_2, of roughly equal size and recursively approximately sort each one. For the merge of the two sublists, A_1 and A_2, we inductively assume that A_1 and A_2 have maximum dislocation at most $3d/2 = (3c/2)\log n$. We then copy the first $3d$ elements of A_1 and the first $3d$ elements of A_2 into a temporary array, S, and we note that, by our induction hypothesis, S contains the smallest $3d/2$ elements currently in A_1 and the smallest $3d/2$ elements currently in A_2. We then call Window-Sort$(S, 4d, d)$, and copy the first d elements from the output of this window-sort to the output of the merge, removing these same elements from A_1 and A_2. Then we repeat this merging process until we have at most $6d$ elements left in $A_1 \cup A_2$, in which case we call window-sort on the remaining elements and copy the result to the output of the merge. The following lemma establishes the correctness of this algorithm.

Lemma 2. *If A_1 and A_2 each have maximum dislocation at most $3d/2$, then the result of the merge of A_1 and A_2 has maximum dislocation at most $3d/2$ with probability at least $1 - 12N2^{-d/8}$.*

Proof. By Lemma 1 and a union bound, each of the calls to window-sort performed during the merge of A_1 and A_2 will result in an output with maximum dislocation at most $d/2$, with at least the claimed probability. So, let us assume each of the calls to window-sort performed during the merge of A_1 and A_2 will result in an output with maximum dislocation at most $d/2$. Consider, then, merge step i, involving the i-th call to Window-Sort$(S, 4d, d)$, where S consists of the current first $3d$ elements in A_1 and the current first $3d$ elements in A_2, which, by assumption, contain the current smallest $3d/2$ elements in A_1 and current smallest $3d/2$ elements in A_2. Thus, since this call to window-sort results in

an array with maximum dislocation at most $d/2$, the subarray, B_i, of the d elements moved to the output in step i includes the $d/2$ current smallest elements in $A_1 \cup A_2$. Moreover, the first $d/2$ elements in B_i have no smaller elements that remain in S. In addition, for the $d/2$ elements in the second half of B_i, let S' denote the set of elements that remain in S that are smaller than at least one of these $d/2$ elements. Since the output of Window-Sort$(S, 4d, d)$ has maximum dislocation at most $d/2$, we know that $|S'| \leq d/2$. Moreover, the elements in S' are a subset of the smallest $d/2$ elements that remain in S and there are no elements in $(A_1 \cup A_2) - S$ smaller than the elements in S' (since S includes the $3d/2$ smallest elements in A_1 and A_2, respectively. Thus, all the elements in S' will be included in the subarray, B_{i+1}, of d elements output in merge step $i+1$. In addition, a symmetric argument applies to the first $d/2$ elements with respect to the d elements in B_{i-1}. Therefore, the output of the merge of A_1 and A_2 will have maximum dislocation at most $3d/2$ with the claimed probability. □

As an internal-memory algorithm, window-merge-sort runs in $O(n \log^2 n)$ time. To convert this algorithm to an external-memory one, we just need to make a few changes. First, rather than divide A into 2 subarrays for the recursive calls, we divide A into $m = \Theta(M/B) \geq 2$ subarrays, $A_1, A_2, \ldots, A_m$, each of roughly equal size, and recursively sort each one. For the merge step, we bring the first $\max\{3d, |A_i|\}$ elements from each A_i, group them together into a list, S, and call Window-Sort$(S, 4md, d)$ on this list, performing this computation entirely in internal memory (so it does not require any additional I/Os). Then we output the first d elements from this window-sort, and continue as in Algorithm 2. This implies the following.

Lemma 3. *If $A_1, A_2, \ldots, A_m$ each have maximum dislocation at most $3d/2$, then the result of the their merge has maximum dislocation at most $3d/2$ with probability at least $1 - 6mN2^{-d/8}$.*

Proof. The proof follows by similar arguments used in the proof of Lemma 2. □

This gives us the following.

Theorem 2. *Given an array, A, of n distinct comparable elements, one can deterministically sort A in internal memory in $O(n \log^2 n)$ time or in external memory with $O((n/B) \log_{M/B}(n/B))$ I/Os subject to comparison errors with probability $p_e \leq 1/16$, so as to have maximum dislocation of $O(\log n)$ w.h.p., assuming $B \geq \log n$.*

4 Window Funnelsort

In this section we describe WindowFunnelSort (see Algorithm 3), a noise-tolerant version of the Funnelsort algorithm that sorts n distinct comparable elements so as to have at most $O(\log n)$ maximum dislocation, with $W(n) = O(n \log^2 n)$ work complexity and $Q(n) = O(1 + (n/B)(1 + \log_M n))$ cache complexity, which matches the lower bound of $\Omega(\frac{n}{B} \log_{M/B} \frac{n}{B})$ for sorting in the

external-memory model. In our pseudocode, n denotes the original input size, while N denotes the size of the input array given to each function call, which can be less than n during recursive calls.

We require a stronger assumption on the cache size for our algorithm: in addition to the tall-cache assumption $M = \Omega(B^2)$, we also require that the block size B be at least logarithmic in the problem size, i.e. $B \geq \gamma \log n$ for some constant $\gamma > 0$ that will be determined in the analysis. In our analysis, we follow the same general proof structure used in [6] with the ideal cache model. For the remainder of this section, we assume that the maximum dislocation bound we would like to obtain, d, is $(3c/2) \log n$ for some constant $c > 0$ that will be determined later. The main difference in our analysis compared to [6] is that in the recursive definition of a k-merger, we define the base cases differently such that each base case k-merger will now use a similar merging method to the one used in Algorithm 2, and our base cases are defined over multiple values of k, instead of just $k = 2$ as done in [6].

Algorithm 3: Window-Funnel-Sort($A = \{a_0, a_1, \ldots, a_{N-1}\}, n$)

```
if N ≤ (c log n)^{3/2} then
    return WindowSort(A, N, c log n)
Divide A into N^{1/3} subarrays, A_1 ..., A_{N^{1/3}}, each of size N^{2/3}
for i = 1, ..., N^{1/3} do
    A_i = Window-Funnel-Sort(A_i)
A ← output of merging A_1 ..., A_{N^{1/3}} using a N^{1/3}-merger, as described in
  Section 4
return A
```

We first describe how to construct a k-merger, which is defined recursively in terms of smaller mergers. We follow the same general structure for constructing a k-merger in the original Funnelsort algorithm [6], however in the recursive definition of a k-merger, instead of having $k = 2$ as the base case, we view k-mergers with $\sqrt{c \log n} \leq k < c \log n$ as base cases. As an invariant, each k-merger outputs the next k^3 elements of the approximately sorted sequence obtained by merging its k input sequences.

Our base case k-merger works similarly to the merging procedure in Algorithm 2. We read in $3c \log n$ elements from each of the k inputs into an array S, call WindowSort($S, 4kc \log n, c \log n$), then output the last $c \log n$ elements from this call. Then, we replace the $c \log n$ elements in the k-merger that were just written to the output as follows: for each element e written to the output, we read into the k-merger a new element from the input queue that e belonged to. We then call WindowSort again on this updated set of elements, and repeat this process until the k-merger has outputted k^3 elements.

For all other values of $k \geq c \log n$, a k-merger will work the same way as in [6], which we describe here for completeness. A (non-base case) k-merger is

built recursively out of $\sqrt{k}$-mergers by first partitioning the k inputs into $\sqrt{k}$ sets of $\sqrt{k}$ elements, which forms the input to $\sqrt{k}$ left mergers $L_1, L_2, \ldots, L_{\sqrt{k}}$, each of which is a $\sqrt{k}$-merger. Each L_i is connected to an output buffer i, implemented as a circular queue that can hold up to $2k^{3/2}$ elements. Each buffer is then connected as input to R, which is another $\sqrt{k}$-merger. The output of R then becomes the output of the whole k-merger. Following our invariant, in order to output k^3 elements, the k-merger will invoke R $k^{3/2}$ times. Since the input queues connected to R might become empty, the k-merger first fills all buffers that have less than $k^{3/2}$ elements before each invocation of R, which is done by invoking the corresponding left merger L_i that connects to buffer i. Since each left merger invocation will output $k^{3/2}$ elements to the corresponding buffer, each L_i will only need to be invoked at most once before each invocation of R.

Let us first consider the cache complexity of WindowFunnelSort. Following the proof in [6], we first consider how much space a k-merger requires.

Lemma 4. *A k-merger requires at most $O(k^2)$ contiguous memory locations when $k \geq c \log n$.*

Proof. A k-merger with $k \geq c \log n$ requires $O(k^2)$ memory locations for the buffers, and it also requires space for its $\sqrt{k} + 1$ $\sqrt{k}$-mergers. Thus, the space $S(k)$ required by a k-merger satisfies the recurrence relation

$$S(k) \leq (\sqrt{k} + 1)S(\sqrt{k}) + \beta k^2,$$

for some constant $\beta > 0$. We prove inductively that $S(k) \leq Zk^2$ for some constant Z. For k-mergers with $\sqrt{c \log n} \leq k < c \log n$, we will read in $c \log n$ elements from k input queues, then perform WindowSort on them, requiring $S(k) = O(k \log n)$ space for some constant $\beta > 0$. Thus for $c \log n \leq k < (c \log n)^2$, we have $S(k) \leq (\sqrt{k} + 1)O(\sqrt{k} \log n) + \beta k^2 \leq Zk^2$ for sufficiently large Z.

For $k \geq (c \log n)^2$, we inductively have

$$\begin{aligned} S(k) &\leq (\sqrt{k} + 1)S(\sqrt{k}) + \beta k^2 \\ &\leq (\sqrt{k} + 1)Zk + \beta k^2 \leq Zk^2 \end{aligned}$$

for sufficiently large Z. Thus we have $S(k) = O(k^2)$ for any $k \geq c \log n$. □

Any k-merger with $\sqrt{c \log n} \leq k < c \log n$ reads in $3c \log n$ elements from less than $c \log n$ inputs, and will require $O(\log^2 n)$ space in total. Therefore we require that the block size B is at least $\gamma \log n$ for an appropriate constant $\gamma > 0$ such that after applying the tall-cache assumption $M = \Omega(B^2)$, any k-merger with $\sqrt{c \log n} \leq k < c \log n$ will fit inside the cache. Therefore, more generally, through Lemma 4, any k-merger with $\sqrt{c \log n} \leq k \leq c \log n \leq \alpha\sqrt{M}$, where α is a sufficiently small constant, will also fit inside the cache and run without any additional cache misses.

The following lemma, which is proved in [6], shows that the $\sqrt{k}$ buffers used in a k-merger can be managed cache-efficiently.

Lemma 5 (Lemma 4.2. in [6]). Performing r insert and remove operations on a circular queue causes $O(1 + r/B)$ cache misses if two cache lines are available for the buffer.

We now bound the cache complexity Q_k of one invocation of a k-merger.

Lemma 6 *One invocation of a k-merger incurs*

$$Q_k = O(k + k^3/B + k^3 \log_M k/B)$$

cache misses.

Proof We first consider the case $\sqrt{c \log n} \leq k \leq \alpha\sqrt{M}$. From Lemma 4 and our assumption on the cache size, we know that any k-merger with $\sqrt{c \log n} \leq k \leq \alpha\sqrt{M}$ will fit inside the cache and run with no additional cache misses. Each k-merger has k input queues, and loads a total of $O(k^3)$ elements. Let r_i denote the number of elements extracted from the ith queue. Since $k \leq \alpha\sqrt{M}$ and $B = O(\sqrt{M})$, there are at least $M/B = \Omega(k)$ cache lines available for the input buffers. Thus, through Lemma 5, the total number of cache misses for accessing the input queues is

$$\sum_{i=1}^{k} O(1 + r_i/B) = O(k + k^3/B).$$

Similarly, the cache complexity of writing to the output queue is $O(1 + k^3/B)$. The k-merger incurs an additional $O(k^2/B)$ cache misses through using its internal data structures, for a total of $Q_k = O(k + k^3/B)$ cache misses.

We then consider the case $k > \alpha\sqrt{M}$. We prove by induction that $Q(k) \leq (Zk^3 \log_M k)/B - A(k)$ for some constant $Z > 0$, where $A(k) = o(k^3)$. We first verify that values of $\alpha M^{1/4} < k \leq \alpha\sqrt{M}$ also satisfy this inequality: from the first case, we have $Q(k) = O(k + k^3/B) = O(k^3/B)$ since $B = O(\sqrt{M}) = O(k^2)$ and $k = \Omega(1)$.

For $k > \alpha\sqrt{M}$, for a k-merger to output k^3 elements, the number of times the left mergers are invoked is bounded by $k^{3/2} + 2\sqrt{k}$. The right merger R is also invoked $k^{3/2}$ times. The k-merger also has to check before each invocation of R whether any of the buffers are empty. This requires at most $\sqrt{k}$ cache misses and is repeated exactly $k^{3/2}$ times, for a total of at most k^2 cache misses. Therefore the cache complexity Q_k of a k-merger satisfies the following recurrence relation:

$$\begin{aligned} Q_k &\leq (2k^{3/2} + 2\sqrt{k})Q_{\sqrt{k}} + k^2 \\ &\leq (2k^{3/2} + 2\sqrt{k})(\frac{Zk^{3/2} \log_M k}{2B} - A(\sqrt{k})) + k^2 \\ &\leq \frac{Z}{B}k^3 \log_M k + k^2(1 + \frac{Z}{B} \log_M k) - (2k^{3/2} + 2\sqrt{k})A(\sqrt{k}), \end{aligned}$$

which is at most $(Zk^3 \log_M k)/B - A(k)$ if $A(k) = k(1 + (2Z \log_M k)/B)$. □

Theorem 3 WINDOWFUNNELSORT *incurs at most* $Q(n)$ *cache misses, where*

$$Q(n) = O(\frac{n}{B} \log_{M/B} \frac{n}{B}).$$

Proof If $n \leq \alpha M$ for a sufficiently small constant α, the algorithm will incur at most $O(1 + n/B)$ cache misses, since only one k-merger will be active at any time, and the largest possible k-merger will require $O(n^{2/3}) < O(n)$ space. This case also covers the base case in Line 1 of Algorithm 3 through our assumption on the cache size.

If $n > \alpha M$, have the recurrence

$$Q(n) = n^{1/3} Q(n^{2/3}) + Q_{n^{1/3}}.$$

From Lemma 6, we have $Q_{n^{1/3}} = O(n^{1/3} + n/B + (n \log_M n)/B)$. Therefore the recurrence simplifies to

$$Q(n) = n^{1/3} Q(n^{2/3}) + O((n \log_M n)/B),$$

which has solution $Q(n) = O(1+(n/B)(1+\log_M n))$ by induction, which matches the $\Omega(\frac{n}{B} \log_{M/B} \frac{n}{B})$ lower bound for sorting in the external-memory model. □

We now prove that WINDOWFUNNELSORT is tolerant to persistent comparison errors.

Lemma 7 *Given* k *input queues with maximum dislocation at most* $\frac{3}{2} c \log n$ *for some constant* $c > 0$*, one invocation a* k*-merger outputs* k^3 *elements with maximum dislocation at most* $\frac{3}{2} c \log n$ *with probability at least* $1 - 2Zk^3 (c \log n)^5 2^{-(c \log n)/8}$ *for some constant* $Z > 0$*.*

Proof We first consider k-mergers with $\sqrt{c \log n} \leq k < c \log n$. Each such k-merger will call WINDOWSORT $\frac{k^3}{c \log n} < (c \log n)^2$ times, with each call working on at most $(c \log n)^2$ elements. Therefore, using a similar argument to Lemmas 2 and 3 and a union bound, the resulting sequence after $(c \log n)^2$ calls to WINDOWSORT will have maximum dislocation at most $(3c/2) \log n$ with probability at least $1 - 2(c \log n)^4 2^{-(c \log n)/8}$.

We then consider the case $k \geq c \log n$. We have $\sqrt{k}$ left $\sqrt{k}$-mergers, along with a $\sqrt{k}$-merger R. Each left merger inductively outputs $k^{3/2}$ elements with dislocation at most $\frac{3}{2} c \log n$, which is used as the input to the $\sqrt{k}$-merger R that also inductively outputs $k^{3/2}$ elements with dislocation at most $\frac{3}{2} c \log n$. Using a similar argument to Lemma 2, the output queue of the k-merger will also have dislocation at most $\frac{3}{2} c \log n$. To find the success probability, we consider the number of times WINDOWSORT is called. Since the number of invocations of smaller k-mergers is bounded by $2k^{3/2} + 2\sqrt{k}$, the number of invocations of WINDOWSORT, $I(k)$, satisfies the recurrence relation

$$I(k) = \begin{cases} (2k^{3/2} + 2\sqrt{k}) I(\sqrt{k}) & k \geq c \log n \\ 1 & \sqrt{c \log n} \leq k < c \log n, \end{cases}$$

which has solution $I(k) = Zk^3 \log k$ for some constant $Z > 0$ using a similar derivation to the one in Lemma 6. Therefore, using a union bound, the probability that a k-merger outputs k^3 elements with maximum dislocation at most $\frac{3}{2}c \log n$ is at least $1 - 2Zk^3(c \log n)^5 2^{-(c \log n)/8}$. □

Theorem 4 *Given an array A of n distinct comparable elements, and assuming $B = \Omega(\log n)$, one can deterministically sort A subject to comparison errors with probability $p_e \leq 1/16$, so as to have maximum dislocation of at most $\frac{c}{2} \log n$ for some constant $c > 0$ w.h.p., with at most $O(\frac{n}{B} \log_{M/B} \frac{n}{B})$ cache misses in the cache-oblivious model, and taking $O(n \log^2 n)$ time in a RAM model.*

Proof By induction, each of the $n^{1/3}$ input sequences given to the $n^{1/3}$-merger has maximum dislocation at most $\frac{3c}{2} \log n$ w.h.p. From Lemma 7, we have that a $n^{1/3}$ merger outputs n elements with maximum dislocation at most $\frac{3c}{2} \log n$ with probability at least $1 - 2Zn(c \log n)^5 2^{-(c \log n)/8}$ for some constant $Z > 0$. Choosing an appropriate value for c establishes this theorem. □

We now bound the work complexity of WindowFunnelSort, by first bounding the work complexity W_k of a k-merger.

Lemma 8 *The work complexity W_k of one invocation of a k-merger is $O(k^3 \log^2 n)$.*

Proof We first consider k-mergers with $\sqrt{c \log n} \leq k < c \log n$. The k-merger reads $3c \log n$ elements from k input queues, each of which have maximum dislocation at most $O(\log n)$ from Theorem 4, for a total of $3kc \log n$ elements, then performs window-sort on these elements, which takes $O(k \log^2 n)$ time. To output k^3 elements, the k-merger needs to repeat this $O(\frac{k^3}{\log n})$ times, taking a total of $O(k^4 \log n)$ time, which is bounded by $O(k^3 \log^2 n)$ since $k < c \log n$.

For k-mergers where $k \geq c \log n$, to output k^3 elements, the left mergers and right merger are invoked at most $k^{3/2} + 2\sqrt{k}$ and $k^{3/2}$ times respectively. The k-merger also has to check before each invocation of R whether any of the buffers are empty. This takes $O(\sqrt{k})$ time and is repeated exactly $k^{3/2}$ times, for a total of $O(k^2)$ time. Therefore the total work complexity $W(k)$ of a k-merger satisfies the following recurrence relation:

$$W_k \leq (2k^{3/2} + 2\sqrt{k}) W_{\sqrt{k}} + O(k^2).$$

Using a derivation similar to the one in Lemma 6, we can show that $W_k = O(k^3 \log^2 n)$ by induction. □

Theorem 5 *The work complexity $W(n)$ of* WindowFunnelSort *is $O(n \log^2 n)$ for any input sequence of n elements.*

Proof We have the recurrence

$$W(n) = n^{1/3} W(n^{2/3}) + W_{n^{1/3}}.$$

From Lemma 8, we have $W_{n^{1/3}} = O(n \log^2 n)$. Therefore the recurrence simplifies to

$$W(n) = n^{1/3} W(n^{2/3}) + O(n \log^2 n),$$

which has solution $W(n) = O(n \log^2 n)$ by induction. □

5 Conclusions and Future Work

We provided efficient sorting algorithms that tolerate noisy comparisons and are cache efficient in both cache-aware and cache-oblivious external memory models. In [6], the authors introduced another cache-oblivious sorting algorithm based on distribution-sort, that has the same work and cache complexities as funnel-sort. One direction for future work could be to design and analyze a windowed version of the cache-oblivious distribution sort algorithm that has similar bounds on the work and cache complexities.

Acknowledgements. We would like to graciously thank Riko Jacob and Ulrich Meyer for several helpful discussions regarding the topics of this paper. This work was partially support by NSF Grant 2212129.

References

1. Bender, M., Demaine, E., Farach-Colton, M.: Cache-oblivious B-trees. In: 41st IEEE Symposium on Foundations of Computer Science (FOCS), pp. 399–409 (2000)
2. Braverman, M., Mossel, E.: Noisy sorting without resampling. In: 19th ACM-SIAM Symposium on Discrete Algorithms (SODA), pp. 268–276 (2008)
3. Brodal, G.S., Fagerberg, R.: Cache oblivious distribution sweeping. In: Widmayer, P., Eidenbenz, S., Triguero, F., Morales, R., Conejo, R., Hennessy, M. (eds.) ICALP 2002. LNCS, vol. 2380, pp. 426–438. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-45465-9_37
4. Brodal, G.S., Fagerberg, R., Vinther, K.: Engineering a cache-oblivious sorting algorithm. ACM J. Exp. Algorithmics **12**, 1–23 (2008). https://doi.org/10.1145/1227161.1227164
5. Feige, U., Raghavan, P., Peleg, D., Upfal, E.: Computing with noisy information. SIAM J. Comput. **23**(5), 1001–1018 (1994)
6. Frigo, M., Leiserson, C.E., Prokop, H., Ramachandran, S.: Cache-oblivious algorithms. ACM Trans. Algorithms **8**(1), 1–22 (2012). https://doi.org/10.1145/2071379.2071383
7. Geissmann, B., Leucci, S., Liu, C.H., Penna, P.: Sorting with recurrent comparison errors. In: Okamoto, Y., Tokuyama, T. (eds.) 28th International Symposium on Algorithms and Computation (ISAAC). LIPIcs, vol. 92, pp. 38:1–38:12 (2017)
8. Geissmann, B., Leucci, S., Liu, C.H., Penna, P.: Optimal sorting with persistent comparison errors. In: Bender, M.A., Svensson, O., Herman, G. (eds.) 27th European Symposium on Algorithms (ESA). LIPIcs, vol. 144, pp. 49:1–49:14 (2019)

9. Geissmann, B., Leucci, S., Liu, C.H., Penna, P.: Optimal dislocation with persistent errors in subquadratic time. Theory Comput. Syst. **64**(3), 508–521 (2020). This work appeared in preliminary form in STACS 2018
10. Gilotte, A., Calauzènes, C., Nedelec, T., Abraham, A., Dollé, S.: Offline A/B testing for recommender systems. In: Eleventh ACM International Conference on Web Search and Data Mining (WSDM), pp. 198–206. New York (2018). https://doi.org/10.1145/3159652.3159687
11. Karp, R.M., Kleinberg, R.: Noisy binary search and its applications. In: 18th ACM-SIAM Symposium on Discrete Algorithms (SODA), pp. 881–890 (2007)
12. Khadiev, K., Ilikaev, A., Vihrovs, J.: Quantum algorithms for some strings problems based on quantum string comparator. Mathematics **10**(3), 377 (2022)
13. Klein, R., Penninger, R., Sohler, C., Woodruff, D.P.: Tolerant algorithms. In: Demetrescu, C., Halldórsson, M.M. (eds.) ESA 2011. LNCS, vol. 6942, pp. 736–747. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-23719-5_62
14. Leighton, T., Ma, Y., Plaxton, C.G.: Breaking the $\Theta(n \log^2 n)$ barrier for sorting with faults. J. Comput. Syst. Sci. **54**(2), 265–304 (1997)
15. Mao, C., Weed, J., Rigollet, P.: Minimax rates and efficient algorithms for noisy sorting. In: Janoos, F., Mohri, M., Sridharan, K. (eds.) Proceedings of Algorithmic Learning Theory. Proceedings of Machine Learning Research, vol. 83, pp. 821–847 (2018)
16. Pelc, A.: Searching with known error probability. Theor. Comput. Sci. **63**(2), 185–202 (1989)
17. Pelc, A.: Searching games with errors–fifty years of coping with liars. Theor. Comput. Sci. **270**(1), 71–109 (2002)
18. Rényi, A.: On a problem in information theory. Magyar Tud. Akad. Mat. Kutató Int. Közl. **6**, 505–516 (1961). https://mathscinet.ams.org/mathscinet-getitem?mr=0143666
19. Vitter, J.S.: External memory algorithms and data structures: dealing with massive data. ACM Comput. Surv. **33**(2), 209–271 (2001)
20. Wang, J., Huang, P., Zhao, H., Zhang, Z., Zhao, B., Lee, D.L.: Billion-scale commodity embedding for e-commerce recommendation in Alibaba. In: 24th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD), pp. 839–848. Association for Computing Machinery, New York (2018). https://doi.org/10.1145/3219819.3219869
21. Wang, Z., Ghaddar, N., Wang, L.: Noisy sorting capacity. arXiv abs/2202.01446 (2022)
22. Xu, Y., Chen, N., Fernandez, A., Sinno, O., Bhasin, A.: From infrastructure to culture: A/B testing challenges in large scale social networks. In: 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD), pp. 2227–2236 (2015)