Optimal Parallel Sorting with Comparison Errors

Michael T. Goodrich
University of California, Irvine
Irvine, CA, USA
goodrich@uci.edu

Riko Jacob
IT University of Copenhagen
Copenhagen, Denmark
rikj@itu.dk

ABSTRACT

We present comparison-based parallel algorithms for sorting n comparable items subject to comparison errors. We consider errors to occur according to a well-studied framework, where the comparison of two elements returns the wrong answer with a fixed probability. In the ***persistent*** model, the result of the comparison of two given elements, x and y, always has the same result, and is independent of all other pairs of elements. In the ***non-persistent*** model, the result of the comparison of each pair of elements, x and y, is independent of all prior comparisons, including for x and y. It is not possible to always correctly sort a given input set in the persistent model, so we study algorithms that achieve a small maximum dislocation and small total dislocation of the elements in the output permutation. In this paper, we provide parallel algorithms for sorting with comparison errors in the persistent and non-persistent models. Our algorithms are asymptotically optimal in terms of their span, work, and, in the case of persistent errors, maximum and total dislocation. The main results are algorithms for the binary-forking parallel model with atomics, but we also provide algorithms for the CREW PRAM model. Our algorithms include a number of novel techniques and analysis tools, including a PRAM-to-binary-forking-model simulation result, and are the first optimal parallel algorithms for the persistent model and the non-persistent model in the binary-forking parallel model with atomics. In particular, our algorithms have $O(\log n)$ span, $O(n \log n)$ work, and, in the case of the persistent model, $O(\log n)$ maximum dislocation and $O(n)$ total dislocation, with high probability. We achieve similar results for the CREW PRAM model, which are the first optimal methods for the persistent model and the first optimal results for the non-persistent model with reasonable constant factors in the performance bounds.

CCS CONCEPTS

• **Theory of computation**;

KEYWORDS

sorting, noisy searching, algorithm analysis, randomized algorithms

SPAA '23, June 17–19, 2023, Orlando, FL, USA

ACM ISBN 978-1-4503-9545-8/23/06…$15.00
https://doi.org/10.1145/3558481.3591093

ACM Reference Format:
Michael T. Goodrich and Riko Jacob. 2023. Optimal Parallel Sorting with Comparison Errors. In *Proceedings of the 35th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '23), June 17–19, 2023, Orlando, FL, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3558481.3591093

1 INTRODUCTION

Given a list, L, of n distinct elements, we study the problem of efficiently sorting L in parallel when comparisons can have random errors. In this framework, which has been extensively studied [6, 10, 13–15, 19, 21–23, 25, 29, 35], when a sorting algorithm performs the comparison of two elements, x and y, a random adversary flips a coin that is "heads" with fixed probability, independent of other pairwise comparisons, and he returns the correct result of the comparison if and only if the coin comes up "tails". In the case of ***persistent*** errors [6, 13–15, 22], the adversary flips the coin once for each pair of compared elements, (x, y), and always returns the same result for the comparison of x and y. In the case of ***non-persistent*** errors [10, 19, 21, 23, 29, 35], however, the adversary flips the coin independently for each comparison, even for the same x and y.

Motivation for sorting with comparison errors comes from multiple sources, including quantum computing [21] and applied cryptography [11, 24]. For example, quantum comparison gates and cryptographic comparison protocols can fail with known probabilities [11, 21, 24, 38]. In both cases, reducing the noise from comparison errors is expensive, and the framework advanced here offers an alternative, possibly more efficient approach, where a higher error rate is tolerated while still achieving the goal of sorting or near-sorting with high probability. Moreover, parallel sorting algorithms with comparison errors imply an efficient number of rounds of computation/communication in a protocol, since parallel steps corresponds to such rounds. Other applications of sorting with comparison errors include ranking sports teams via pairwise matches [6] and ranking objects in social networks via group A/B testing [37], both of which are inherently noisy.

Of course, with non-persistent errors, one can simply repeat each comparison $\Theta(\log n)$ times and take the majority of the results, which will be correct with high probability.[1] Thus, with non-persistent errors, the challenge is to design a sorting algorithm that correctly sorts L without incurring a multiplicative logarithmic overhead.

With persistent errors, however, it is impossible to always correctly sort. Indeed, given $\binom{n}{2}$ distinct comparison results that can be arbitrarily faulty, the problem of finding a permutation that minimizes the number of contradictory comparisons, i.e., the permutation of the input elements that has the highest probability

[1] We say that an event regarding n objects occurs ***with high probability*** if it occurs with probability at least $1 - 1/n$.

to produce the observed comparisons, is equivalent to the feedback arc problem, which is known to be NP-complete [12]. We don't persue this approach, however, and instead follow the formulation of Geissmann, Leucci, Liu, and Penna [13–15], which is based on a dislocation distance metric. Define the ***dislocation*** of an element, x, in a list, L, as the absolute value of the difference between x's index in L and its index in the correctly sorted permutation of L. Further, define the ***maximum dislocation*** of L as the maximum dislocation that some element in L has, and define the ***total dislocation*** of L is the sum of the dislocations of the elements in L. In [13–15], a lower bound is presented, that shows, for persistent comparison errors, that a maximum dislocation of $O(\log n)$ and a total dislocation of $O(n)$ is optimal.

In this paper, we are interested in parallel algorithms for sorting with comparison errors, in both the persistent and non-persistent error models. The models of parallel computation that we consider are the parallel random access machine (PRAM) and binary-forking models. The ***PRAM*** is a shared-memory model where processors operate synchronously, and it has variants that are distinguished by how simultaneous memory accesses are handled (see, e.g., [18, 20]), including (i) allowing for concurrent reads and concurrent writes (CRCW), according to some write conflict-resolution method (e.g., arbitrary), (ii) a more-realistic variant allowing for concurent reads but requiring exclusive writes (CREW), and (iii) a more-restrictive variant requiring exclusive reads and exclusive writes (EREW). The PRAM model can be viewed as a hardware-inspired model, in that it defines the actions of parallel processors. In contrast, the ***binary-forking model*** can be viewed as a software-inspired parallel model, in that it is a loosely-synchronous shared-memory model that mimicks how parallelism is often provided in programming languages for multicore shared-memory machines, where computational threads are created with a ***fork*** operation and synchronzied back into one by a ***join***, e.g., see [1, 5, 8, 17]. The binary-forking model allows for asynchronous concurrent memory reads and supports two submodels based on whether asynchronous concurrent writes are allowed through the use of ***atomic*** operations, such as test-and-set (see, e.g., [1, 5]), or whether such atomic operations are disallowed (see, e.g., [8, 17]).

In both parallel models, costs are measured in terms of ***work*** (the total number of executed instructions) and ***span*** (the longest sequence of logically dependent instructions), the latter of which is also known as parallel ***time*** in the PRAM model. By known lower bounds (see, e.g., [5, 18, 20]), an optimal parallel comparison-based sorting algorithm in either the CREW PRAM or binary-forking models must have $\Omega(n \log n)$ work and $\Omega(\log n)$ span. In this paper, we are interested in algorithms that are asymptotically optimal in terms of span, work, and, in the case of persistent errors, maximum and total dislocation.

1.1 Prior Results

The non-persistent error model traces its roots to a classic problem from 1961 by Rényi [33] of posing yes/no questions to someone who lies with a given probability. The literature on this topic is too voluminous to completely review here, however; hence, the reader is referred to a survey by Pelc [30]. Notable work in this space includes work by Pippenger [31] on computing Boolean functions with probabilistically noisy gates and by Yao and Yao [39] on sorting networks built from noisy comparators. Braverman and Mossel [6] 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 for sorting. Klein, Penninger, Sohler, and Woodruff [22] improve the running time to $O(n^2)$, but with $O(n \log n)$ total dislocation w.h.p. The sequential running time for sorting in the persistent-error model optimally with respect to maximum and total dislocation was subsequentially 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 [13–15]. In terms of the relevant prior ***parallel*** results, Feige, Raghavan, Peleg, and Upfal [10] provide a randomized parallel algorithm for sorting with non-persistent errors that, with high probability, runs in $O(\log n)$ parallel time and uses $O(n \log n)$ work in the CRCW PRAM model, and Leighton, Ma, and Plaxton [23] show how to achieve these bounds in the EREW PRAM model. Both of these algorithms make extensive use of the AKS sorting network [2, 3], however, which implies that their asymptotic performance bounds have very large constant factors. We are not familiar with any optimal parallel algorithm for sorting with non-persistent errors that avoids using the AKS sorting network or is for the binary-forking parallel model. We are also not familiar with any previous efficient parallel algorithm for sorting with persistent errors. Optimistically parallelizing the best sequential algorithm for sorting with persistent errors, by Geissmann, Leucci, Liu, and Penna [14], seems to require at least $\Omega(\log^3 n)$ span; hence, this approach would not achieve an optimal parallel sorting algorithm.

1.2 Our Results

In this paper, we describe parallel sorting algorithms in the CREW PRAM and binary-forking parallel models that have $O(\log n)$ span and $O(n \log n)$ work w.h.p., in the presence of either random persistent or non-persistent comparison errors according to a fixed probability. In the case of non-persistent errors, our algorithms correctly sort w.h.p., and in the case of persistent errors our algorithms return a permutation of the input that achieves maximum dislocation of $O(\log n)$ and total dislocation of $O(n)$ w.h.p.

Our algorithms are inspired by the ***box-sort*** algorithm [28, 32], which is a type of parallel quick-sort that is also known as ***distribution-sort*** [34] or ***sample-sort*** [5, 8]. At a high level, box-sort involves choosing a sample, S, of $n^{1/k}$ elements at random (e.g., for $k = 2$ or $k = 3$), sorting S by brute-force, and performing a binary search for each unchosen element, x, in parallel to determine the "box" x belongs to, where a box is defined by each pair of consecutive elements in the sorted copy of S. Box-sort completes by distributing the unchosen elements to their respective boxes (which is admittedly a non-trivial step) and recursively sorting each box in parallel.

Implementing an algorithm inspired by box-sort while dealing with comparison errors presents a number of interesting challenges, for which we present novel techniques to overcome. For example, brute-force sorting is not even possible with 100% accuracy with presistent errors. More importantly, it is not clear how to efficiently perform noisy binary searching w.h.p. with either persistent or non-persistent errors for subproblems (defined at the later stages

of the algorithm) that are much smaller than the size of the original input list. Finally, simple recursion for subproblems seems like it could "lock in" elements far from their proper location in a sorted output, so simple recursion also appears to be out.

We overcome these challenges by developing a number of tools, to design a parallel algorithm that has $O(\log n)$ span using a sequence of rounds (each a parallel algorithm), each of which significantly reduces the maximum dislocation of the list being sorted. For example, one of our tools is a parallel noisy multi-way search that is guaranteed to come close to the proper location of a query element, x, using a reasonably small number of processors/threads. We also give a parallel error-reduction technique that uses parallel sorting-with-small-radius to reduce a maximum dislocation from a bound, k, to $k^{7/8}$ w.h.p. This parallel sorting-with-small-radius is done on annotations rather than original elements; hence, it can be done deterministically and error-free. For the CREW PRAM model, for example, we can utilize optimal deterministic parallel sorting, such as the parallel merge-sort algorithm by Cole [7], here. However, for the binary-forking model, which is our primary interest, we are not aware of any published optimal deterministic error-free parallel sorting algorithms. The best prior-work results appear to be the classic odd-even merge-sort [4], which has $O(\log^2 n)$ span and $O(n \log^2 n)$ work, a determinsitic method by Cole and Ramachandran [8], which has $O(\log n \log\log n)$ span and $O(n \log n)$ work, and the $O(\log n)$-span randomized algorithm by Blelloch, Fineman, Gu, and Sun [5], none of which are sufficient for our purposes. We overcome this challenge by providing a simulation result that implies an optimal deterministic error-free sorting algorithm in the binary-forking model (with atomic operations).

2 BUILDING BLOCKS

2.1 Rank Estimation

Suppose we are given a list, L, of n distinct comparable elements. By a slight abuse of notation, since the elements in L are distinct, we may simultaneously view L as a set and as an indexed list.[2] Given an element, x, which may or may not be in L, define $\text{rank}(x, L)$ to be the number of elements in L smaller than x. If the list L is understood by context, then we may simply use $\text{rank}(x)$ instead of $\text{rank}(x, L)$. Thus, if x is the smallest element in L, then $\text{rank}(x) = 0$, and if x is the largest element in L, then $\text{rank}(x) = n - 1$. In other words, $\text{rank}(x)$ is equal to the index of x in the correctly sorted listing of $L \cup \{x\}$. Also, note that if $x = L[i]$, i.e., x is the element at index i in L, then the dislocation of x in L is $|\text{rank}(x) - i|$.

Lemma 2.1 (Rank estimate from comparison count). *Assume we perform the comparisons of one element x with n other elements x_i. Let r be the true rank of x, i.e., the number of comparisons that would turn out "smaller" if there were no errors. Assume S out of the n comparisons come out "smaller" in the persistent-error model with error probability $p < 1/2$. If we compute a maximum likelihood estimate of the (0-based) rank r of x in S as $r_n(S) = \min\{n, \max\{0, (S-np)/(1-2p)\}\}$, then, for any $t \geq 1$, $\Pr(|r_n(S) - r| \geq t) \leq 2\exp(-2(t^2(1-2p)^2)/n)$.*

Proof. Note that $E[S] = r(1-p) + (n-r)p = r - rp + np - rp = r(1-2p) + np$. Let S' denote the sum of indicator variables X_i with value 0 or $1/(1-2p)$, such that if $x_i < x$, then X_i is $1/(1-2p)$ iff there is a true comparison for x_i, and if $x_i > x$, then X_i is $1/(1-2p)$ iff there is a false comparison for x_i. Thus, $S' = S/(1-2p)$; hence, $E[S'] = r + np/(1-2p)$. Insisting on $r_n(S)$ being at most n and at least 0 can only move it closer to r. Hence, instead of $r_n(S)$, for our analysis, we may conservatively use $R' = r'_n(S) = (S - np)/(1-2p) = S/(1-2p) - np/(1-2p) = S' - np/(1-2p)$, thereby only strengthening the result. Thus, by linearity of expectation, $E[R'] = E[S'] - np/(1-2p) = r + np/(1-2p) - np/(1-2p) = r$. We may then apply a Hoeffding bound (see appendix) as follows:

$$\begin{aligned}\Pr\left(|R' - r| \geq t\right) &= \Pr\left(\left|S' - \left(r + \frac{np}{1-2p}\right)\right| \geq t\right) \\ &\leq 2\exp\left(-\frac{2t^2}{n/(1-2p)^2}\right).\end{aligned}$$

◀

2.2 Deterministic Error-Free Sorting in the Binary-Forking Model

For the remainder of this paper, we assume that we are working in the version of the binary-forking model that supports atomic operations, such as test-and-set; see, e.g., [1, 5]. We are not aware of an optimal determistic error-free sorting algorithm for this model, given that the algorithm of by Cole and Ramachandran [8] (which does not use atomic operations) has $O(\log n \log\log n)$ span and the $O(\log n)$-span algorithm by Blelloch, Fineman, Gu, and Sun [5] (which does use atomic operations) is not deterministic. In Appendix B, we prove the following simulation result.

Theorem 2.2. *If $\mathcal{A}$ is an EREW PRAM algorithm where each memory cell is written to and subsequently read at most once, such that $\mathcal{A}$ runs in T time using P processors and W work, then one can simulate $\mathcal{A}$ in the binary-forking model (with atomic operations) with $O(T + \log P)$ span and $O(W)$ work.*

This gives us the following result, which we will use as a subroutine in our algorithms for the persistent and non-persistent error models.

Corollary 2.3. *We can deterministically sort n elements (with error-free comparisons) in $O(\log n)$ span and $O(n \log n)$ work in the binary-forking model (with atomic operations).*

Proof. Cole's EREW PRAM parallel mergesort algorithm can be described so that each memory cell is written to and subsequently read at most once (see, e.g., § 5 in [7]), with a running time of $O(\log n)$ using $O(n)$ processors. The proof follows by Theorem 2.2. ◀

Given that our main results are for the binary-forking parallel model, for the remainder of this paper, we describe our algorithms generically and state their performance in both this model and in the CREW PRAM model. In this unified exposition, we sometimes make the assumption that there is a thread for every element of an array (or every kth element by index). For example, this means that a computation filling an array starts the corresponding threads. Using the simulation ideas from Theorem 2.2, we assume these started programs and their threads stop and put their state into the memory cell if an entry of the array is not computed yet.

[2] Throughout this paper we assume list indexing starts at 0.

2.3 Noisy Searching

Another tool we use is noisy searching an approximately sorted list. We begin by reviewing a previous result for noisy binary search.

Theorem 2.4 (from [14]). *Let S be a sequence of n elements having maximum dislocation at most $d \geq \log n$ and let $x \notin S$. Under the persistent error model, with $p < 1/32$, an index r_x such that $r_x \in [\text{rank}(x,S) - \alpha d, \text{rank}(x,S) + \alpha d]$ can be found in $O(\log n)$ (serial) time with probability at least $1 - O(n^{-6})$, where $\alpha > 1$ is an absolute constant.*

In this section, we prove a theorem (2.7) that generalizes and extends noisy binary search to the parallel setting. Let S be a sequence of n elements having maximum dislocation at most d, a parameter λ with $d \geq \lambda \geq \log n$ and let $x \notin S$. Here, the lower bound $\log n$ simplifies the exposition as taking floors or ceilings is going to be irrelevant for asymptotic considerations, so, for the sake of simplifying the analysis, we assume d divides λ. In addition, suppose we have at least $\lambda^{1+\varepsilon}$ processors (i.e., threads) assigned to x, where $\varepsilon > 0$ is any fixed constant. Our goal is to use these processors to perform a search in S to determine an estimate for $\text{rank}(x,S)$.

Our method is as follows. We begin by conceptually organizing S into a sequence of contiguous chunks, each of size λ, according to the ordering of the elements in S, and numbering the chunks left-to-right as $1, 2, \ldots$.[3] For each chunk, C, let $\text{range}(C)$ denote the interval, $[a,b]$, which we call the ***range*** of C, such that a is the true smallest element in C and b is the true largest element in C. Let S' denote the subsequence of chunks numbered with a multiple of $3\lceil d/\lambda \rceil$, e.g., if $d = \lambda$, then the chunks are numbered 3, 6, 9, and so on. Note that the ranges of the chunks in S' are disjoint, by the assumed dislocation. That is, each element in a chunk in S' is actually less than every element in a higher-numbered chunk in S'.

We perform a parallel search on S' that is based on majority comparisons with whole chunks, succeeding with high probability, as made precise in the following lemma. See Figure 1.

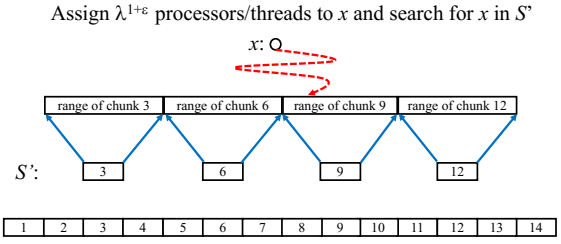

Figure 1: Our approach to noisy parallel searching, for the case when $d = \lambda$.

Lemma 2.5. *Suppose $p \leq 1/4e$ is the probability of erroneous comparisons. If x is less than every element in a chunk, C, or x is greater than every element in C, then a majority of comparisons of x with elements in C with $|C| = \lambda$ will reflect this fact with probability at least $1 - 2^{-\lambda/2}$.*

[3] If n is not a multiple of λ, then we pad the last chunk with $+\infty$ values.

Proof. Let X denote the number of false comparisons of x with the elements in C, and note that $E[X] = \lambda p$. Thus, since these comparisons are mutually independent, and noting that $(1/2p) \geq 2e$, by a Chernoff bound (see appendix), we have the following:

$$\Pr(X \geq \lambda/2) = \Pr(X \geq (1/2p)\lambda p) \leq 2^{-\lambda/2}.$$

◀

In the parallel search, we maintain two boundary chunks r, l, such that with high probability $l < x < r$. Initially, $l = 0$ and $r = |S'| + 1$. In each search-step, we equidistantly select λ^{ε} chunks, and perform, using $\lambda^{1+\varepsilon}$ threads, all comparisons of elements in chunks with x. Lemma 2.5 guarantees that majority comparisons with chunks that actually are bigger or smaller come out correctly with high probability. In contrast, chunks that contain x can have an arbitrary majority, and we expect to see such chunks once the search has progressed close to the dislocation of S. Hence, the rule to update l is to set it to the highest numbered chunk such that all selected chunks left of it came out as "smaller" in majority. The rule for r is symmetric. If the number of selected chunks between l and r is not more than two, stop the search and return the average position in S between r and l. The number of such search-steps is at most $h = \lceil \frac{\log(n/3\lambda)}{\varepsilon \log(\lambda/2)} \rceil$. One search-step can be computed in $O(\log \lambda)$ parallel time or span, using $\lambda^{1+\varepsilon}$ processors on the PRAM, or work $\lambda^{1+\varepsilon}$ in the non-atomic binary fork-join model.

Lemma 2.6. *Let S be a sequence of n elements having maximum dislocation at most $d \geq \lambda \geq \log n$, where λ is a parameter, and let $x \notin S$. Under the persistent error model, with $p \leq 1/4e$, an index r_x with $r_x \in [\text{rank}(x,S) - 4d, \text{rank}(x,S) + 4d]$ can be found with probability at least $1 - 2^{-\lambda/2}(h\lambda^{\varepsilon})$. Let $h = \lceil \frac{\log(n/3\lambda)}{\varepsilon \log(\lambda/2)} \rceil$ and $\varepsilon > 0$ be any fixed constant. In the CREW-PRAM, the algorithm uses $\lambda^{1+\varepsilon}$ processors for $O((1/\varepsilon)\log(n/\lambda))$ time-units. In the binary-forking model with atomic operations, the algorithm has span $O((1/\varepsilon)\log(n/\lambda))$ and $O(h\lambda^{1+\varepsilon})$ work.*

Proof. The definition of h is the maximal number of search-steps in the preceding discussion. The overall span is the number of search-steps times the span per search-step, including forking and joining $\lambda^{1+\varepsilon}$ threads per search-step, i.e.,

$$\frac{\log(n/\lambda)}{\varepsilon \log \lambda} \cdot \log \lambda = \frac{\log(n/\lambda)}{\varepsilon}.$$

The success probability stems from a union bound overall comparisons of x with chunks as computed in Lemma 2.5. The method itself does not actually require atomic operations, so any requirements for atomic operations would only come from synchronization of this parallel search method with other operations. ◀

In the following we use the parameter N to denote the size of the original instance, such that we can both refer to the size n of the subproblem we are working on, and the notion of with high probability for the overall instance.

Theorem 2.7. *Let S be a sequence of n elements having maximum dislocation at most $d \geq (1+2c)\log n$ and let $x \notin S$. Let $N \geq \max(n, (\log n)^4)$ be a parameter. Under the persistent error model, with $p \leq 1/4e$, an index r_x with $r_x \in [\text{rank}(x,S) - 4d, \text{rank}(x,S) + 4d]$ can be found with probability at least $1 - N^{-c}$. In the CREW-PRAM,*

the algorithm uses $O((\log N)^2)$ processors for $O(\log n)$ steps. In the binary-forking model with atomic operations, the algorithm has span $O(\log n + \log\log N)$ and $O((\log n)(\log N)^2)$ work.

Proof. In Lemma 2.6, choose $\lambda = (1+2c)\log N$ and $\varepsilon = 1$. Observe $h = \lceil \frac{\log(n/3\lambda)}{\varepsilon \log(\lambda/2)} \rceil \le \log n$. Then $2^{-\lambda/2}(h\lambda^{\varepsilon}) \le N^{-c}\frac{(\log n)^2}{\sqrt{N}} \le N^{-c}$. ◀

2.4 Sorting with Small Radius

Recall that if L is a list of distinct elements and x is in L, then $\mathrm{rank}(x, L)$ is equal to x's index in a sorted ordering of L. The following lemma shows that sorting a list by good rank annotations for its elements can achieve a good bound on the list's maximum dislocation.

Lemma 2.8. *Let L be a list of n distinct elements, and suppose we have an annotation, R_x, for each element x in L, such that $|R_x - \mathrm{rank}(x, L)| < k$. Then sorting L by R_x values results in a list with maximum dislocation less than $2k$, i.e., $|\mathrm{rank}(L'[i], L) - i| < 2k$, for each $i = 1, 2, \ldots, n$, where L' is the list of elements from L sorted by R_x values.*

Proof. Consider the element x at rank $r = \mathrm{rank}(x, L)$ and let us determine what is the maximal number of elements that can have an annotation smaller than R_x. To maximize this number, we can assume that the annotation of x is maximal, i.e., $R_x = r + k - 1$. Precisely the elements with rank at most $r + 2k - 1$ can have an annotation less than $r + k$; hence, the highest rank of x in the annotation-sorted sequence is $r + 2k - 1$. By a symmetrical argument, the lowest rank of x in the annotation-sorted sequence is $r - 2k + 1$. ◀

The following lemma shows that we can quickly sort the elements of L by their R_x values in parallel, if L has maximum dislocation bounded by k.

Lemma 2.9. *Let L be a list of n distinct elements, such that $|\mathrm{rank}(L[i]) - i| < k$, and suppose we have an annotation, R_x, for each element x in L, such that $|R_x - \mathrm{rank}(x, L)| < k$. L can be (deterministically) sorted in parallel by the R_x values. In the CREW-PRAM, the algorithm uses n processors for $O(\log k)$ steps. In the binary-forking model with atomic operations, assuming there are $P = |L|/k$ threads available, the algorithm has $O(\log k)$ span and $O(n \log k)$ work.*

Proof. Divide L into contiguous chunks of size k each. By assumption, there is one thread per chunk, and the calculation of the start- and end-index of the chunk is deterministic.

For each chunk, C, sort the elements of C together with the three chunks to the left of C and the three chunks to the right of C, by R_x values (breaking ties by current position in L), using a deterministic parallel sorting algorithm, which runs in $O(\log k)$ time and $O(k \log k)$ work, as detailed in Corollary 2.3.[4] Consider the resulting list as split by the original 7 chunks, and use the fourth (middle) chunk as output for C.

With respect to the correctness of this algorithm, we show that this algorithm is equivalent to sorting the entire list L as a whole by R_x. Consider a chunk, $C = L[i : i + k - 1]$. The algorithm is equivalent to treating an element $y = L[j]$ with $j < i - 2k$ as having $R_y = -\infty$, and for $j > i + 3k - 1$ as $R_y = +\infty$, sorting the whole sequence with the result L' and outputing $C' = L'[i : i + k - 1]$. The algorithm is correct, if the rank in L' of an $x \in C'$ is the rank in the sorted sequence $\hat{L}$ where elements are sorted by R_x. For $x = \hat{L}[j]$ by Lemma 2.9 $|\mathrm{rank}(x, L) - j| < 2k$.

Now, starting from $\hat{L}$, we can perform all changes one by one. The algorithm is correct, if none of the changes affects the position of an element x in $\hat{L}[i : i + k - 1]$, i.e. $\mathrm{rank}(R_x, L) \in [i : i + k - 1]$. Let $y = L[j] = \hat{L}[h]$ be an element whose R_y value is changed to $-\infty$, i.e., $j < i - 3k$. Then by the assumed dislocation of L, $|\mathrm{rank}(y, L) - j| < k$, $\mathrm{rank}(y, L) < k + j$, and by the above argument $|h - \mathrm{rank}(y, L)| \le 2k$ i.e. $h \le 2k + \mathrm{rank}(y, L)$, and we can conclude $h \le 2k + k + j < 3k + i - 3k = i$, i.e., the position of y in $\hat{L}$ is to the left of $\hat{C}$. Similarly, if the value R_y is changed to $+\infty$, the position of y in $\hat{L}$ is to the right of $\hat{C}$. Hence, $L[i : i + k - 1] = \hat{L}[i : i + k - 1]$, and the algorithm is correct. ◀

[4] We pad the beginning of L with $2k$ $-\infty$ values, and we pad the end of L with $2k$ $+\infty$ values, so the required neighboring chunks for any chunk C always exist.

3 NOISY-BOX-SORT

In this section, we describe how to repeatedly reduce the maximum dislocation of a list via a series of parallel algorithms, so as to implement a type of noisy-box-sort, which nearly sorts a list of n elements in $O(\log n)$ time with quasi-linear work, in the presence of persistent errors. The whole section is dedicated to the proof of the following theorem, which is the main result of this paper:

Theorem 3.1 (Noisy-Box-Sort). *For an array with N elements, we can sort the array with comparison errors in the persistent error model with $p \le 1/32$, such that the maximal dislocation is $O(\log N)$, and the total dislocation is $O(N)$ with probability $1 - N^{-1.5}$. In the CREW-PRAM, the algorithm uses N processors for $O(\log N)$ steps. In the binary-forking model with atomic operations, the algorithm has span $O(\log N)$ and $O(N \log N)$ work.*

Because some of our algorithms operate on subsets of a larger list of elements, in the discussion that follows, as before, we use N to denote the size of the original input list and $n \le N$ to denote the size of a subset we might be operating on at any given point.

3.1 Structure of the overall algorithm and general considerations

In Section 3.2, we describe an approximate parallel sorting algorithm, $\mathcal{A}$, that has span $O(\log N)$ but uses $O(N \log^2 N)$ processors. To sort with a linear number of processors, we choose a random sample of $O(N/\log^2 N)$ elements and approximately sort them with $\mathcal{A}$ in span $O(\log N)$. Then for every element, x, not in the sample, we assign a single processor to x and perform a (serial) noisy binary search for x in the approximately sorted sample, as described in Section 3.3. This gives us an approximately sorted list of the input elements with polylogarithmic dislocation, in span $O(\log N)$ with N processors. Given this list, as explained in Section 3.4, we use a parallel version of the algorithm behind Theorem 2.4, together with a parallel version of RiffleSort [14, 15], to reduce the maximum dislocation to $O(\log n)$ with high probability. Finally, as detailed in Section 3.4.7, we finish with a parallel and local version

of WindowSort [14, 15], from which we inherit the dislocation guarantees of the final result.

Clearly, any result on sorting with errors needs an assumption on how big the error probability might be. Throughout our exposition, we assume that the error probabibilty p is known a priori (so to speak at compile time), and that $p < 1/32$. This assumption is inherited from [14, 15] because we use their WindowSort as the final step. As is argued there, the assumption can be relaxed to $p < 1/16$. To simplify the exposition, we only use milder assumptions on p if this does not incur any additional considerations and is hence more natural for the flow of the argument.

3.2 Achieving Polylogarithmic Dislocation with Superlinear Work

In this section, we describe a parallel approximate sorting algorithm, which runs in $O(\log N)$ time using $O(N \log^3 N)$ work and achieves a dislocation of $(c \log N)^4$. One iteration of the main loop of this algorithm is characterized by the following.

Lemma 3.2. *For a list, L, of n distinct elements with maximum dislocation k, we can sort L with persistent comparison errors, error probability $p < 1/2$, such that the maximal dislocation is at most $2\frac{\sqrt{k(1+c\ln N)}}{1-2p}$ with probability $1 - nN^{-c}$. In the CREW-PRAM, the algorithm uses kn processors for $O(\log k)$ steps. In the binary-forking model with atomic operations, the algorithm has span $O(\log k)$ and $O(kn \log k)$ work.*

Proof. We assign k processors to (fork k threads for) each element $x = x_i = L[i]$ and perform all comparisons of 'distance at most k': let $i_- = i - k$ and $i_+ = i + k$, only if this would cross the boundary of the array, keep the number $2k$ of elements different from i and place the interval at the (left or right) end of the array. Perform the erroneous comparisons with the elements $x_{i_-}, \ldots, x_{i_+}$ in $O(1)$ parallel time. Collect the count of how many elements are reported smaller than x, denoted by the value a_x. On the PRAM, this takes $O(\log k)$ parallel time using a balanced binary tree communication scheme. In binary fork-join, the span is $O(\log k)$ stemming from forking and joining. Compute $R_x = i_- + r_{2k}(a_x)$, as defined in Lemma 2.1, with the guarantee that, for r_x the real rank of x_i, $P[|R_x - r_x| \geq t] \leq 2\exp(\frac{-2(t(1-2p))^2}{2k})$. Solve $2\exp(\frac{-(t(1-2p))^2}{k}) = N^{-c}$ for t: $\frac{t^2(1-2p)^2}{k} = 1 + c\ln N$, $t = \frac{\sqrt{k(1+c\ln N)}}{1-2p}$. Sort all elements according to R_x in $O(\log k)$ parallel time as detailed in Lemma 2.9, using only one thread per k elements (deterministically selecting the first by index). By Lemma 2.8, the dislocation of the resulting array is $2t$. The high probability statement follows from a union bound over the error event over the n elements. ◀

An overhead in processors of the dislocation is prohibitive if used on the list directly. The following algorithm avoids this problem using a type of sampling technique. It is the main technical lemma of this paper.

Lemma 3.3 (The Polynomial Progress Lemma). *For an array (storing a list) with $n \geq 4$ elements and dislocation $k \geq \left(\frac{8c\ln N}{(1-2p)^2}\right)^4$, we can sort the array with persistent comparison errors, for error probability $p < 1/2$, such that the maximal dislocation of any element is $\leq k^{7/8}$ with probability $1 - N^{-c}$ for $N \geq \max(n, (\log n)^4)$. In the CREW-PRAM, the algorithm uses $n \log^2 N$ processors for $O(\log k)$ steps. In the binary-forking model with atomic operations, assuming there are already $n/\sqrt{k}$ threads, the algorithm has span $O(\log k + \log\log N)$ and work $O(n(\log k)(\log N)^2)$.*

Proof. Partition (statically by index) the input array into contiguous blocks of length $\sqrt{k}$ (using one thread per block), and independently sample one element in each block uniformly. The dislocation in the original array was k, in the sample it is hence $\sqrt{k} + 2 \leq \sqrt{2k}$. Using the algorithm of Lemma 3.2, approximately sort the sample in $O(\log k)$ parallel time, resulting in the list L. This requires $\sqrt{k}$ processors per sampled element, which is overall linearly many processors. The resulting new array of samples has, by Lemma 3.2, with probability N^{1-c}, a dislocation of at most $2\frac{\sqrt{\sqrt{2k}(1+c\ln N)}}{1-2p} \leq 2\frac{\sqrt{2c\ln N}}{1-2p}k^{1/4} =: d$.

For each non-sampled element, x, fork one thread in span $O(\log k)$. Let L_x be the subarray of L that contains all the sampled elements that where at distance at most k in the original array. Observe that $|L_x| \leq 4k$: by the dislocation k, these elements can have a true position that is $3k$ different from the position of x, and the new dislocation $d \leq k$. Perform a noisy parallel search for x in L_{i_x} as detailed in Theorem 2.7, leading to a position in L that is off by at most $4d$ positions in L. This takes $O(\log k)$ steps on $n(\log N)^2$ PRAM processors or span $O(\log k + \log\log N)$ and work $O((\log k)(\log N)^2)$ in binary forking. Label each element with the rank of this predecessor times $\sqrt{k}$ (and the original position as tiebreaker) to sort. This label has a radius polynomial in k and hence the algorithm of Lemma 2.9 takes $O(\log k)$ span (or PRAM steps), because there is one thread per element available.

More precisely, the dislocation of element x is given by a set D of elements that occupy the positions between x and its true rank. We argue that the probability that $|D| \geq k^{7/8}$ is small. Among the elements of D at most $4d$ can be sampled (by the guarantee of the noisy binary search). By the dislocation guarantee of the initial array, all elements of D must be within rank $3k$ of x in the original array. Hence, the relevant sampling process consists of at most $n' = 3\sqrt{k}$ independent random experiments, each selecting one of some subset of elements with a certain probability. Because each element is sampled with a probability of $1/\sqrt{k}$, the expected number of sampled elements in D is $k^{7/8}/\sqrt{k} = k^{3/8}$. We apply a Hoeffding bound (the lower bounding one) on these $\sqrt{k}$ many 0-1 random variables, with the bound $t = E[]/2$, such that the fail event of the Hoeffding bound is implied by the failing to sample sufficiently many elements, i.e. $k^{3/8}/2 > 4d = 4 \cdot 2\frac{\sqrt{2c\ln N}}{1-2p}k^{1/4}$ which rewrites as $k^{1/8} > 16\frac{\sqrt{2c\ln N}}{1-2p}$, or $k \geq \left(\frac{8c\ln N}{(1-2p)^2}\right)^4$, as assumed in the lemma. The failure probability is hence $\exp(-t^2/n') = \exp(-(k^{3/8}/2)^2/3\sqrt{k}) = \exp(-k^{1/8}/36) < c\log N$ for $k^{1/8}/36 > \ln(c\log N)$ which holds (with a large margin) by the previous lower bound on k. ◀

Repeating the above algorithm results in an array with polylogarithmic dislocation.

Theorem 3.4 (The Polylogarithmic Dislocation Theorem). *For an array with $n \geq 4$ elements, we can sort the array with persistent comparison errors, error probability $p \leq 1/32$, such that the maximal dislocation is $(c \ln N)^4$, with probability $1 - nN^{-c}$ for $N \geq n$. In the CREW-PRAM, the algorithm uses $O(n \log^2 N)$ processors for $O(\log n)$ steps. In the binary-forking model with atomic operations, the algorithm has span $O(\log n)$ and $O(n(\log n)(\log N)^2)$ work.*

Proof. The unsorted array has dislocation $n = k$. Run the algorithm of Lemma 3.3 $r = O(\log \log n)$ times with the sequence of dislocations being $k_i = n^{(7/8)^i}$ and $k_r \leq (c \ln N)^4$. Initially, fork $n/\sqrt{k_0}$ threads, as needed by the algorithm of Lemma 3.3. At the end of the final sorting step for k_i, join all the threads that belong to one chunk of size $\sqrt{k_{i+1}}$ into one, and use this thread as required by the next iteration of that algorithm. At the end of the last round of the algorithm, join all threads. This leads to a span of $O(\log n + \sum_i (\log \log N + \log k_i)) = O(\log n)$ because $(\log \log n)(\log \log N) = O(\log n)$ and because k_{r-1} conforms to the lower bound on k in the Lemma, by $p \leq 1/32$. ◀

We should at this point say something about synchronization, because joining completely at the end of a round and forking completely for the next round would exceed the claimed span. Instead of doing complete joins and forks, therefore, we do synchronization on a localized basis on $O(k)$-sized groups of chunks, and use atomic operations for the boundaries between groups to coordinate the synchronizations, so as to keep the span for each round to be $O(\log k + \log \log N)$.

3.3 Sampling and Approximately Sorting the Non-sampled Elements

To avoid the polylogarithmic overhead in the number of processors or threads, as would be required by a direct application of Theorem 3.4 on the entire array, we apply this theorem only on a random sample of the elements in the array. For the non-sampled elements perform a noisy binary search, as in Theorem 2.4, using one processor per element. This takes $O(\log N)$ parallel time using $O(N \log N)$ work and achieves a maximum dislocation of $O(\log^6 N)$ w.h.p.

Theorem 3.5 (Optimal Work with Polylogarithmic Dislocation). *For an array with $N \geq 4$ elements, we can sort the array with persistent comparison errors, error probability $p \leq 1/32$, such that the maximal dislocation is $4\alpha (\log N)^6$, with probability at least $1 - O(N^{-6})$. In the CREW-PRAM, the algorithm uses N processors for $O(\log N)$ steps. In the binary-forking model with atomic operations, the algorithm has span $O(\log N)$ and $O(N \log N)$ work.*

Proof. Let S be the sample resulting from choosing each element independently with probability $(1/\log N)^2$. Forking and joining threads takes $O(\log N)$ span. The expected size of the sample is $E[|S|] = N/(\log N)^2$, and with high probability, $|S| \leq 2N/(\log N)^2$, by the Chernoff bound

$$\Pr(|S| > 2E[|S|]) = \Pr(X - \mu \geq \delta\mu) < e^{-\frac{\delta^2 \mu}{3}} \leq e^{-\frac{N}{3(\log N)^2}} \ll N^{-6}.$$

Use $(\log N)^2$ processors per element to sort the sample up to a dislocation of $(\log N)^4$ using the algorithm of Theorem 3.4. Use one processor per non-sampled element to perform a noisy binary search as in Theorem 2.4. This takes $O(\log N)$ parallel time. Sort the non-sampled and sampled elements together by annotating the non-sampled elements with the index of the outcome of the noisy binary search and the sampled elements annotated with their index. We call this process to ***flatten*** the outcome of the binary search into the sampled elements. All of this can be done with span $O(\log N)$. We analyze the dislocation of an element x in this list. To this end, we consider the worst case that x ends in a position with the highest possible index. Then x is sorted into the highest numbered bucket allowed by Theorem 2.4, which is off by $\alpha d = \alpha(\log N)^4$. Additionally, all elements in the next $\alpha(\log N)^4$ higher true buckets could contribute to the dislocation of x by being misclassified by the binary search as being in their smallest allowed bucket. To bound the number of elements in these buckets, we consider the probability that among $4\alpha(\log N)^6 + b$ (consecutive) elements, less than $b = 2\alpha(\log N)^4$ are sampled. The expected number of sampled elements is $\mu \geq 4\alpha(\log N)^4$, so we look at a Chernoff bound with $\delta = 1/2$:

$$\Pr(X \leq (1-\delta)\mu) \leq e^{-\frac{\delta^2 \mu}{2}} = e^{-\frac{4\alpha(\log N)^4}{8}} \ll N^{-6}$$

Hence, whp, the dislocation of x is $4\alpha(\log N)^6$. ◀

3.4 From Polylogarithmic Dislocation to Logarithmic Maximal Dislocation

In this section, we describe how to go from an array with polylogarithmic maximum dislocation to one with optimal $O(\log N)$ dislocation. Achieving this efficiently in parallel turns out to be non-trivial, however, and we perform this dislocation reduction by dovetailing various sampling and merging steps, together with a parallel "failure-sweeping" technique [10, 16, 26], which we spell out in detail in Algorithm 1. Disregarding the filtering of misclassified elements, the structure of the algorithm resembles RiffleSort [13–15].

The discussion in this section assumes that the input array has $O(\log^d N)$ maximum dislocation.

3.4.1 Reduction to Polylogarithmic-Size Inputs for Algorithm 1. The phrasing of Algorithm 1 assumes that it operates on independent chunks of size $n = O(\log^{d+1} N)$. Similar to the sorting algorithm for small radius of Lemma 2.9, we partition the array that has dislocation $(\log N)^d$ into chunks of size $(\log N)^{d+1}$. To each chunk, we add $(\log N)^d$ neighboring elements from each of the neighboring (up to 2) chunks, so-called shadow elements. As we will argue for, Algorithm 1 sorts these chunks of $n \leq (\log N)^{d+1} + 2(\log N)^d$ elements such that the average dislocation is constant, and the probability that any element exceeds a dislocation of $c \log N$ is at most N^{-c}. Note that n is small compared to N and that we are willing to use $O(\log N)$ parallel time.

Observe that each element is in one or two chunks. This leads to one or two rank estimates from the sorting: If an element has rank r within the sorted chunk (including shadow elements), the rank in the overall sorted can be estimated as r plus the number of elements to the left of the chunk (including shadow elements). By the assumptions about the dislocation within the chunks, for all elements that are at least $c \log N$ from the chunk boundary, the dislocation guarantee carries over. For other elements, there

Algorithm 1: to reduce maximum dislocation to optimal from $O(\log^d N)$

Input : Array A of size $n = O(\log^{d+1} N)$
Output: Array A is sorted with optimal dislocation

1 **foreach** $x_i \in A$ ***in parallel*** **do** //place x_i in U_{g_i} and hence $S_{<g_i}$
2 Choose g_i, independently, from geometric distribution up to k, with $|S_k| \le c \log N$, and place x_i into U_{g_i}
3 Initialize $F = \emptyset$ //Consider S_k approximately sorted
4 **for** $j = k$ ***down to*** $j = 1$ **do** //merge U_j, S_j, reduce dislocation to $c \log N$
5 **foreach** $x_i \in U_j$ **do**
6 Perform $c\sqrt{\log N}$ steps of serial noisy binary search for x_i into S_j
7 //Error probability is $\ll n^{-c}$
8 Determine rank of x_i within $4c \log N$ neighbors $L_{i,j}$ in S_j using 2^j processors
9 **if** rank$(x_i, L_{i,j})$ ***is outside the*** $2c \log N$***-center*** **then** move x_i from U_j to F
10 Flatten U_j and S_j into S_{j-1} by outcome of binary search
11 **for** $h = 30c \log N$ ***down to*** $h = 1$***, step*** $h = h/2$ **do**
12 **foreach** $x_i \in S_{j-1}$ **do** determine rank of x_i within h neighbors, 2^j processors
13 Sort S_{j-1} by extrapolated rank
14 **foreach** $x_i \in F$ **do**
15 Perform $c \log N$ steps of serial noisy binary search for x_i into S_0
16 Flatten F and S_0 into S by outcome of binary search //Finish with Windowsort of [14, 15]
17 **for** $h = 30c \log N$ ***down to*** $h = 1$***, step*** $h = h/2$ **do**
18 **foreach** $x_i \in S$ **do** determine rank of x_i within h neighbors, use one processor
19 Sort S by extrapolated rank
20 **return** S

might be $c \log N$ elements that would be relevant for the rank, but not having been compared to, potentially adding $c \log N$ to the dislocation. If there is a single such estimate, like for most of the elements, we adjust this rank estimate to not be in the shadow. If there are two such estimates, we use the average of the two – as each of them has a guarantee, the average has as well. Now use the algorithm of Lemma 2.9 to sort according to this estimated rank, in parallel time $O(\log \log N)$, resulting in a maximal dislocation of $2c \log N$ with high probability.

3.4.2 Sampling: Line 2 of Algorithm 1. For every element x_i, we choose g_i from the geometric distribution with probability 1/2 (where $P(0) = 0, P(1) = 1/2, \ldots$) and collect the elements by these choices: Define $U_j = \{x_i \mid g_i = j\}$ (Unsorted) and $S_j = \{x_i \mid g_i > j\}$ (Sorted, as maintained by the algorithm) with the property $E[|S_j|] = E[|U_j|] = n/2^j$ for $j > 0$. Define $k = (d+1)\log\log N - \log(4c \ln N) = \Theta(\log \log N)$ and observe $E[|S_k|] = (\log N)^{d+1}/2^{(d+1)\log\log N - \log(4c \ln N)} = 4c \ln N$. We think of S_k not only as a set but also as a subsequence. Because the input array has dislocation $(\log N)^d$, if the sample was taking precisely every 2^k-th element, the dislocation in S_k would be constant (actually 0). In our algorithm, we will keep the invariant that the dislocation is at most $c \log N$ with high probability. For S_k, it is true deterministically, just because of its size.

3.4.3 Balanced Sizes of S_j and U_j, Mainly Line 8 of Algorithm 1. Further, the probability that any S_j or U_j is more than twice its expected size is bounded by a Chernoff bound (see appendix): For a chunk C with $n = |C|$, we have $\mu = E[|S_j|] = E[|U_j|] = n/2^j \ge 4c \ln N$. We use a Chernoff bound with $\delta = 1/2$ to show that it is up to a factor of 2 within this:

$$\Pr(|S_j| > 2E[|S_j|]) = \Pr(X - \mu \ge \delta\mu) < e^{-\frac{\delta^2\mu}{3}} \le e^{(-4c \ln N)/4} = N^{-c}.$$

Lemma 3.6. *With probability $1 - N^{-(c-1)}$ in all chunks and all j we have $|S_j| \le 2n/2^j$ and $|U_j| \le 2n/2^j$, where n is the size of the chunk.*

3.4.4 Round j of the Algorithm, Body of Line 4 of Algorithm 1. As an invariant, the array S_j is approximately sorted with dislocation $c \log N$. We present an algorithm that takes $O((\log N)2^{-j} + \log \log N)$ parallel time for round j. There are $p = n = |C|$ (elements in the chunk) processors available. By Lemma 3.6, both $|S_j| < 2n/2^j$ and $|U_j| < 2n/2^j$, such that in round j there are 2^{j-1} processors available per element. In sorting complexity $O(\log n) = O(\log \log N)$, we assign 2^{j-1} processors to each element.

Before we use several processors per element, we use a single processor to perform a "noisy binary search" as in Theorem 2.4 for $O(\log n)$ rounds. This takes $O(\log n) = O(\log \log N)$ time, and with success probability $1 - |U_j|^{-6}$, determines a rank up to $\alpha d + \log n \le 2\alpha c \log N$, where d is the current dislocation $c \log N$ and α is the constant of Theorem 2.4.

For each element in U_j, the 2^{j-1} processors compare to the $4\alpha c \log N$ neighboring (outcome of the noisy binary search) elements of S_j, and extrapolate a rank from the outcome as in Lemma 2.1. If the extrapolated rank is more than $2\alpha c \log N$ away, we consider the element "bad" and place it into the set F. This happens with probability at most U_j^{-6}, which means that the expected number of bad elements in round j is at most U_j^{-5} (in the following calculations, basically a 0). Observe that the event of being bad is independent for each element of U_j; hence, by a Chernoff bound (see appendix),

$$\mathrm{P}\left(|S_n - \mathrm{E}[S_n]| \ge t\right) \le 2\exp\left(-\frac{2t^2}{U_j}\right).$$

To have the leeway to apply a union bound, we choose the acceptable error as t_j such that $t^2/U_j = (c+1)\log N$, i.e., $t_j = \sqrt{U_j(c+1)\log N}$. Bad elements are "swept" into the set F, which we can afford if the number of bad elements is small. By a geometric sum, this leads to the following bound on the size of F:

Lemma 3.7. $|F| = O(\sqrt{nc \log N}) = o(n)$, *w.h.p.*

3.4.5 Flattening the Dislocation. Assume we have the almost sorted sequence S_j with dislocation $c \log N$ and the set of elements U_j,

where each element is assigned to a bucket/position in S_j that is correct up to an additive $2c \log N$, as described above.

In a first sorting step, we "flatten" the buckets, i.e., we assign the bucket number (and a random/arbitrary tie-breaker) to each element, and sort the whole sequence. This takes $O(\log n) = O(\log \log N)$ parallel time (observe: $O((\log n)^2)$ would be good enough as well, so we could even use one of the easier algorithms). Now, we want to bound the dislocation of this sorted sequence. We start by analyzing the "true buckets":

Lemma 3.8. *Let S'_i be the true sorted sequence of S_i, and let b_j be the number of elements from U_i that truly have rank j in S'_i, $b_j = |\{x \in U_i \mid j = |\{y \in S'_i \mid y < x\}|\}|$. Assume that U_i and S'_i are chosen independently for each element with probability 1/2. Then the probability that a particular set of $c \ln N$ consecutive buckets contains more than $8c \ln N$ elements is at most N^{-c}.*

Proof. Assume that the set of $c \ln N$ consecutive buckets contains more than $8c \ln N$ elements. Consider the $m \geq 8c \ln N$ elements in the buckets together with internal bucket separators from S_i. In the random experiment, at most $c \ln N$ elements must have been chosen to be in S_i, where the expected value was $1/2m$. Here, it is important that the choice of being in U_i or S_i is uniform (prob 1/2) and (perhaps more importantly) independent between different elements. We use a Chernoff bound (see appendix)

$$\Pr(X \geq (1-\delta)\mu) \leq e^{-\frac{\delta^2\mu}{2}},$$

with $\mu = 1/2m$ and $\delta = 3/4$ leading to $\frac{\delta^2\mu}{2} = \frac{9}{16}\frac{1}{2}\frac{8}{2}c \ln N = \frac{9}{8}c \ln N \geq c \ln N$, and hence $e^{-\frac{\delta^2\mu}{\delta}} \leq N^{-c}$. ◀

Thus, the assumptions of the following lemma are fulfilled with high probability for $D = c \log N$ for the right c.

Lemma 3.9. *Suppose the dislocation in S_i is at most D, the error in each noisy binary search is at most D, and that for any set of D consecutive true buckets, the number of elements in these buckets is at most $7D$. Then dislocation in the flattened sequence S_j is at most $30D$.*

Proof. Any element $x_k \in U_j$ is placed in a bucket at most $2D$ bucket-positions away from its true bucket: One D for the dislocation in S_j, and another, additive D, for the noisy binary search. This also implies, that all elements in a bucket have their true bucket at most $2D$ buckets away.

We bound the dislocation of an element x_i (true index) in the flattened sequence: If $x_i \in S_j$, the dislocation of S_j implies that its dislocation is at most D elements of S_j and the content of the $D-1$ U_j-buckets in between them. The number of true elements in these buckets is, by assumption, at most $8D$. Additionally, true elements from another $2D$ buckets (D each to the left and to the right) could have been included, giving an overall maximal dislocation of $8 \cdot 3D = 24D$.

If $x_i \in U_j$, it lands in a bucket at most $2D$ buckets from its true bucket. These can contain, together with their boundaries, at most $16D$ elements. Additionally, $14D$ elements can have switched into buckets on the wrong side (and into the bucket itself). Hence, the dislocation is at most $30D$. ◀

3.4.6 Final Merge of the Failed Elements, Lines 14–19 of Algorithm 1. By Lemma 3.7, the size of F is whp $o(n)$. Hence, the final merging by searching, flattening and dislocation reduction works in the same way (actually easier) than the previous rounds of the algorithm.

3.4.7 Final Parallel WindowSort. Finally, in Line 17 of Algorithm 1, we finish with a parallel and local version of WindowSort. This means that we use a geometrically (factor 1/2) decreasing sequence of window sizes, starting from the whp maximal dislocation $2c \log N$. Because the only randomness in WindowSort is in the comparisons with persistent errors, the output of our parallel version of WindowSort is the same as of the serial version, and by the original analysis in [14, 15], the total dislocation is $O(N)$ with high probability. As discussed in Section 3.1, this requires the error probability to be at most 1/16. For each window size, we (a) compute for each element its noisy rank in the window, and estimate a (global) rank from this, and (b) sort by this rank. Step (a) is implemented by a single processor per element in time proportional to the window size. Because the window size is geometrically decreasing, this totals to $O(\log N)$ parallel steps. Step (b) is done by sorting with small radius as detailed in Theorem 2.9. There are $O(\log \log N)$ rounds, and each takes $O(\log \log N)$ parallel time, so this is (easily) $O(\log N)$ parallel time (span). The requirement of having sufficiently many threads available is easily achieved.

The high probability bound for the maximal dislocation is $1 - O(N^{-4})$. In the failing case, we can observe that the total dislocation is at most N^2, leading to a contribution to the expected total dislocation of $O(1/N^2) = O(1)$.

4 AN OPTIMAL PARALLEL ALGORITHM FOR NON-PERSISTENT ERRORS

Theorem 4.1. *Given a list, L, of n distinct elements, we can sort L in $O(\log n)$ span and $O(n \log n)$ work, w.h.p., in the CREW PRAM or binary-forking model (with atomic operations).*

Our algorithms given above for the persistent-error model also work in the non-persistent error model. In this section, we show how to start with a list with logarithmic maximum dislocation and fully sort it optimally in parallel in the non-persistent model. In particular, we first apply the parallel algorithm of the previous section and then perform a "clean-up" algorithm we describe below to completely sort the input. This clean-up algorithm itself can experience failures in sorting subproblems, but we can apply a "failure-sweeping" technique [10, 16, 26], where we detect the failing subproblems, compact the elements in such subproblems, and then apply additional resources to sort the elements in these failing subproblems.

Suppose we have a list L of n distinct elements such that L has maximum dislocation at most $d = c \log n$, where $c \geq 4$ is a constant. Further, suppose that the result of each comparison of two elements x and y in L is false with probability $p \leq 1/4e$, independent of all other comparisons, even for previous comparisons of x and y.

Our method for completely sorting L in this model in parallel is as follows.

(1) Divide L into chunks of consecutive elements of size d. For each chunk, C, in parallel, sort C together with its immediately smaller and larger chunks (i.e., we sort each

such set of $3d$ elements in parallel), and output the middle set of d elements. Note that if all these sorts are done correctly, then the entire list will be sorted. Our method for doing each of these sorts is to use a modified version of the well-known parallel odd-even mergesort algorithm [4], which has $O(\log^2 d)$ span and $O(d \log^2 d)$ work, in either the CREW PRAM or binary-forking model. The modification is that rather than perform each comparison in the algorithm just once, we assign $\lceil d/\log^2 d \rceil$ processors to the comparison and perform the comparison independently for each such processor, taking the answer to be the majority of the results (each majority can be determined in $O(\log d)$ span and $O(d/\log^2 d)$ work). Thus, the sort we perform for each chunk has $O(\log^3 d)$ span and $O(d^2)$ work. That is, the total span needed for this step is $O(\log^3 d) = O((\log\log n)^3)$ and the total work is $O((n/d)d^2) = O(nd) = O(n \log n)$.

(2) For each chunk, C, let L_C denote the sublist of size $3d$ that was output from the previous step. For each L_C, in parallel, perform $O(d)$ comparisons between each consecutive pair of elements to confirm that they are in the correct order. If the majority of such comparisons for any consecutive pair of elements L_C reports these elements as out-of-order, then we mark C as a "failure." This step requires $O(d) = O(\log n)$ span and $O(d^2)$ work per chunk, so the total work is $O((n/d)d^2) = O(nd) = O(n \log n)$.

(3) Compact the elements in each L_C such that C is marked as a "failure" into an array of size $3\lceil n/d \rceil$, i.e., so we can accommodate up to $\lceil n/d^2 \rceil$ chunks marked as "failures." (We show that with high probability this is always possible to do.) The span for this step is $O(\log n)$ and the work is $O(n)$.

(4) For each failure chunk, C, we repeat the sorting algorithm of Step 1, except this time we apply d processors for each comparison, taking the result as the majority of the d outcomes. Thus, the sort we perform for each "failure" chunk has $O(\log^3 d)$ span and $O(d^2 \log^2 n)$ work. That is, the total span needed for this step is $O(\log^3 d) = O((\log\log n)^3)$ and the total work is $O((n/d^2)d^2 \log^2 d) = O(n \log^2 d) = O(n(\log\log n)^2)$.

Let us conservatively bound the probability that the sort we perform for a chunk is a failure by the probability that any one of its $O(d \log^2 d)$ comparisons is wrong, i.e., that a majority of the $\lceil d/\log^2 d \rceil$ processors assigned to this comparison got it wrong. By a Chernoff bound (see appendix), such a comparison is wrong with probability at most $2^{-d/(2\log^2 d)}$, since $mu = \lceil d/\log^2 d \rceil p$ and $p \leq 1/4e$. Thus, the probability that a chunk is a "failure" is $p_f = ad^2(\log^2 d)2^{-d/(2\log^2 d)}$, by a union bound, where a is a constant.

Note that the check to see if a chunk is a failure simply tests if it is sorted or not, based on performing $3d$ tests involving d independent comparisons. The probability that a majority of such comparisons wrongly reports that two out-of-order elements are in the correct order can be bound by a Chernoff bound. Let X denote the number of in-order comparison results for a pair of out-of-order elements. Then $\mu = dp$, and

$$\Pr\left(X \geq \frac{d}{2}\right) = \Pr\left(X \geq \frac{\mu}{2p}\right) \leq 2^{-d/2} \leq n^{-2}.$$

Thus, with high probability, all the chunk tests for failure are correct. Finally, note that failures of chunks are mutually independent. Thus, we can use another application of a Chernoff bound to bound the probability that more than $\lceil n/d^2 \rceil$ chunks are "failures." In this case, let X denote the number of failure chunks and note that $\mu = \lceil n/d \rceil p_f$. Thus,

$$\Pr\left(X \geq \frac{n}{d^2}\right) = \Pr\left(X \geq \frac{\mu}{p_f d}\right) \leq 2^{-n/d^2},$$

since $p_f d = ad^3(\log^2 d)2^{-d/2\log^2 d} \leq 1/2e$. Thus, with high probability, the algorithm is correct.

REFERENCES

[1] Zafar Ahmad, Rezaul Chowdhury, Rathish Das, Pramod Ganapathi, Aaron Gregory, and Mohammad Mahdi Javanmard. Low-span parallel algorithms for the binary-forking model. In *33rd ACM Symp. on Parallelism in Algorithms and Architectures (SPAA)*, pages 22–34, 2021. doi:10.1145/3409964.3461802.

[2] Miklós Ajtai, János Komlós, and Endre Szemerédi. An $O(n \log n)$ sorting network. In *15th ACM Symp. on Theory of Computing (STOC)*, pages 1–9, 1983.

[3] Miklós Ajtai, János Komlós, and Endre Szemerédi. Sorting in $c \log n$ parallel steps. *Combinatorica*, 3(1):1–19, 1983.

[4] K. E. Batcher. Sorting networks and their applications. In *AFIPS Spring Joint Computer Conference (SJCC)*, pages 307–314. ACM, 1968. doi:10.1145/1468075.1468121.

[5] Guy E. Blelloch, Jeremy T. Fineman, Yan Gu, and Yihan Sun. Optimal parallel algorithms in the binary-forking model. In *32nd ACM Symp. on Parallelism in Algorithms and Architectures (SPAA)*, pages 89–102, 2020.

[6] Mark Braverman and Elchanan Mossel. Noisy sorting without resampling. In *19th ACM-SIAM Symposium on Discrete Algorithms (SODA)*, page 268–276, 2008.

[7] Richard Cole. Parallel merge sort. *SIAM Journal on Computing*, 17(4):770–785, 1988.

[8] Richard Cole and Vijaya Ramachandran. Resource oblivious sorting on multicores. *ACM Trans. Parallel Comput.*, 3(4), mar 2017. doi:10.1145/3040221.

[9] Michael Dillencourt and Michael T. Goodrich. Simplified Chernoff bounds with powers-of-two probabilities. *Information Processing Letters*, page 106397, 2023. doi:https://doi.org/10.1016/j.ipl.2023.106397.

[10] Uriel Feige, Prabhakar Raghavan, David Peleg, and Eli Upfal. Computing with noisy information. *SIAM Journal on Computing*, 23(5):1001–1018, 1994.

[11] Marc Fischlin. A cost-effective pay-per-multiplication comparison method for millionaires. In David Naccache, editor, *Topics in Cryptology CT-RSA*, pages 457–471. Springer, 2001.

[12] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W.H. Freeman and Co., 1979.

[13] Barbara Geissmann, Stefano Leucci, Chih-Hung Liu, and Paolo Penna. Sorting with recurrent comparison errors. In Yoshio Okamoto and Takeshi Tokuyama, editors, *28th Int. Symp. on Algorithms and Computation (ISAAC)*, volume 92 of *LIPIcs*, pages 38:1–38:12, 2017. doi:10.4230/LIPIcs.ISAAC.2017.38.

[14] Barbara Geissmann, Stefano Leucci, Chih-Hung Liu, and Paolo Penna. Optimal sorting with persistent comparison errors. In Michael A. Bender, Ola Svensson, and Grzegorz Herman, editors, *27th European Symposium on Algorithms (ESA)*, volume 144 of *LIPIcs*, pages 49:1–49:14, 2019. URL: https://arxiv.org/abs/1804.07575.

[15] Barbara Geissmann, Stefano Leucci, Chih-Hung Liu, and Paolo Penna. Optimal dislocation with persistent errors in subquadratic time. *Theory of Computing Systems*, 64(3):508–521, 2020. This work appeared in preliminary form in STACS'18.

[16] Mujtaba R Ghouse and Michael T Goodrich. In-place techniques for parallel convex hull algorithms. In *3rd ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 192–203, 1991.

[17] Michael T. Goodrich, Riko Jacob, and Nodari Sitchinava. Atomic power in forks: A super-logarithmic lower bound for implementing butterfly networks in the nonatomic binary fork-join model. In *ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 2141–2153, 2021. doi:10.1137/1.9781611976465.128.

[18] Joseph JáJá. *An Introduction to Parallel Algorithms*. Addison-Wesley, Reading, MA, 1992.

[19] Richard M Karp and Robert Kleinberg. Noisy binary search and its applications. In *18th ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 881–890, 2007.

[20] Richard M. Karp and Vijaya Ramachandran. Parallel algorithms for shared-memory machines. In *Handbook of Theoretical Computer Science (Vol. A): Algorithms and Complexity*, pages 869–941. MIT Press, Cambridge, 1991.

[21] Kamil Khadiev, Artem Ilikaev, and Jevgenijs Vihrovs. Quantum algorithms for some strings problems based on quantum string comparator. *Mathematics*, 10(3):377, 2022.
[22] Rolf Klein, Rainer Penninger, Christian Sohler, and David P. Woodruff. Tolerant algorithms. In Camil Demetrescu and Magnús M. Halldórsson, editors, *European Symposium on Algorithms (ESA)*, pages 736–747. Springer, 2011.
[23] Tom Leighton, Yuan Ma, and C. Greg Plaxton. Breaking the $\Theta(n \log^2 n)$ barrier for sorting with faults. *Journal of Computer and System Sciences*, 54(2):265–304, 1997. doi:10.1006/jcss.1997.1470.
[24] Wen Liu, Shou-Shan Luo, and Ping Chen. A study of secure multi-party ranking problem. In *Eighth ACIS Int. Conf. on Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing (SNPD)*, volume 2, pages 727–732, 2007. doi:10.1109/SNPD.2007.367.
[25] Cheng Mao, Jonathan Weed, and Philippe Rigollet. Minimax rates and efficient algorithms for noisy sorting. In Firdaus Janoos, Mehryar Mohri, and Karthik Sridharan, editors, *Proceedings of Algorithmic Learning Theory*, volume 83 of *Proceedings of Machine Learning Research*, pages 821–847, 2018.
[26] Yossi Matias and Uzi Vishkin. Converting high probability into nearly-constant time—with applications to parallel hashing. In *23rd ACM Symposium on Theory of Computing (STOC)*, pages 307–316, 1991.
[27] Michael Mitzenmacher and Eli Upfal. *Probability and Computing: Randomization and Probabilistic Techniques in Algorithms and Data Analysis*. Cambridge University Press, second edition, 2017.
[28] Rajeev Motwani and Prabhakar Raghavan. *Randomized Algorithms*. Cambridge University Press, 1995.
[29] Andrzej Pelc. Searching with known error probability. *Theoretical Computer Science*, 63(2):185–202, 1989. doi:10.1016/0304-3975(89)90077-7.
[30] Andrzej Pelc. Searching games with errors—fifty years of coping with liars. *Theoretical Computer Science*, 270(1):71–109, 2002. doi:doi.org/10.1016/S0304-3975(01)00303-6.
[31] Nicholas Pippenger. On networks of noisy gates. In *26th IEEE Symp. on Foundations of Computer Science (FOCS)*, pages 30–38, 1985. doi:10.1109/SFCS.1985.41.
[32] Rüdiger Reischuk. Probabilistic parallel algorithms for sorting and selection. *SIAM Journal on Computing*, 14(2):396–409, 1985. doi:10.1137/0214030.
[33] Alfréd Rényi. On a problem in information theory. *Magyar Tud. Akad. Mat. Kutató Int. Közl.*, 6:505–516, 1961. See mathscinet.ams.org/mathscinet-getitem?mr=0143666.
[34] Jeffrey Scott Vitter and David Alexander Hutchinson. Distribution sort with randomized cycling. *J. ACM*, 53(4):656–680, 2006. doi:10.1145/1162349.1162352.
[35] Ziao Wang, Nadim Ghaddar, and Lele Wang. Noisy sorting capacity. *arXiv*, abs/2202.01446, 2022. arXiv:2202.01446.
[36] Cailiang Xu, Wei Wang, Deng Zhou, and Tao Xie. An SSD-HDD integrated storage architecture for write-once-read-once applications on clusters. In *IEEE International Conference on Cluster Computing*, pages 74–77, 2015. doi:10.1109/CLUSTER.2015.20.
[37] Ya Xu, Nanyu Chen, Addrian Fernandez, Omar Sinno, and Anmol Bhasin. From infrastructure to culture: A/B testing challenges in large scale social networks. In *21th ACM SIGKDD Int. Conf. on Knowledge Discovery and Data Mining (KDD)*, pages 2227–2236, 2015. doi:10.1145/2783258.2788602.
[38] Andrew C. Yao. Protocols for secure computations. In *23rd IEEE Symp. on Foundations of Computer Science (FOCS)*, pages 160–164, 1982. doi:10.1109/SFCS.1982.38.
[39] Andrew C. Yao and F. Frances Yao. On fault-tolerant networks for sorting. *SIAM Journal on Computing*, 14(1):120–128, 1985. doi:10.1137/0214009.

A TAIL BOUNDS

Let $X_1, X_2, \ldots, X_n$ be independent random variables such that $a_i \leq X_i \leq b_i$. Consider the sum of these random variables, $S_n = X_1 + \cdots + X_n$. Then Hoeffding's theorem states that, for all $t > 0$:

$$\mathrm{P}\left(|S_n - \mathrm{E}[S_n]| \geq t\right) \leq 2\exp\left(-\frac{2t^2}{\sum_{i=1}^{n}(b_i - a_i)^2}\right)$$

where the $b_i - a_i = 1$, i.e. the sum is n.

Suppose $X_1, X_2, \ldots, X_n$ are independent random variables taking with in $\{0, 1\}$. Let X be their sum and let $\mu = E[X]$. Then for any $\delta > 0$, the following Chernoff bound holds:

$$\Pr(X > (1+\delta)\mu) < \left(\frac{e^{\delta}}{(1+\delta)^{1+\delta}}\right)^{\mu}.$$

There are also simplified Chernoff bounds, e.g., see [9, 27]:

$$\Pr(X \leq (1-\delta)\mu) \leq e^{-\frac{\delta^2\mu}{2}}, \qquad 0 \leq \delta \leq 1,$$

$$\Pr(X \geq (1+\delta)\mu) \leq e^{-\frac{\delta^2\mu}{2+\delta}}, \qquad 0 \leq \delta,$$

$$\Pr(|X-\mu| \geq \delta\mu) \leq 2e^{-\frac{\delta^2\mu}{3}}, \qquad 0 \leq \delta \leq 1.$$

$$\Pr(X \geq R) \leq 2^{-R}, \qquad R \geq 2e\mu.$$

B AN EREW SIMULATION RESULT

It is often possible to describe a parallel algorithm so that each of its memory cells is written to and subsequently read at most once [36]. For example, one can often index memory cells by "rounds" or super-steps in a parallel algorithm, and reference memory cells by address-index pairs, copying values from previous rounds as needed. For example, Cole's parallel mergesort algorithm [7] can be described this way.

Theorem B.1 (Same as Theorem 2.2). *If $\mathcal{A}$ is an EREW PRAM algorithm where each memory cell is written to and subsequently read at most once, such that $\mathcal{A}$ runs in T time using P processors and W work, then one can simulate $\mathcal{A}$ in the binary-forking model (with atomic operations) with $O(T + \log P)$ span and $O(W)$ work.*

Proof. We assume that initially each memory cell, M_j, used by $\mathcal{A}$, contains an input data value or a null-data value, $\perp$. We begin by forking P threads, in $O(\log P)$ span, to associate a thread, T_p, with each processor, and we simulate step $i > 0$ in parallel for each processor thread T_p as follows:

- If p performs no read or write in step i, then T_p simply performs the internal computation for p in step i of $\mathcal{A}$, and T_p continues to step $i + 1$.
- If p performs a read of a memory cell, M_j, in step i, then T_p atomically checks if $M_j = \perp$, and if $M_j \neq \perp$, then T_p reads M_j and continues to step $i+1$; otherwise, T_p writes R_p to M_j, where R_p is the state of p's registers (including p's identity and p's program counter) and T_p dies.
- If p performs a write to a memory cell, M_j, in step i, then T_p atomically checks if $M_j = \perp$, and if $M_j = \perp$, then T_p writes its indended value to M_j; otherwise, T_p reads $R_{p'}$ from M_j and then T_p writes its indended value to M_j. In the latter case, before T_p continues to step $i + 1$ it forks the thread $T_{p'}$, initializing its register set to $R_{p'}$.

The span for this simulation is $O(T + \log P)$, given the dependencies between memory cells and processor threads, since we spend $O(1)$ span per interaction between a memory cell and processor per step of $\mathcal{A}$, after the initial forking of P processor threads, which requires $O(\log P)$ span. In addition, the work is $O(W)$, since we perform $O(1)$ work per step performed by a processor in $\mathcal{A}$. ◀

By the way, atomic operations are essential for achieving this result as we do, since it is impossible to simulate Cole's parallel mergesort algorithm with less than $\Omega(\log n \log\log n)$ span in the nonatomic binary-forking model, due to a lower-bound of Goodrich, Jacob, and Sitchinava [17] from SODA'21.