# Parallel Integer and Learning-Augmented Sorting Algorithms in the Binary-Forking Model

Michael T. Goodrich
*Dept. of Computer Science*
*Univ. of California, Irvine*
Irvine, CA USA
goodrich@uci.edu

Yan Gu
*Dept. of Computer Science*
*Univ. of California, Riverside*
Riverside, CA USA
ygu@cs.ucr.edu

***Abstract*—We provide parallel algorithms for integer sorting and predictive sorting in the binary-forking model, which is designed to reflect the asynchronous and dynamic nature of modern multi-threaded architectures and languages, where synchronization of threads is accounted for while also allowing for asynchronous memory writes using atomic operations, such as test-and-set. We show how to sort $n$ integers in the range from $0$ to $n$ in $O(\log n)$ span and $O(n)$ work with overwhelming probability, which is asymptotically optimal and appears to be the first such bounds for the binary-forking model. More importantly, we also provide a parallel learning-augmented sorting algorithm, where a machine-learning model provides a prediction of the rank of each element in the sorted sequence. Given such predictions for $n$ elements in an array, $X$, we show how to sort $X$ in $O(\log n)$ span with work that is proportional to $n$ times the logarithm of the average prediction alignment error of $X$, with overwhelming probability.**

***Index Terms*—parallel algorithms, binary-forking model, sorting, learning-augmented algorithms, machine learning.**

## I. Introduction

The design and analysis of algorithms has long been dominated by the pursuit of worst-case performance guarantees; see, e.g., [7], [21]. While such guarantees are essential for ensuring reliable and predictable performance across a broad range of inputs, they do not always reflect the complexities or subtleties of real-world problems. Worst-case analysis often leads to overly conservative and inefficient solutions, for example, especially when the inputs exhibit structure or when certain inputs are more likely than others. To address these limitations, the field of ***beyond worst-case*** algorithm design has emerged as a promising direction, aiming to improve algorithmic performance in practice by exploiting realistic assumptions about the input distributions or problem characteristics; see, e.g., [44], [45]. A particularly promising type of beyond worst-case algorithm design is ***learning-augmented algorithms***, which considers algorthmic improvements that can take advantage of predictions from machine-learning models; see, e.g., [8], [11], [13], [25]–[27], [32], [33], [46]–[48], [51]. (See also the website [52] for numerous more theory papers on the approach.) This field offers ways to achieve practical improvements in real-world scenarios where data patterns can be learned and utilized.

Learning-augmented algorithms is a subfield of algorithm design that aims to improve algorithm performance by incorporating machine-learned predictions about the problem instance. As such, it is also a subfield of artificial intelligence, since it incorporates machine-learning quality metrics into algorithm runtimes, space complexities, and/or the quality of solutions in scenarios where traditional worst-case analysis may be overly pessimistic. Such algorithms are designed to utilize predictions generated by machine-learning models to guide their operations. For example, a prediction about the location of an element in a sorted array could inform a search algorithm, potentially improving upon a standard binary search. Indeed, Mitzenmacher and Vassilvitskii [42] discuss the field of predictive algorithmics using predictive search in a sorted array as a motivating example.

The performance of algorithms with predictions is typically tied to the quality of the provided predictions. When predictions are accurate, the algorithm can achieve significantly better performance. Even with imperfect predictions, the algorithms are designed to maintain reasonable performance guarantees, ideally falling back to a worst-case performance that is comparable to traditional algorithms. Research in this area also focuses on ensuring the robustness of algorithms to potentially inaccurate predictions and exploring techniques like calibration to effectively integrate uncertainty estimates from machine-learning models.

There has been considerable research on sequential learning-augmented algorithms, which shows that sequential learning-augmented algorithms can achieve improvements over state-of-the-art algorithms designed with worst-case performance in mind using classical design techniques; see, e.g., [8], [11], [13], [25]–[27], [32], [33], [46]–[48], [51]. Nevertheless, in spite of this attention, there has been much less work on parallel learning-augmented algorithms. Thus, this paper seeks to advance the theory of parallel learning-augmented algorithms, with a focus on developing sorting algorithms that perform optimally in terms of their span and work, as a function of the input size, $n$, as well as metrics derived from machine-learning models. We believe the approaches advanced in this paper provide a more nuanced view of algorithmic efficiency, allowing for the design of algorithms that outperform worst-case bounds in practice, while still maintaining rigorous mathematical guarantees based on metrics derived from machine-learning models.

Importantly, we are interested in parallel algorithms for the binary-forking model, which is designed to reflect the asynchronous and dynamic nature of modern multi-threaded architectures and languages, where synchronization of threads is accounted for while also allowing for asynchronous memory writes using atomic operations, such as test-and-set.

Suppose, then, that we are given an input array, $X = [x_1, x_2, \ldots, x_n]$ of distinct comparable elements, such that, for each $i = 1, 2, \ldots, n$, we are given a prediction, $p(x_i)$, derived from a machine-learning model such that $p(x_i)$ is estimating the rank of $x_i$ in $X$, that is, the number of elements that are smaller than $x_i$. At a high level, our approach for sorting $X$ is simple—we first perform an integer sorting of $X$ using the $p(x_i)$ predictions as the keys, giving a semi-sorted array, $X'$, and then we sort $X'$ taking the sortedness of $X'$ into consideration. See Figure 1. As simple as this approach sounds, there are number of challenges for implementing it in the binary-forking parallel model so as to have $O(\log n)$ span and efficient work performance in terms of a reasonable metric of the prediction error. For example, we desire a ***robust*** work bound, that is, a work performance that is sensitive to the prediction error while never being asymptotically inferior to the best known worst-case bound that does not take knowledge about the input into consideration.

### A. Related Prior Work

Prior to our work, we are not aware of any integer sorting algorithm in the binary-forking model (formally defined in section II) that achieved optimal $O(\log n)$ span and $O(n)$ work. The best previous method in this model is an algorithm by Blelloch, Fineman, Gu, and Sun [5] for the semisorting problem, which reorders an input array of $n$ keys such that equal keys are contiguous but different keys are not necessarily in sorted order [23]. Blelloch *et al.* show how to solve the semisorting problem in $O(\log n)$ span and $O(n)$ work in expectation in the binary-forking model. There are also existing general sorting algorithms in the binary-forking model that use $O(\log n)$ span and $O(n \log n)$ work [5], [19]. Dong, Dhulipala, Gu, and Sun [14] study the integer sorting problem from a practical point of view in the binary-forking model, giving a number of interesting experimental results as well as a theoretical method with $O(\log^2 n)$ span and linear work with high probability.

There is much more work on parallel algorithms for integer sorting in PRAM models, which assumes a much stronger unit-cost global synchronization. Translating the span bounds in PRAM to the binary-forking model introduces an additional factor of $O(\log n)$, thus non-optimal. For example, Goodrich, Matias, and Vishkin [20] show how to sort a set of $n$ integers into an array of size $O(n)$ in $O(\log^* n)$ time and linear work with overwhelming probability in the CRCW PRAM model. Also, Rajasekaran and Reif [43] show how to sort $n$ integers in a PRAM model in $O(\log n)$ time using $O(n)$ work with overwhelming probability. Also, Han and Shen [24] show how to solve the integer sorting problem deterministically in a PRAM model in $O(\log n)$ time and $O(n\sqrt{\log n})$ work.

The previous sequential algorithms for learning-augmented sorting, which are due to Bai and Coester [3], have running times that are proportional to $n$ times prediction error metrics, but their methods all appear to be inherently sequential; hence, trying to convert these algorithms into efficient parallel methods does not appear to be a productive approach. Our approach, instead, is to adapt a parallel block-sorting algorithm of Levcopoulos and Petersson [35], which is for a PRAM model and runs in $O(((n \log(\mathrm{Inv}(X)/n))/p) + \log n)$ time using $p$ processors, where $\mathrm{Inv}(X)$ is the number of inversions in the input array. Interestingly, adapting their approach to the binary-forking model involved our making a number of non-trivial modifications to parallel block sorting, as well as using new Chernoff bounds [9], [10] in our analysis.

### B. Our Results

In this paper, we show how to sort an array, $A$, of $n$ integers in the range from 0 to $O(n)$ with $O(\log n)$ span and $O(n)$ work with overwhelming probability (w.o.p.). We provide the formal definitions of the probabilistic guarantees in section II.

More importantly, we also provide what we believe is the first parallel learning-augmented sorting algorithm. In the domain of learning-augmented algorithms, there are a number of different metrics that one can use to measure the error between the predictions that come from a machine-learning model and the ground truth. In this paper, we focus on a metric that we refer to as the ***prediction alignment error***, $\mathrm{PAE}(X)$. For each $i = 1, 2, \ldots, n$, given a prediction, $p(x_i)$, for each element $x_i \in X$, we define the ***prediction alignment error for*** $i$ to be

$$\mathrm{PAE}_i(X) = |\{j \mid p(x_j) \leq p(x_i) \text{ and } x_j > x_i\}|.$$

That is, $\mathrm{PAE}_i(X)$ is the the number of elements, $x_j$, in $X$ whose predicted location is not greater than the predicted location for $x_i$ but $x_j$ is larger than $x_i$.[1] In other words, the elements that are counted in $\mathrm{PAE}_i(X)$ should come after $x_i$ in a sorting of $X$, but their predicted locations either match the predicted location for $x_i$ or come before the predicted location for $x_i$. Given predicted ranks for the elements of $X$, we define the ***prediction alignment error of*** $X$, $\mathrm{PAE}(X)$, as

$$\mathrm{PAE}(X) = \sum_{i=1}^{n} \mathrm{PAE}_i(X).$$

In this paper, we show how to sort an input array, $X$, in $O(\log n)$ span and $O(n(1+\log((\mathrm{PAE}(X)/n)+1)))$ work with overwhelming probability. Our algorithm involves a number of non-trivial steps, including adaptations and extensions of work by Mannila [40] and Levcopoulos and Petersson [35], and our analysis is also non-trivial, including applications of new Chernoff bounds by Dillencourt *et al.* [9], [10].

[1] Bai and Coester [3] introduce what we are calling the "prediction alignment error for $i$," which they denote by $\eta_i^l$ and use to analyze some of their sequential learning-augmented sorting algorithms.

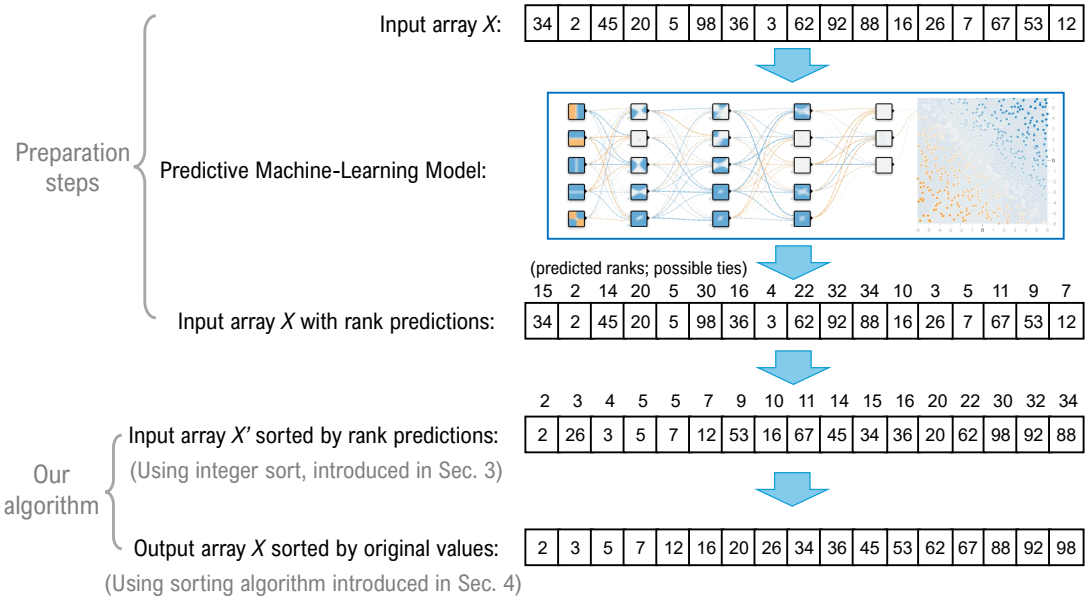


Fig. 1. A schematic for our parallel learning-augmented sorting algorithm. Neural network image was generated using TensorFlow playground [1].

## II. Preliminaries

### A. Notations for Probabilistic Guarantees

In this paper, we discuss randomized algorithms and aim to demonstrate that our algorithm achieves a very high "success rate" despite randomness.

We adopt the standard definition of high probability. Specifically, we say that a probability, $p$, is ***high probability***, if $p \geq 1 - O(1/n^c)$ for some constant $c \geq 1$.

In addition, we use an even stronger version, referred to as overwhelming probability [28]. We say that a probability $p$ is ***overwhelming probability***, if $p > 1 - O(1/n^c)$ for ***any*** constant $c \geq N_0$ given a constant integer, $N_0$. Note that the overwhelming probability is crucial for sorting algorithms, since they are widely used as a subroutine of more complicated algorithms. The overwhelming probability guarantees a very high success rate by calling the sorting subroutine a polynomial number of times, by directly applying the union bound. To the best of our knowledge, we are unaware of work-efficient (see the definition below) and highly parallel integer sort and predictive sort with overwhelming probability (w.o.p.).

### B. Chernoff Bounds

We use Chernoff bounds extensively in this paper. Most of the forms we used are from Dillencourt *et al.* [9], [10].

**Theorem 1** (Chernoff Bounds)**.** *Let $X_1, X_2, \ldots, X_n$ be independent random variables taking values in $\{0, 1\}$. Let $X = \sum_{i=1}^{n} X_i$ and let $\mu = \mathbb{E}[X]$ denote $X$'s expected value.*

$$\Pr\big(X < (1-\delta)\mu\big) < 2^{-\mu/2}, \qquad \text{for } 0.7064 \leq \delta < 1. \quad (1)$$

$$\Pr\big(X > (1+\delta)\mu\big) < 2^{-\mu/2}, \qquad \text{for } \delta \geq 1. \quad (2)$$

$$\Pr\big(X < (1-\delta)\mu\big) < 2^{-\mu/5}, \qquad \text{for } 1/2 \leq \delta < 1. \quad (3)$$

### C. Computational Model

We use the *work-span model* in the classic multithreaded model with *binary-forking* [2], [5], [6]. We assume a set of threads that share a common memory. Each thread acts like a sequential RAM plus a fork instruction that forks two threads to then be running in parallel. When both threads finish, the original thread continues. A parallel-for is implemented by forks in a logarithmic number of steps in a binary-tree fashion. We assume unit-cost atomic operation TEST-AND-SET($p$), which atomically changes $p$ from false to true. It returns true if successful and false otherwise, and it can be used to implement a "join" operation where a forking thread must wait until its most recently forked child thread ends. A computation can be viewed as a directed acyclic graph (DAG) whose nodes are operations and edges are logical dependencies. Importantly, unlike parallel RAM (PRAM) models, the binary-forking model does not require that executing threads are synchronized at the fine-grain level—instead all synchronization occurs via threads finishing and/or using join operations. The *work* $W$ of a parallel algorithm is the total number of operations, and the *span (depth)* $S$ is the longest path in the DAG. That is, the span of a parallel algorithm corresponds to parallel time without regarding the number of processors and the work of a parallel algorithm corresponds to its running time if we had to simulate it on a single processor. In practice, we can execute a parallel computation in the binary-forking model with work $W$ and span $S$ using a randomized work-stealing scheduler [6], [22] in time $W/P + O(S)$ with $P$ processors with high probability. A parallel algorithm is *work-efficient*, if its work is $O(W)$, where $W$ is the work of the best known or the corresponding sequential algorithm. Also, note that any non-trivial computation that uses $\Omega(n^\epsilon)$ threads, for a constant, $\epsilon > 0$, requires $\Omega(\log n)$ span.

## III. Integer Sorting

In this section, we show the design of a critical component in our predictive sorting algorithm, integer sort on key range 0 to $n$. The algorithm uses $O(n)$ work and $O(\log n)$ span, and we show how to achieve high probability and even overwhelming probability with these performance bounds. The work and span bounds are optimal modulo randomization.

### A. Building Blocks

**Stable Integer Sorting.** This problem takes a sequence $A$ of $n$ elements, each element's key is an integer in the range $[0, r]$ with $r \leq n$. The output is elements of $A$ in sorted order, with the additional requirement that the elements with the same key must appear in the same relative order as in the input sequence $A$. This problem can be solved in $O(n)$ work and $O(r + \log n)$ span. We refer the audience to [43] for details and provide only a high-level overview here. The algorithm first (conceptually) partitions $A$ into $A/r$ subarrays, each with $r$ elements. Then a sequential counting sort is applied to each subarray to obtain both the sorted subarray and the counts for each key. This step uses linear work and $O(r + \log n)$ span. After that, a prefix sum for the counts across all subarrays computes the offsets for each key in each subarray (linear work and $O(\log n)$ span). Finally, a sequential scan in each subarray scatters all elements to their final locations based on the offsets, again with linear work and $O(r + \log n)$ span.

**Coin Flipping.** Many randomized processes in algorithms can be considered as coin flips, which eases the analysis of the probability. In the analysis of the algorithms in this paper, when we say a coin flip, we refer to a random event, and use the "head" of a flip to indicate a success and a "tail" as a failed one.

### B. The Parallel Integer Sorting Algorithm

In this section, we show how to integer sort $n$ elements with key range 0 to $n$ in linear work and $O(\log n)$ span, both with high probability and with overwhelming probability. Our algorithm follows the high-level idea of Rajasekaran and Reif [43], but with several key differences. First, we parameterize the algorithm to enable a careful analysis across possible parameter choices. Second, we redesign a major component that scatters the elements to their destinations, which can provide various levels of probabilistic guarantees on the success rate of the algorithm. We overview the algorithm in Algorithm 1 and discuss the scatter step in detail in section III-C.

Our integer-sort algorithm is randomized and takes samples. Each input element has $p = \Theta(1/\log n)$ probability to be selected as a sample, where $p$ is the sample rate here. The samples are then sorted with linear time and $O(\log n)$ span [19]. With the sorted samples, we can estimate the sizes of $np/\log^\gamma n$ evenly partitioned buckets, each with a key range of $\Theta(\log^\gamma n/p)$, where $\gamma$ is a positive integer. Here, $\log^\gamma n$ is the oversampling rate that reduces the number of buckets, which allows for the gathering of more samples and provides a stronger estimation for each bucket. With $s$ samples in one bucket, we take an upper bound estimate of $f(s) = 4(s + \log^\gamma n)/p$ for the number of elements in this range. We will first describe the algorithm and later show how to select the parameters that give the theoretical guarantees of this algorithm.

**Algorithm 1** Parallel Integer Sort

**Input:** An array $A$ with $n$ records each containing an integer key in the range from 0 to $n$.

**Output:** An array $A'$ storing the records of $A$ in sorted order.

1: Select a sample $S$ of the keys, independently with probability $p = \Theta(1/\log n)$.
2: Sort $S$.
3: Evenly partition the key range into $\Theta(np/\log^\gamma n)$ buckets, and create an array of size $f(s)$ for each bucket, where $s$ is the number of samples in this range.
4: Insert the records in $A$ into the associated array based on the key range.
5: Pack all of the arrays into a contiguous output array $A'$.
6: Run the stable integer sort for two rounds (each with range $\Theta(\log n)$) for sort each bucket, and return $A'$.

With the size estimation, we will allocate an array of size $2f(s)$ for each bucket. This can be achieved by allocating a sufficiently large array, and in parallel computing the offsets of each bucket based on the prefix sum, with $O(np/\log^\gamma n) = O(n)$ work and $O(\log n)$ span. The next step is to scatter the elements to the corresponding arrays of the buckets, which we will discuss later in section III-C, followed by packing. Finally, for each bucket, we run $(1 + \gamma)$ rounds of stable integer sort introduced in section III-A to sort the elements in each bucket, which takes $O((1 + \gamma)n)$ work and $O((1 + \gamma)\log n)$ span.

We now analyze the choice of the parameter $\gamma$.

**Lemma 2.** *Picking $\gamma = 1$ guarantees that $f(s)$ correctly upper bounds the number of elements in a bucket with high probability; $\gamma = 2$ yields overwhelming probability.*

**Proof:** To analyze the success rate that $f(s)$ is an upper bound for the number of elements in each bucket, we use a Chernoff bound on the event $X$ when there are $f(s)$ elements in this bucket while our sampling scheme only chooses no more than $s$ samples. Note that it is possible that the bucket contains more than $f(s)$ elements, but the probability only decreases. In this case, the expected number of samples is $\mu = f(s) \cdot p = 4(s + \log^\gamma n)$, and if the actual number of samples is no more than $s$, we have $\mu \geq 3/4$. By applying a Chernoff bound (Case (1) in Theorem 1), we have:

$$\Pr[X] < 2^{-\mu/2} < 2^{-2(s+\log^\gamma n)} \leq 2^{-2\log^\gamma n}.$$

Hence, plugging in $\gamma = 1$ gives the high probability bound, while $\gamma = 2$ provides the overwhelming probability. Taking the union bound across all buckets gives the stated bound. ■

### C. The Scatter Step

We now discuss different approaches to the scatter step that provide different probabilistic guarantees for the success rate. We start with the simple approach, and then show how to further boost the success rate to the overwhelming probability.

**Simple uniform scattering.** For our first, simple approach, we just scatter all elements to the buckets in parallel. For each element, we find a random location and try to reserve this slot. This can be achieved by applying an atomic TEST-AND-SET (TAS) (see section II for definition) to a boolean array that was initialized as false. It is possible that the slot is occupied by another previous or concurrent attempt, causing the TAS to fail. In this case, we just repeatedly find the next random location until an empty slot is found.

**Lemma 3.** *The above scatter algorithm uses $O(n)$ work and $O(\log n)$ span with high probability.*

**Proof:** Given that each bucket has been allocated at least twice the number of elements with high probability, each try has at least $1/2$ chance to succeed. Hence, scattering one element requires $O(\log n)$ tries with high probability in $n$ (getting one head in coin flipping), and taking the union bound gives the same bounds for scattering all elements. The total number of tries for all elements is $O(n)$ with high probability in $n$ (getting $n$ heads in coin flipping). Indeed, the work bound holds with probability at least $1-2^{-cn}$ for any constant $c > 0$, by applying a Chernoff bound (Case (2) in Theorem 1). ■

**Double-down scattering.** In our improved approach, ***double-down scattering***, we still follow the overall idea that scatters all elements to the buckets in parallel. The difference is that when the tries fail, we double the number of retries in the next round, so that the entire process will terminate much faster. In particular, in the $i$-th round, we in parallel try $2^{i-1}$ slots, namely, 1, 2, 4, 8, ..., until one empty slot is found. Note that to try $2^{i-1}$ slots in parallel, we need to fork this number of threads, which needs $\log_2 2^{i-1} = O(i)$ span. The work is dominated by the last round. Since this approach can block more slots with more tries in later rounds, each bucket is now allocated an array of size $4f(s)$ instead of $2f(s)$.

**Lemma 4.** *The improved scatter algorithm uses $O(n)$ work and $O(\log n)$ span with overwhelming probability.*

**Proof:** We start with the span analysis. Note that to achieve $O(\log n)$ span, the number of retrying rounds needs to be $O(\sqrt{\log n})$ since each round costs more than constant span. Now consider the $c\sqrt{\log n}$-th round for constant $c \geq 1$. The probability that this round fails is:

$$2^{-2^{c\sqrt{\log n}}} < 2^{-c\log^2 n} = n^{-c\log n}$$

which holds for any positive integer $n$. An overall overwhelming probability across all $n$ elements can therefore be obtained by taking a union bound.

The analysis for the work bound is more involved, since now the tries are in rounds, and the number of tries differs across rounds. We consider the $j$-th round across all $n$ elements. Here we simplify the analysis and assume that as long as the $2^{j-2}$ tries in the $(j-1)$-th round for the $i$-th element are not successful, we will apply the $2^{j-1}$ tries in the current ($j$-th round). Note that here we will overestimate the work since the $i$-th element can find a slot in previous rounds, but we will show that the relaxed work bound is still linear with overwhelming probability.

For an element in the $j$-th round, the probability that it does not find a slot in the previous round is $2^{-2^{j-2}}$, and the work on the $j$-th round for this element is $2^{j-1}$. Hence, we consider the work for each element as a coin flip, and with $2^{-2^{j-3}} > 2^{-2^{j-2}} \cdot 2^{j-1}$ (for $j \geq 5$) probability to see a head. We now show that the probability that we see more than $\Theta(n \cdot 2^{3-j})$ heads is negligible by using Chernoff bound. The expected number of heads is $\mu = n \cdot 2^{-2^{j-3}}$, and $\delta = 2^{3-j}/2^{-2^{j-3}} - 1 > 1$ for $j > 5$. By applying a Chernoff bound (Case (2) in Theorem 1), the probability that we see more than $\Theta(n \cdot 2^{3-j})$ heads is less than $2^{-\mu/2}$, which is exponentially small (smaller than the overwhelming probability). Also, we know that with overwhelming probability, none of the $n$ elements will need more than $\Theta(\sqrt{\log n})$ rounds. Hence, the total number of attempts is $\sum_{j=0}^{\Theta(\sqrt{\log n})} \Theta(n \cdot 2^{3-j}) = \Theta(n)$, and the success rate is achieved by taking the union bounds on all rounds.

In addition, note that trying multiple slots in a single round can reserve more than one slot per element, reducing the success rate of random attempts. However, we will show that the success rate, despite being reduced, remains over $1/2$. As mentioned before, currently the array size for each bucket is $4f(s)$, so compared to the previous approach, the success rate remains sufficient as long as the wasted space $w$ is no more than $f(s) = \Omega(\log^3 n)$. Indeed, we can show $w = O(\log^2 n)$ with overwhelming probability. Consider the last round of all elements, and assume the worst case that all slots are not occupied and thus wasted (ignoring the one that succeeds). Let this number be $w'$. In this case, it means all the tries in the previous round with $w'/2$ slots failed (if there is no previous round for an element, then no slot is wasted). The probability is $2^{-w'/2}$, and the probability is overwhelmingly small when $w' = \Omega(\log^2 n)$. Hence, with overwhelming probability, $w \leq w' = O(\log^2 n)$.

Combining the work and span analysis proves this lemma. ■

### D. The Overall Bounds

With all the lemmas given above, we can now give the main theorem of this section.

**Theorem 5.** *Sorting $n$ integers in the range of 0 to $n$ can be done in $O(n)$ work and $O(\log n)$ span with overwhelming probability in the binary-forking model.*

**Proof:** We can use Alg. 1 with $\gamma = 2$ and the improved scatter algorithm in Sec. III-C to achieve the stated bound. Lemma 2 guarantees that during scatter, each random attempt has at least 1/2 chance to find an empty slot, which allows theorem 4 to

show the work and span bounds for the scatter phase. The costs for all other steps are discussed in Sec. III-B. Combining all parts gives the stated bounds. ■

## IV. Predictive Sorting

In this section, we describe a robust parallel learning-augmented algorithm in the binary-forking model for sorting an array, $X = [x_1, x_2, \ldots, x_n]$, of $n$ comparable elements. Without loss of generality, let us assume that the elements in $X$ are distinct, since we can compare elements by a lexicographic combination of their values and initial index in $X$. For each element, $x_i \in X$, we are given a prediction, $p(x_i)$, for the ***rank***, $\mathrm{rank}(x_i)$, of $x_i$ in $X$, where $\mathrm{rank}(x_i)$ is the number of elements in $X$ that are less than $x_i$. One practical approach to acquire these ranks can be via *learned indexes* [12], [16]–[18], [29]–[31], [33], [34], [36]–[39], [41], [49], [50], [53]–[56] that learn the CDF of a given input. The high-level idea is to predict the locations via recursive piecewise linear functions. For sorting, we could, for example, build such an index based on a small random sample of the input (e.g., $\sqrt{n}$ elements) and use its prediction as the ranks for the later sorting algorithm. Note that even though we may assume that the elements in $X$ are distinct, we cannot assume that their predicted ranks are distinct.

Let $X'$ denote an ordering of the elements of $X$ according to their predicted ranks, breaking ties arbitrarily. That is, $X'$ is a possible result of performing an integer sorting of $X$ using the prediction, $p(x_i)$, as the key for each element, $x_i$ in $X$, e.g., based on our integer sorting algorithm given above.

Before we present our algorithm for sorting $X'$ (and, hence, sorting $X$), we observe that we can relate the prediction alignment error, $\mathrm{PAE}(X)$, to a classic metric of disorder for $X'$. Given an array, $X' = [x_1, x_2, \ldots, x_n]$, we define an ***inversion*** in $X'$ to be a pair $(x_i, x_j)$ such that $i < j$ but $x_i > x_j$. Also, define $\mathrm{Inv}(X')$ to be the total number of inversions in $X'$; see, e.g., [15]. Then we have the following:

**Lemma 6.** *Let $X'$ be a sorted ordering of the elements of $X$ according to their predicted ranks, breaking ties arbitrarily. Then $\mathrm{Inv}(X') \leq \mathrm{PAE}(X)$.*

**Proof:** Let $(x_i, x_j)$ be a pair of elements that define an inversion in $X'$, i.e., such that $i < j$ but $x_i > x_j$. Since $X'$ is sorted by predictions, this implies that $p(x_i) \leq p(x_j)$, which means that this pair is counted in $\mathrm{PAE}_j(X)$. ■

Note that it is possible that $\mathrm{Inv}(X') < \mathrm{PAE}(X)$, because we might have a pair, $(x_i, x_j)$, such that $p(x_i) = p(x_j)$ and $x_i > x_j$ but $(x_i, x_j)$ does not define an inversion in $X'$ because $x_i$ comes after $x_j$ in $X'$. We analyze our algorithm with respect to $\mathrm{PAE}(X)$ because it is a measure of prediction error, and we use $\mathrm{Inv}(X')$ to aid in this analysis, even though $\mathrm{Inv}(X')$ is an artifact of our algorithm rather than the prediction error. For example, we use the above lemma in our parallel learning-augmented sorting algorithms, which we characterize in terms of the logarithm of the average prediction alignment error, $\log(\mathrm{PAE}(X)/n)$. For example, Lemma 6 implies that $\log(\mathrm{Inv}(X')/n) \leq \log(\mathrm{PAE}(X)/n)$. Note that $\log(\mathrm{PAE}(X)/n)$ is always $O(\log n)$, but it can be as small as $O(1)$ if our machine-learning predictions are accurate. This property is useful for showing that our learning-augmented parallel sorting algorithms are robust and that they degrade slowly as a function of prediction error, i.e., they achieve a measure of smoothness.

As a preprocessing step for our parallel sorting algorithms, we sort the elements of $X$ using their predicted ranks as the keys, resulting in an array, $X'$, which we then use as the input array. For example, this preprocessing step can be performed using our parallel integer sorting algorithm given above.

### A. Our Base Sorting Algorithm

As a ***base sorting algorithm***, for sorting the array, $X'$, of size $n$, we use a classic sequential algorithm by Mannila [40], which we will apply to very small subproblems. Mannila gave a sorting algorithm that runs in $O(n(1+\log(\mathrm{Inv}(X')/n)+1))$ time, which, by Lemma 6, implies that this sequential base algorithm runs in $O(n(1+\log(\mathrm{PAE}(X)/n)+1))$ time. That is, considered as an algorithm in the binary-forking model, this algorithm has $O(n(1+\log(\mathrm{PAE}(X)/n)+1))$ span and work.

### B. Estimating $\mathrm{Inv}(X')$

As it turns out, an important common step in our parallel sorting algorithms is to estimate $\mathrm{Inv}(X')$. Our probabilistic decision algorithm independently chooses $cn$ pairs, $(x_i, x_j)$, from $X'$, for constant, $c \geq 5/4$, and with $i < j$, and counts the number of inversions, that is, we count when $x_i > x_j$. Let $Y$ denote the number of inversions that are found in this way and observe that $Y$ is defined as a sum of $cn$ independent $0/1$ random variables and

$$\mu = E[Y] = cn \cdot \frac{\mathrm{Inv}(X')}{\binom{n}{2}} = \frac{2c \cdot \mathrm{Inv}(X')}{n-1}.$$

The random sample and computation of $Y$ can easily be done with $O(\log n)$ span and $O(n)$ work in the binary-forking model.

We take as our first test to probabilistically test whether $\mathrm{Inv}(X')$ is ***small***, which we do by testing whether

$$Y < 2c \log^2 n,$$

and we accept $\mathrm{Inv}(X')$ as being small if this is true. Note that if $\mathrm{Inv}(X') = 0$, then $Y = 0$; hence, our test never fails for this boundary case. So, to bound our error probabilities, let us conservatively assume that $\mathrm{Inv}(X') > 0$, and consider the possible error probabilities. We begin by bounding the probability that our test underestimates the size of $\mathrm{Inv}(X')$.

**Lemma 7.** *If $\mathrm{Inv}(X') \geq 2n \log^2 n$ and if $c \geq 5/4$, then $\Pr(Y < 2c \log^2 n) < n^{-\log n}$, which is negligible.*

**Proof:** Since $\mathrm{Inv}(X') \geq 2n \log^2 n$, let us conservatively assume, for the sake of the analysis for this proof, that

$\mathrm{Inv}(X') = 2(n-1)\log^2 n$. So $\mu = 4c\log^2 n$. Then, by a Chernoff bound (Case (3) in Theorem 1),

$$\Pr(Y < 2c\log^2 n) = \Pr(Y < \mu/2) < 2^{-\mu/5} = n^{-(4c/5)\log n},$$

which is at most $n^{-\log n}$. ∎

Our next lemma bounds the probability that our test overestimates the size of $\mathrm{Inv}(X')$.

**Lemma 8.** *If $\mathrm{Inv}(X') \le ((n-1)/2)\log^2 n$ and $c \ge 1$, then $\Pr(Y > 2c\log^2 n) < n^{-c\log n}$, which is negligible.*

**Proof:** Let us conservatively assume, for the sake of analysis, that $\mathrm{Inv}(X') = ((n-1)/2)\log^2 n$. So $\mu = c\log^2 n$. Then, by a Chernoff bound (Case (2) in Theorem 1),

$$\Pr(Y > 2c\log^2 n) = \Pr(Y > 2\mu) < 2^{-\mu/2} = n^{-c\log n}.$$

∎

Thus, if the outcome of our test is that $Y < 2c\log^2 n$, for $c \ge 5/4$, then, with overwhelming probability, $\mathrm{Inv}(X')$ is at most $2n\log^2 n$. That is, our test for whether $\mathrm{Inv}(X')$ is small succeeds with overwhelming probability.

Moreover, if the outcome of our probabilistic test is that $Y \ge 2c\log^2 n$, then, with overwhelming probability, $\mathrm{Inv}(X')$ is at least $((n-1)/2)\log^2 n$; hence, in this case, $\log(\mathrm{Inv}(X')/n)$ is $\Omega(\log\log n)$ with overwhelming probability. Also, in this case, let us estimate $\mathrm{Inv}(X')$ as

$$\lambda = \frac{Y}{cn} \cdot \binom{n}{2} = \frac{Y(n-1)}{2c}.$$

The following lemma bounds the probability that $\lambda$ is an underestimate, which would be a bad scenario for our algorithms.

**Lemma 9.** *Suppose $\mathrm{Inv}(X') \ge ((n-1)/2)\log^2 n$, $c \ge 1$, and $n \ge 32$. Then $\Pr(\lambda < \mathrm{Inv}(X')/2) < n^{-\log n}$, which is negligible.*

**Proof:** Note that in this case, using the same $\mu$ as above, $\mu \ge c\log^2 n \ge 5\log n$. Also,

$$\begin{aligned}\Pr\left(\lambda < \frac{\mathrm{Inv}(X')}{2}\right) &= \Pr\left(\frac{Y(n-1)}{2c} < \frac{\mathrm{Inv}(X')}{2}\right)\\ &= \Pr\left(Y < \frac{c\cdot\mathrm{Inv}(X')}{n-1}\right)\\ &= \Pr(Y < \mu/2).\end{aligned}$$

Then, by a Chernoff bound (Case (3) in Theorem 1),

$$\Pr\left(\lambda < \frac{\mathrm{Inv}(X')}{2}\right) = \Pr\left(Y < \frac{\mu}{2}\right) < 2^{-\mu/5},$$

which is at most $n^{-\log n}$. ∎

To sum up, then, if our test determines that $\mathrm{Inv}(X')$ is small, then, with overwhelming probability, $\mathrm{Inv}(X') \le 2n\log^2 n$. Also, if our test determines that $\mathrm{Inv}(X')$ is not small, then, with overwhelming probability, $\mathrm{Inv}(X') > ((n-1)/2)\log^2 n$ and $\lambda \ge \mathrm{Inv}(X')/2$.

### C. Our Level-2 Parallel Sorting Algorithm

In this subsection, we describe our level-2 parallel sorting algorithm, which has polylogarithmic span and work that is $O(n(1+\log((\mathrm{PAE}(X)/n)+1)))$ w.o.p. Our method is based on combining our base sorting algorithm with a non-trivial adaptation of a "block-sort" algorithm by Levcopoulos and Petersson [35]. Our algorithm is shown in Algorithm 2.

At a high level, our algorithm first divides the input into a set of blocks of size, $L$, where our choice of $L$ depends on whether we determined that $\mathrm{Inv}(X')$ is small or not. See Figure 2.

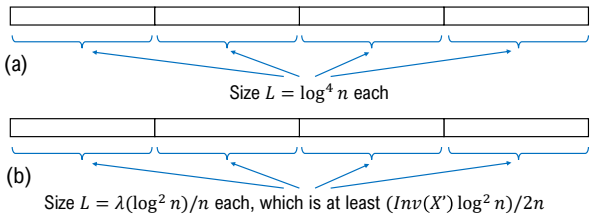


Fig. 2. How we divide the array, $X'$, into blocks of size $L$. (a) the scenario when we determine that $\mathrm{Inv}(X')$ is small; (b) the scenario when we determine that $\mathrm{Inv}(X')$ is not small.

Then, we sort each block using an existing method. If we are in the scenario where $\mathrm{Inv}(X')$ was determined to be small, then we do this using the sequential algorithm of Mannila [40], which runs in polylogarithmic time, since $L = \log^4 n$, and with work that is proportional to $n$ times the logarithm of the average number of inversions in the block. Alternatively, if we are in the scenario where $\mathrm{Inv}(X')$ was determined not to be small, then we sort each block using the sorting algorithm of Goodrich and Jacob [19], which is deterministic and takes $O(\log L)$ span and $O(L\log L)$ work.

Next, we order the blocks according to a sorting of their medians, which we can do in $O(\log(n/L))$ span and $((n/L)\log(n/L))$ work, by another call to the deterministic parallel sorting algorithm of Goodrich and Jacob [19] for the binary-forking model. See Figure 3.

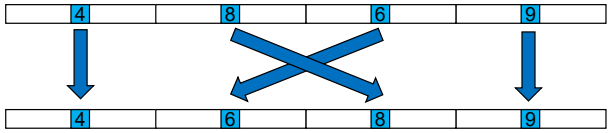


Fig. 3. Sorting the blocks by their medians.

We follow this step by trimming away a total of $R$ elements, by removing elements between the medians that are out of order, which we can do by first performing a set of parallel binary searches and then doing a parallel prefix computation and compressing the trimmed and untrimmed elements into respective lists. Then, we sort the list of $R$ trimmed by yet another call to the deterministic parallel sorting algorithm of Goodrich and Jacob [19]. Note that the untrimmed elements are already in sorted order. So we complete the sorting algorithm by merging the sorted list of trimmed elements with the set of sorted untrimmed elements. See Figure 4.

**Algorithm 2** Our level-2 parallel sorting algorithm.

1: Perform a parallel integer sorting of $X$ based on using the machine-learning predictions as keys to give $X'$.
2: Use the method of the previous subsection to compute $Y$ and $\lambda$ (recall that $\lambda$ is our estimate for $\mathrm{Inv}(X')$).
3: **if** $\lambda > n^{3/2}$ **then**
4: Sort $X'$ using the sorting algorithm of Goodrich and Jacob [19], which in this case will require $O(\log n)$ span and $O(n \log n) = O(n(1+\log((\mathrm{Inv}(X')/n)+1)))$ work, w.o.p., and we are done.
5: **if** $Y < 2c \log^2 n$ **then**
6: Divide $X'$ into $n/L$ blocks of size $L$ each, where $L = \log^4 n$ and sort each block using the sequential algorithm of Mannila [40].
7: **else**
8: Divide $X'$ into $n/L$ blocks of size $L$ each, where $L = \lambda(\log^2 n)/n$, and sort each block using the sorting algorithm of Goodrich and Jacob [19], which will require $O(\log L)$ span and $O(L \log L)$ work per block; hence, all together, w.o.p., this step uses $O(n \log(\mathrm{Inv}(X')/n))$ work, since, w.o.p., in this case $\log L$ is $O(\log(\mathrm{Inv}(X')/n) + \log\log n) = O(\log(\mathrm{Inv}(X')/n))$.
9: Sort the medians of each block using the sorting algorithm of Goodrich and Jacob [19], which in this step will require $O(\log n)$ span and $O((n/L)\log(n/L)) = O(n)$ work.
10: Permute the blocks so that the block are ordered by non-decreasing values of their medians. Denote this sequence of blocks as $B_1, B_2, \ldots, B_{n/L}$.
11: Trim the blocks. Let $X_1, Y_1, X_2, Y_3, X_3, \ldots, X_{n/L}, Y_{n/L}$ denote the list of consecutive half blocks (divided by the medians) such that $B_i = X_i Y_i$. For each $i = 1, 2, \ldots, (n/L) - 1$, determine the minimum value, $d_i$, such that if we remove $d_i$ elements from the end of $Y_i$ and $d_i$ elements from the beginning of $X_{i+1}$, the remaining sequence of elements in $Y_i X_{i+1}$ is in sorted order. This step can be performed by a collection of parallel binary searches followed by a parallel prefix computation to have $O(\log n)$ span and linear work.
12: Sort the removed elements using the sorting algorithm of Goodrich and Jacob [19], which will require $O(\log n)$ span and $O(R \log R)$ work, where $R$ is the number of removed elements in the previous step. We will show that $R$ is $O(n/\log n)$ with overwhelming probability; hence, this step takes $O(n)$ work w.o.p.
13: Merge the ordered list of removed elements and the sorted list of elements that were not trimmed. This can be done using a binary-forking implementation of the merge algorithm of Bilardi and Nicolau [4] to be done in $O(\log n)$ span and $O(n)$ work.

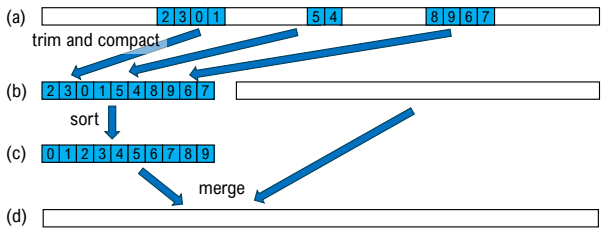


Fig. 4. Trimming the blocks and then merging the resulting two sorted lists. (a) trimming elements in adjacent blocks; (b) compacting the trimmed elements; (c) sorting the trimmed elements; (d) merging the trimmed and untrimmed elements.

**Lemma 10.** *All together, Step 6 requires $O(L \log L)$ span and $O(n(1 + \log((\mathrm{Inv}(X')/n) + 1)))$ work.*

**Proof:** If we let $B_i$ denote the $i$-th block, then sorting $B_i$ using Mannila's sequential algorithm in Step 6 requires span $O(L(1+\log((\mathrm{Inv}(B_i)/L)+1)))$, which is at most $O(L \log L)$. The work for performing the sorting of $B_i$ is likewise $O(L(1+\log((\mathrm{Inv}(B_i)/L) + 1)))$. Let

$$\beta = \sum_{i=j}^{n} I_j = \sum_{i=1}^{n/L} \mathrm{Inv}(B_i).$$

Also, for $j = 1, 2, \ldots, n$, if $\lceil j/L \rceil = i$, then let $I_j = \mathrm{Inv}(B_i)/L$. Then

$$\sum_{i=j}^{n} I_j = \beta \leq \mathrm{Inv}(X').$$

Let us consider the total work performed in this step for all the blocks, which is proportional to

$$\begin{aligned}
&\sum_{i=1}^{n/L} L(1 + \log((\mathrm{Inv}(B_i)/L) + 1)) && (1)\\
&= \sum_{j=1}^{n} (1 + \log(I_j + 1)) && (2)\\
&= n + \log\left(\prod_{j=1}^{n} (I_j + 1)\right) && (3)\\
&= n + n \log\left(\prod_{j=1}^{n} (I_j + 1)\right)^{1/n} && (4)\\
&\leq n + n \log\left(\frac{\sum_{j=1}^{n} (I_j + 1)}{n}\right) && (5)\\
&= n\,(1 + \log((\beta/n) + 1)) && (6)\\
&\leq n\,(1 + \log((\mathrm{Inv}(X')/n) + 1))\,, && (7)
\end{aligned}$$

where equation (5) is due to the fact that the geometric mean is never greater than the arithmetic mean. Also, note that $L$ term is dropped when we go from (1) to (2) because the sum in (2) goes to $n$, rather than $n/L$ in (1). This, together with the definition of $I_j$, means we are counting the term in the sum in (2) $L$ times for each $i$. ■

Our next set of lemmas are for bounding the size of $R$.

**Lemma 11.** $\mathrm{Inv}(X_1X_2\cdots X_{2n/L}) \leq 3\cdot \mathrm{Inv}(X')$.

**Proof:** The proof follows by the proof of a similar lemma (Lemma 3) of Levcopoulos and Petersson [35]. ∎

**Lemma 12.** *$R$ is $O(n/\log n)$ w.o.p.*

**Proof:** The proof follows a proof technique of Levcopoulos and Petersson [35], but with better probability bounds. Note that, by the definition of the trimming step,

$$R = \sum_{i=1}^{n/L-1} 2d_i.$$

Also by the definition of the trimming step, for each $i = 1, 2, \ldots, (n/L)-1$, the $d_i$ elements we remove from $Y_i$ and the $d_i$ elements we remove from $X_{i+1}$ cause $d_i^2$ inversions in $X'$. Thus, by Lemma 11,

$$\sum_{i=1}^{n/L-1} d_i^2 \;\leq\; 3\cdot \mathrm{Inv}(X').$$

Recall that for two sequences of numbers, $(a_1, a_2, \ldots, a_m)$ and $(b_1, b_2, \ldots, b_m)$, the Cauchy-Schwarz inequality implies that

$$\left(\sum_{i=1}^{m} a_ib_i\right)^2 \leq \left(\sum_{i=1}^{m} a_i^2\right)\cdot\left(\sum_{i=1}^{m} b_i^2\right).$$

Now, let $m = n/L-1$ and, for each $i = 1, 2, \ldots, n/L-1$, take $a_i = d_i$ and $b_i = 1$. Then Lemma 11 and the Cauchy-Schwarz inequality imply that

$$\begin{aligned}\left(\sum_{i=1}^{n/L-1} d_i\right)^2 &\leq \left(\sum_{i=1}^{n/L-1} d_i^2\right)\cdot\left(\sum_{i=1}^{n/L-1} 1^2\right)\\ &= (n/L-1)\sum_{i=1}^{n/L-1} d_i^2\\ &\leq 3(n/L-1)\mathrm{Inv}(X').\end{aligned}$$

Thus,

$$R \leq 2\sqrt{3(n/L-1)\mathrm{Inv}(X')} \leq 2\sqrt{(3n)\mathrm{Inv}(X')/L}.$$

Accordingly, if we perform Step 6, then, with overwhelming probability, $\mathrm{Inv}(X') \leq 2n\log^2 n$; hence, choosing $L = \log^4 n$ implies that $R$ is $O(n/\log n)$ w.o.p. Alternatively, if we perform Step 8, then our choosing $L = \lambda(\log^2 n)/n$ implies that $R$ is $O(n/\log n)$ w.o.p. ∎

Thus, we have the following.

**Lemma 13.** *We can sort $X'$ in $O(\log^5 n)$ span and work $O(n(1+\log(\mathrm{Inv}(X')/n+1)))$ w.o.p.*

### *D. Our Level-3 Parallel Sorting Algorithm*

Our level-3 algorithm is the same as our level-2 parallel sorting algorithm, except for a couple of modifications. First, in our level-3 algorithm we perform Step 6 by using our level-2 sorting algorithm, which will run in polyloglog time and total work that is at most $O(n(1+\log(\mathrm{Inv}(X')/n+1)))$. These calls succeed with overwhelming probability in terms of the problem size of $L = \log^4 n$, which is admittedly not necessarily with overwhelming probability in terms of $n$. So, we do a simple test in $O(\log n)$ span and linear work to determine which subproblems failed to sort their blocks, and we then use a simplified failure-sweeping technique for them. Namely, for each failed subproblem, we use the deterministic method of Goodrich and Jabob [19] to sort each of them in $O(\log L)$ span and $O(L\log L)$ work.

**Lemma 14.** *There will be fewer than $2n/\log^8 n$ failing subproblems w.o.p.*

**Proof:** Since subproblems succeed with overwhelming probability in terms of subproblems of size $L = \log^4 n$, let us conservatively assume each subproblem fails (independently) with probability at most $1/\log^4 n$. Then the expected number of failed subproblems is at most $\mu = n/\log^8 n$. Thus, by a Chernoff bound (Case (2) in Theorem 1), with overwhelming probability, there will be fewer than $2\mu$ such failed subproblems. ∎

Thus, by this lemma, the total size of all failed subproblems is $O(n/\log^4 n)$ w.o.p.; hence, w.o.p. we can sort all of them by the Goodrich and Jacob algorithm [19] with $O(\log n)$ span and $O(n)$ work. Furthermore, the overhead of compressing such subproblems and merging the two resulting sorted lists of (now-sorted) failed subproblems and the successfully sorted subproblems at the end can be done in $O(\log n)$ time and $O(n)$ work. Thus, all together, this implies the following.

**Theorem 15.** *Suppose we are given an array, $X = (x_1, x_2, \ldots, x_n)$, of elements that come from a total order, such that, for each element $x_i$ in $X$, we have a machine-learning prediction, $p(x_i)$, for the rank of $x_i$ in $X$. Then we can sort $X$ with $O(\log n)$ span and $O(n(1+\log(\mathrm{PAE}(X)/n+1)))$ work w.o.p.*

## V. Conclusion and Future Work

We have shown how to solve the integer sorting problem in $O(\log n)$ span and linear work with overwhelming probability in the binary-forking parallel model. We have also provided a parallel learning-augmented sorting algorithm that runs in $O(\log n)$ span and work that is proportional to $n$ times the logarithm of the average prediction alignment error.

For future work, it would be interesting to study the learning-augmented parallel algorithms for more practical sorting algorithms such as samplesort. Also, we have taken the rank predictions as a "black box" in this paper; it would be interesting to explore parallel algorithms for learning input distributions, e.g., parallelizing sequential algorithms for this [12], [16]–[18], [31], [33], [33], [38], [55], [56].

## ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for their careful reading of a preliminary version of this paper and also for their helpful comments. This work was supported by NSF grant CCF-2212129 and CCF-2339310.

## REFERENCES

[1] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015, software available from tensorflow.org.

[2] N. S. Arora, R. D. Blumofe, and C. G. Plaxton, "Thread scheduling for multiprogrammed multiprocessors," *Theory of Computing Systems (TOCS)*, vol. 34, no. 2, pp. 115–144, 2001.

[3] X. Bai and C. Coester, "Sorting with predictions," in *37th Int. Conf. on Neural Information Processing Systems (NeurIPS)*, no. 1155, 2023, pp. 26 563–26 584.

[4] G. Bilardi and A. Nicolau, "Adaptive bitonic sorting: An optimal parallel algorithm for shared-memory machines," *Inform. Comput.*, vol. 18, pp. 216–228, 1989.

[5] G. E. Blelloch, J. T. Fineman, Y. Gu, and Y. Sun, "Optimal parallel algorithms in the binary-forking model," in *32nd ACM Symp. on Parallelism in Algorithms and Architectures (SPAA)*, 2020, pp. 89–102.

[6] R. D. Blumofe and C. E. Leiserson, "Scheduling multithreaded computations by work stealing," *J. ACM*, vol. 46, no. 5, pp. 720–748, 1999.

[7] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 3rd ed. The MIT Press, 2009.

[8] S. Davies, B. Moseley, S. Vassilvitskii, and Y. Wang, "Predictive flows for faster Ford-Fulkerson," in *Int. Conf. on Machine Learning (ICML)*, A. Krause, E. Brunskill, K. Cho, B. Engelhardt, S. Sabato, and J. Scarlett, Eds., vol. 202, 2023, pp. 7231–7248.

[9] M. B. Dillencourt and M. T. Goodrich, "Simplified Chernoff bounds with powers-of-two probabilities," *Inf. Process. Lett.*, vol. 182, p. 106397, 2023.

[10] M. B. Dillencourt, M. T. Goodrich, and M. Mitzenmacher, "Leveraging parameterized Chernoff bounds for simplified algorithm analyses," *Inf. Process. Lett.*, vol. 187, p. 106516, 2025.

[11] J. Ding, R. Marcus, A. Kipf, V. Nathan, A. Nrusimha, K. Vaidya, A. van Renen, and T. Kraska, "SageDB: An instance-optimized data analytics system," *Proc. VLDB Endow.*, vol. 15, no. 13, pp. 4062–4078, 2022.

[12] J. Ding, U. F. Minhas, J. Yu, C. Wang, J. Do, Y. Li, H. Zhang, B. Chandramouli, J. Gehrke, D. Kossmann *et al.*, "ALEX: an updatable adaptive learned index," in *ACM Int. Conf. on Management of Data (SIGMOD)*, 2020, pp. 969–984.

[13] M. Dinitz, S. Im, T. Lavastida, B. Moseley, A. Niaparast, and S. Vassilvitskii, "Binary search with distributional predictions," in *Conf. on Neural Information Processing Systems (NeurIPS)*, A. Globersons, L. Mackey, D. Belgrave, A. Fan, U. Paquet, J. M. Tomczak, and C. Zhang, Eds., 2024.

[14] X. Dong, L. Dhulipala, Y. Gu, and Y. Sun, "Parallel integer sort: Theory and practice," *arXiv preprint:2401.00710*, 2024.

[15] V. Estivill-Castro and D. Wood, "A survey of adaptive sorting algorithms," *ACM Comput. Surv.*, vol. 24, no. 4, p. 441–476, December 1992.

[16] P. Ferragina, F. Lillo, and G. Vinciguerra, "Why are learned indexes so effective?" in *Int. Conf. on Machine Learning*. PMLR, 2020, pp. 3123–3132.

[17] P. Ferragina and G. Vinciguerra, "The PGM-index: a fully-dynamic compressed learned index with provable worst-case bounds," *Proc. IEEE International Conf. on Very Large Databases*, vol. 13, no. 8, pp. 1162–1175, 2020.

[18] J. Ge, H. Zhang, B. Shi, Y. Luo, Y. Guo, Y. Chai, Y. Chen, and A. Pan, "SALI: A scalable adaptive learned index framework based on probability models," *Proc. Int. Conf. on Management of Data*, vol. 1, no. 4, pp. 1–25, 2023.

[19] M. T. Goodrich and R. Jacob, "Optimal parallel sorting with comparison errors," in *Proceedings of the 35th ACM Symposium on Parallelism in Algorithms and Architectures, SPAA 2023, Orlando, FL, USA, June 17-19, 2023*, K. Agrawal and J. Shun, Eds. ACM, 2023, pp. 355–365.

[20] M. T. Goodrich, Y. Matias, and U. Vishkin, "Optimal parallel approximation for prefix sums and integer sorting," in *ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 1994.

[21] M. T. Goodrich and R. Tamassia, *Algorithm design and applications*. Wiley Publishing, 2015.

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

[23] Y. Gu, J. Shun, Y. Sun, and G. E. Blelloch, "A top-down parallel semisort," in *27th ACM Symp. on Parallelism in Algorithms and Architectures (SPAA)*, 2015, pp. 24—-34.

[24] Y. Han and X. Shen, "Parallel integer sorting is more efficient than parallel comparison sorting on exclusive write PRAMs," *SIAM Journal on Computing*, vol. 31, no. 6, pp. 1852–1878, 2002.

[25] C.-Y. Hsu, P. Indyk, D. Katabi, and A. Vakilian, "Learning-based frequency estimation algorithms," in *7th International Conference on Learning Representations*, 2019.

[26] P. Indyk, F. Mallmann-Trenn, S. Mitrovic, and R. Rubinfeld, "Online page migration with ML advice," in *Int. Conf. on Artificial Intelligence and Statistics*. PMLR, 2022, pp. 1655–1670.

[27] P. Indyk, A. Vakilian, and Y. Yuan, "Learning-based low-rank approximations," in *Conf. on Neural Information Processing Systems (NeurIPS*, H. M. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. B. Fox, and R. Garnett, Eds., 2019, pp. 7400–7410.

[28] J. Katz and Y. Lindell, *Introduction to Modern Cryptography: Principles and Protocols*. Chapman and hall/CRC, 2007.

[29] C. Kim, J. Chhugani, N. Satish, E. Sedlar, A. D. Nguyen, T. Kaldewey, V. W. Lee, S. A. Brandt, and P. Dubey, "Fast: fast architecture sensitive tree search on modern cpus and gpus," in *Proc. Int. Conf. on Management of Data*, 2010, pp. 339–350.

[30] A. Kipf, R. Marcus, A. van Renen, M. Stoian, A. Kemper, T. Kraska, and T. Neumann, "SOSD: A benchmark for learned indexes," *arXiv preprint arXiv:1911.13014*, 2019.

[31] ——, "RadixSpline: a single-pass learned index," in *Int. Workshop on Exploiting Artificial Intelligence Techniques for Data Management*, 2020, pp. 1–5.

[32] E. R. Knorr, B. Lemaire, A. Lim, S. Luo, H. Zhang, S. Idreos, and M. Mitzenmacher, "Proteus: A self-designing range filter," in *Int. Conf. on Management of Data (SIGMOD)*, Z. G. Ives, A. Bonifati, and A. E. Abbadi, Eds., 2022, pp. 1670–1684.

[33] T. Kraska, A. Beutel, E. H. Chi, J. Dean, and N. Polyzotis, "The case for learned index structures," in *Proc. Int. Conf. on Management of Data*, 2018, pp. 489–504.

[34] V. Leis, A. Kemper, and T. Neumann, "The adaptive radix tree: ARTful indexing for main-memory databases," in *IEEE Int. Conf. on Data Engineering (ICDE)*. IEEE, 2013, pp. 38–49.

[35] C. Levcopoulos and O. Petersson, "Exploiting few inversions when sorting: Sequential and parallel algorithms," *Theoretical Computer Science*, vol. 163, no. 1, pp. 211–238, 1996.

[36] M. Li, H. Chai, S. Luo, H. Dai, R. Gu, J. Zheng, and G. Chen, "VEGA: An active-tuning learned index with group-wise learning granularity," *Proc. Int. Conf. on Management of Data*, vol. 3, no. 1, pp. 1–26, 2025.

[37] P. Li, Y. Hua, J. Jia, and P. Zuo, "FINEdex: a fine-grained learned index scheme for scalable and concurrent memory systems," *Proc. IEEE International Conf. on Very Large Databases*, vol. 15, no. 2, pp. 321–334, 2021.

[38] P. Li, H. Lu, R. Zhu, B. Ding, L. Yang, and G. Pan, "DILI: A distribution-driven learned index," *Proc. IEEE International Conf. on Very Large Databases*, vol. 16, no. 9, pp. 2212–2224, 2023.

[39] M. Maltry and J. Dittrich, "A critical analysis of recursive model ikraska2018casendexes," *Proc. VLDB Endow.*, vol. 15, no. 5, p. 1079–1091, January 2022.

[40] H. Mannila, "Measures of presortedness and optimal sorting algorithms," *IEEE Transactions on Computers*, vol. C-34, no. 4, pp. 318–325, 1985.

[41] Y. Mao, E. Kohler, and R. T. Morris, "Cache craftiness for fast multicore key-value storage," in *ACM European Conf. on Computer Systems*, 2012, pp. 183–196.

[42] M. Mitzenmacher and S. Vassilvitskii, *Algorithms with Predictions*. Cambridge University Press, 2021, p. 646–662.

[43] S. Rajasekaran and J. H. Reif, "Optimal and sublogarithmic time randomized parallel sorting algorithms," *SIAM J. on Computing*, vol. 18, no. 3, pp. 594–607, 1989.

[44] T. Roughgarden, "Beyond worst-case analysis," *Communications of the ACM*, vol. 62, no. 3, pp. 88–96, 2019.
[45] ——, *Beyond the Worst-Case Analysis of Algorithms*. Cambridge University Press, 2021.
[46] I. Sabek and T. Kraska, "The case for learned in-memory joins," *Proc. VLDB Endow.*, vol. 16, no. 7, pp. 1749–1762, 2023.
[47] R. Shahout and M. Mitzenmacher, "SkipPredict: When to invest in predictions for scheduling," in *Conf. on Neural Information Processing Systems (NeurIPS)*, A. Globersons, L. Mackey, D. Belgrave, A. Fan, U. Paquet, J. M. Tomczak, and C. Zhang, Eds., 2024.
[48] R. Shahout, I. Sabek, and M. Mitzenmacher, "Learning-augmented frequency estimation in sliding windows," in *32nd IEEE Int. Conf. on Network Protocols (ICNP)*, 2024, pp. 1–6.
[49] Z. Sun, X. Zhou, and G. Li, "Learned index: A comprehensive experimental evaluation," *Proc. IEEE International Conf. on Very Large Databases*, vol. 16, no. 8, pp. 1992–2004, 2023.
[50] C. Tang, Y. Wang, Z. Dong, G. Hu, Z. Wang, M. Wang, and H. Chen, "XIndex: a scalable learned index for multicore data storage," 2020, pp. 308–320.
[51] K. Vaidya, S. Chatterjee, E. Knorr, M. Mitzenmacher, S. Idreos, and T. Kraska, "SNARF: a learning-enhanced range filter," *Proceedings of the VLDB Endowment*, vol. 15, no. 8, pp. 1632–1644, 2022.
[52] Various, "Algorithms with predictions," https://algorithms-with-predictions.github.io/, 2025.
[53] C. Wongkham, B. Lu, C. Liu, Z. Zhong, E. Lo, and T. Wang, "Are updatable learned indexes ready?" *arXiv preprint arXiv:2207.02900*, 2022.
[54] J. Wu, Y. Zhang, S. Chen, Y. Chen, J. Wang, and C. Xing, "Updatable learned index with precise positions," *Proc. IEEE International Conf. on Very Large Databases*, vol. 14, no. 8, pp. 1276–1288, 2021.
[55] J. Zhang and Y. Gao, "CARMI: a cache-aware learned index with a cost-based construction algorithm," *arXiv preprint arXiv:2103.00858*, 2021.
[56] S. Zhang, J. Qi, X. Yao, and A. Brinkmann, "Hyper: A high-performance and memory-efficient learned index via hybrid construction," *Proc. Int. Conf. on Management of Data*, vol. 2, no. 3, pp. 1–26, 2024.