UNIVERSITY OF CALIFORNIA,
IRVINE

Computing under Constraints: Efficient Fault-Tolerant and Memory-Bound Algorithms

DISSERTATION

submitted in partial satisfaction of the requirements
for the degree of

DOCTOR OF PHILOSOPHY

in Computer Science

by

Vinesh Sridhar

Dissertation Committee:
Distinguished Professor Michael T. Goodrich, Chair
Distinguished Professor David Eppstein
Professor Sandy Irani

2026

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ALGORITHMS

ACKNOWLEDGMENTS

I want to thank my advisor, Mike Goodrich. The past three years have not always been a smooth ride, but I was always sure I had your full support. You inspired and challenged me to develop as a researcher and communicator. Under your guidance, I have done work I am proud of.

Erica and Jon, thank you for the opportunity to do undergraduate research at the University of Maryland. You made me feel welcome in a research community. Thanks to Taj as well, who gave me the confidence to pursue a PhD.

I am grateful to all the other coauthors I have had the privilege of working with: Rolf Svenning, Ryuto Kitagawa, Ofek Gila, David Eppstein, Auguste Gezalyan, David Mount, Prabhav Goyal, and Wilson Zheng. I want to especially thank Rolf Svenning, with whom I wrote my first paper. I am glad I asked to meet you for lunch that day during your visit from Aarhus University, which started a fruitful mentorship and collaboration. I also want to thank Ofek Gila. You helped me understand the big and little details that turn good research into great research. Prabhav and Wilson, thank you for trusting in me to guide your first research experiences.

Thanks to the current and former members of the theory lab during my time here, especially Abraham, Ofek, Claire, Ryuto, and Alvin. I felt like a member of a community with you all, which made the lonely parts of the PhD easier to bear.

Christian, thank you for the rides to all the shows, gigs, and rehearsals, and for being a friend I could geek out about jazz with. Arushi and Sadhana, thanks for showing me that there is more to Orange County than my office and my apartment.

Thanks to Dr. Amirthakadeswarar, who showed me that the sorrows of life are lessened by the sweetness of Tamil language.

Lillian, despite being over two thousand miles away, I have always felt you by my side. I am so grateful to have lived and grown with you these past three years and so happy to be together again soon. All my love.

Amma, Appa, and Nitesh, you have cared for me since long before I knew to appreciate it. A thank-you seems like it could never be enough. So with this acknowledgement comes a promise from me to return in kind what you have given me, to you and to others.

This work was partially supported by NSF grant CCF-2212129. Chapter 4 is reproduced with permission from Springer Nature as it appears in [73]. Chapter 5 is reprinted, with permission, from [77].

VITA

Vinesh Sridhar

EDUCATION

Doctor of Philosophy in Computer Science — **2026**
University of California, Irvine — *Irvine, CA*

Bachelor of Science in Computer Science — **2022**
University of Maryland, College Park — *College Park, MD*

EXPERIENCE

Teaching Assistant — **2024-2026**
University of California, Irvine — *Irvine, CA*

Research Fellow — **2023-2024**
University of California, Irvine — *Irvine, CA*

Product Manager — **2022**
EqualLevel, Inc. — *Rockville, MD*

Undergraduate Researcher — **2021-2022**
University of Maryland, College Park — *College Park, MD*

Software Engineering Intern — **2019-2021**
EqualLevel, Inc. — *Rockville, MD*

PUBLICATIONS

Improved Algorithms for Voronoi Diagrams, Delaunay Triangulations, and Support Hulls in Polygonal Hilbert Geometries **2026**
Auguste Gezalyan, Michael T. Goodrich, David M. Mount, and Vinesh Sridhar
Canadian Conference on Computational Geometry

Raiders of the Lost Log: Synchronous Parallel In-Place Models and Algorithms **2026**
Michael T. Goodrich and Vinesh Sridhar
Workshop on Advances in Parallel and Distributed Computational Models

How to Sort in a Refrigerator: Simple Entropy-Sensitive Strictly In-Place Sorting Algorithms **2026**
Ofek Gila, Michael T. Goodrich, and Vinesh Sridhar
Latin American Theoretical Informatics Symposium

Optimal Parallel Algorithms for Convex Hulls in 2D and 3D under Noisy Primitive Operations **2025**
Michael T. Goodrich and Vinesh Sridhar
Canadian Conference on Computational Geometry

Computational Geometry with Probabilistically Noisy Primitive Operations **2025**
David Eppstein, Michael T. Goodrich, and Vinesh Sridhar
Algorithms and Data Structures Symposium

Fast Geographic Routing in Fixed-Growth Graphs **2025**
Ofek Gila, Michael T. Goodrich, Abraham M. Illickan, and Vinesh Sridhar
International Conference on Algorithms and Complexity

Dynamic Accountable Storage: An Efficient Protocol for Real-Time Cloud Storage Auditing **2024**
Michael T. Goodrich, Ryuto Kitagawa, and Vinesh Sridhar
International Symposium on Algorithmic Aspects of Cloud Computing

Fast Area-Weighted Peeling of Convex Hulls for Outlier Detection **2024**
Vinesh Sridhar and Rolf Svenning
Canadian Conference on Computational Geometry

ABSTRACT OF THE DISSERTATION

Computing under Constraints:
Efficient Fault-Tolerant and Memory-Bound Algorithms

By

Vinesh Sridhar

Doctor of Philosophy in Computer Science

University of California, Irvine, 2026

Distinguished Professor Michael T. Goodrich, Chair

Theoretical analysis of algorithms relies on idealized models of computers. This inevitably creates a gap between what is efficient in theory and in practice, motivating fine-grained models which better emulate real-world constraints. This thesis makes new developments in two such models.

First, we consider the *noisy primitives* model. Here, each comparison, such as "is $a < b$?", independently returns the wrong answer with probability $p < 1/2$. This model has applications in quantum and distributed computing. We extend the model to various geometric problems with Boolean geometric primitive comparisons such as "is point p above line L?". We give optimal sequential and parallel algorithms that solve these geometric problems using new techniques in random walks and parallel divide-and-conquer.

Second, we consider *in-place* models of computation, in which memory usage is restricted. We give the first sorting algorithm that is optimal in terms of the run-entropy of the input and only uses $O(1)$ additional memory. We also define a new model of memory-efficient parallelism called *Strict Synchronous Parallel In-Place* and give efficient parallel algorithms that require no auxiliary shared memory. We introduce novel techniques to ensure that our algorithms in this model remain efficient and in-place at all processor counts.

Chapter 1

Introduction

1.1 Models of Computation

Models of computation are foundational to the study of theoretical computer science. They describe abstract machines with precise rules for what computations are allowed and how data is manipulated. This allows us to study algorithms in a machine-independent way, without needing to accommodate numerous variations in architectures or implementation decisions that have developed in the construction of computer hardware. Models of computation gave theoretical computer scientists a common framework to compare the performance of computer programs analytically. They have also enabled investigations into the difficulty of problems themselves, the field of computational complexity.

There is rich study in the fields of computability theory, the design of different models of computation, and the relationship between models and formal languages (see, e.g., Savage [127] and Fernández [62]). However, this thesis begins with the simple observation that these models are fundamentally *idealized* views of computing. They make assumptions about the

capabilities of computers that are not true in general, leading to the design of algorithms that are efficient in theory but may fail to perform well in practice.

To illustrate this, let us consider the *random-access machine* (RAM) model of computation, a common model used to analyze the complexity of algorithms (see also, e.g., [127, Chapter 3.4]). In this model of computation, there exists a CPU that executes instructions and an unbounded amount of memory which can be read from and written to. Each memory location, or *register*, stores a single natural number. In addition, there is a small designated set of registers that the CPU is allowed to perform operations on. At each time step, the CPU is allowed to read from or write to a register or perform instructions on its own set of registers. The allowed instruction set varies between definitions, but most include operations such as copy, increment, decrement, jump, and halt.

This model has a natural appeal because it mimics the way actual CPUs work. However, the RAM model makes simplifying assumptions that do not reflect the way real machines operate. In particular, we focus on two limitations that motivate the works in this thesis. First, the model always assumes all operations are performed correctly and without error. This fails to hold true under certain hardware architectures, such as quantum computers. Also, there exist distributed algorithms that apply randomness to truncate messages, adding noise to primitive operations. Second, the model assumes fast access to an unbounded store of memory. Needless to say, real computers do not have infinite memory. Indeed, in embedded computing environments, such as the computers that operate sensors and smart devices, available memory can be extremely limited.

These issues motivate fine-grained models of computing that emulate the limitations of physical devices. The noisy primitives and in-place models respectively address the above concerns and are the main subjects of our investigation. We now describe them in detail.

1.2 The Noisy Primitives Model

In 1961, Rényi [125] introduced a binary search problem where comparisons between two values of the form "is $a < b$?" return the wrong answer independently with probability $p < 1/2$; see also, e.g., Pelc [116, 117]. These works established the study of the noisy primitives model, in which certain computations are faulty with some random error. We distinguish this field of study from issues that arise, for example, from round-off or measurement errors. Instead, the motivation comes from potential applications involving quantum computing, where a quantum computer is used to answer primitive queries, which return an incorrect answer with a fixed probability at most $p < 1/2$; see, e.g., [8, 90, 97]. A second motivation for this noise model comes from communication complexity, in which hash functions can be used to reduce communication costs in distributed algorithms at the expense of introducing error. Viola models this behavior with noisy primitives to efficiently construct higher-level, fault-tolerant algorithms [136].

A simple observation, made for sorting by Feige, Raghavan, Peleg, and Upfal [61], is that we can use any polynomial-time algorithm based on correct primitives by repeating each primitive operation $O(\log n)$ times and taking the majority answer as the result (in this work, all logs have base 2 unless stated otherwise). We call this the *trivial repetition strategy*, and it is easily proven by applying the parameterized Chernoff bounds of Dillencourt, Goodrich, and Mitzenmacher [51].

Theorem 1.1 (Theorem 6 of [51]). *Let $X_1, X_2, \ldots, X_m$ be independent random variables taking values in $\{0, 1\}$. Let $X = \sum_{i=1}^{m} X_i$ and let $\mu = E[X]$ denote X's expected value. Then*

$\Pr(X > R) < 2^{-xR}$, *for $x > 0$ and $R \geq (2^x e - 1)\mu$.*

For simplicity, we assume that the probability of failure $p < 1/10$. Also note that, in this work, an event succeeds "with high probability" (w.h.p.) if there exists some constant $c \geq 1$ such that the event succeeds with probability $\geq 1 - 1/n^c$.

Theorem 1.2 (Trivial repetition strategy)**.** *If each comparison independently and randomly fails with fixed, constant probability $p < 1/10$ and we repeat a comparison $O(\log n)$ times, the majority result is the correct result with high probability in n.*

Proof. We will apply the above theorem to show that, if a primitive comparison is repeated $\beta \log n$ times for some constant $\beta > 0$, fewer than $\beta \log n/2$ will report the incorrect result with high probability in n.

Let $m = \beta \log n$ and let each X_i take a 1 if repeated primitive i returns the incorrect answer and a 0 otherwise. Then $\mu < \beta \log n/10$ due to our choice of $p < 1/10$. Set $x = 1$ and $R = \beta \log n/2 > 5\mu > (2^x e - 1)\mu$. Then the chance that a majority of the repetitions report the incorrect result is $\Pr(X > R) < 2^{-xR} = 2^{-\beta \log n/2} = n^{-\beta/2}$. For $\beta > 2$, we choose the correct result w.h.p. □

By a union bound over all polynomially-many invocations of the trivial repetition strategy, the entire algorithm succeeds with high probability in n, assuming an appropriate choice of β. However, this increases running time by a $\log n$ factor. Much of the work in this field of study goes towards developing algorithms that avoid this log-factor penalty (see, e.g., [79, 116, 117, 125, 137]).

To improve the above assumption from $p < 1/10$ to $p < 1/2$, we now show that a similar strategy applied a constant amount of times can decrease the failure probability of a comparison by a constant.

Theorem 1.3. *If each comparison independently and randomly fails with fixed, constant probability $p < 1/2$, there exists a constant f such that, if we repeat the comparison f times, the majority result is correct with probability $1 - p'$, for some fixed choice of $p' < p$.*

Proof. We apply Chebyshev's inequality, which states that, for a random variable X with finite variance σ^2 and mean μ, the following holds.

$$\Pr[|X - \mu| \geq a] \leq \frac{\sigma^2}{a^2}$$

Like before, X represents the sum of f independent indicator random variables that take a 1 with a probability p and a 0 otherwise. Then $\sigma^2 = fp(1-p)$ and $\mu = fp$.

Our failure case occurs if a majority of the indicator random variables take a 1, so $a = f/2 - \mu = f(1/2 - p)$. Then we get the following.

$$\Pr[|X - fp| \geq f(1/2 - p)] \leq \frac{fp(1-p)}{(f(1/2-p))^2}$$

Lastly, we set the right-hand side equal to p' and solve for f. This gives $f = \frac{p(1-p)}{p'(1/2-p)^2}$. Since p and p' are fixed constants, f is also constant. □

Then if $1/10 \leq p < 1/2$, we select some $p' < 1/10$, repeat each comparison $\frac{p(1-p)}{p'(1/2-p)^2}$ times, and choose the majority. This reduces the failure probability to p', at which point we can apply Theorem 1.2.

We note that manipulating and comparing other data, such as pointers or metadata, are not noisy operations. This is true even if the metadata was constructed using noisy comparisons. For example, if we sort a list of elements under noisy comparisons and assign each element a rank from 1 to n, further comparisons on ranks are not noisy.

In Chapters 2 and 3, we generalize this model to apply to various fundamental geometric problems under noisy Boolean geometric primitive comparisons. We give several optimal sequential and parallel algorithms that solve these geometric problems.

1.3 In-Place Algorithms

Embedded systems are designed for a specific function within a larger device, and they comprise a processor and a limited amount of memory. They are found in diverse applications such as household appliances, vehicles, and medical equipment. Unlike with general-purpose computers, it is often difficult to extend the limited memory in embedded systems. Further, embedded systems that utilize non-volatile main memory (NVMM) are additionally restricted because these memories require routine costly processor cache flush operations to maintain data persistence [142]. A smaller memory footprint would reduce the cost of such operations. As a result, embedded systems are a natural application domain for the field of *strictly in-place* algorithms. This field of study concerns the design and analysis of efficient algorithms that allocate only $O(1)$ additional words of memory besides that used for the input. The need for efficiency is particularly important for embedded systems, since they also often have real-time performance requirements. Much prior work has investigated in-place algorithms for sorting [13, 55, 57, 88, 138], computational geometry [29, 30, 32, 33, 38, 39], graph algorithms [20, 35, 80, 94], and other fields.

In parallel computing, there is also a strong need for space-efficient algorithms. Gu, Obeya, and Shun show that the purchase and rental price of server infrastructure is dominated by memory configuration [80]. Furthermore, memory bandwidth/latency, cache misses, and page faults all present larger bottlenecks to performance and scalability if shared memory usage is high. Many classic parallel algorithms, despite having excellent performance in theory, require $\Omega(n)$ auxiliary memory (see, e.g., [9, 23, 25, 26, 74, 141]). As a result, they scale poorly in practice.

In response, researchers have developed *Parallel In-Place* (PIP) algorithms [80, 81, 82, 102, 114, 143], which restrict auxiliary space usage. Recent work by Gu, Obeya, and Shun [80] synthesizes past efforts and introduces two models for parallel in-place processing: Strong PIP and Relaxed PIP. A shared goal of these models is to strike a balance between parallelism

and space usage by requiring PIP algorithms to scale down to space-efficient sequential algorithms.

In Chapter 4, we develop the first in-place sequential sorting algorithm that has a runtime sensitive to the run-based entropy of the input array. In Chapter 5, we observe that the above models of parallel in-place provide strong in-place guarantees only when the number of processors is unbounded or when the algorithm is run sequentially. No in-place guarantees are made at intermediate processor counts. To address this gap, we propose a new and stronger model for memory-efficient parallelism called *Synchronous Strict PIP*. We introduce new techniques to provide several efficient algorithms in this model.

1.4 Preliminaries

1.4.1 Parallel Algorithms

In this work, we use the PRAM (Parallel Random Access Machine) model [91, 96, 122]. This parallel model assumes that an algorithm has access to unbounded shared memory which is accessed concurrently at each step of computation. Parallel In-place variants restrict the amount of shared memory available. There are variants of the PRAM that specify whether or how processors are allowed to access the same memory location concurrently. They are respectively the Exclusive Read, Exclusive Write (EREW) PRAM, which prohibits any concurrent access to the same memory location, the Concurrent Read, Exclusive Write (CREW) PRAM, where processors can read from shared memory concurrently but cannot write to the same location concurrently, and the Concurrent Read, Concurrent Write (CRCW) PRAM, which allows for both concurrent reads and writes to the same memory location. Furthermore, in the priority-write CRCW PRAM version, write conflicts are decided by choosing the largest value written and, in the arbitrary CRCW PRAM version, write conflicts are decided arbitrarily.

Another model of parallel computing mentioned in Chapters 3 and 5 is the binary fork-join model. In this model, parallel processors operate asynchronously. Parallelism is created via "fork" operations, in which a processor branches off to work on multiple tasks simultaneously. Subsequent "join" operations return processors back to sequential execution.

We measure the performance of parallel algorithms in terms of *work*, which is the total number of operations performed, and *span*, which is the maximum number of operations performed by a single processor if we execute the algorithm without bounding the number of processors. This is motivated by Brent's theorem [31], which states that an algorithm with input size n, work $W(n)$, and span $T(n)$ can be run on p processors in time $O(W(n)/p+T(n))$.

In Chapters 3 and 5 we extend various known parallel algorithms, which we review now.

Parallel Prefix. In the prefix problem, we are given a binary associative operator $\oplus$ and an input array $A = [x_1, x_2, \ldots, x_n]$ of n values. Our goal is to compute the following array.

$$P = \left[0, x_1, x_1 \oplus x_2, x_1 \oplus x_2 \oplus x_3, \ldots, \bigoplus_{i=1}^{n-1} x_i\right]$$

The standard work-optimal parallel algorithm works as follows. We visualize each entry of A as leaves of a binary tree. In an "up-sweep," we compute intermediate sums to fill in the internal nodes of this implicit binary tree. Each internal node applies the operator $\oplus$ to the values held by its children. In a subsequent "down-sweep," these intermediate sums are carefully augmented to populate each element of the completed prefix array. Both phases can be implemented on the EREW PRAM in $O(\log n)$ span and $O(n)$ work. See, e.g., [91, Chapter 2.1.2] for more details.

Parallel Packing. In the packing problem, we are given an array of 1s and 0s. Our goal is to move all 1s to the beginning of the array and move all 0s to the end of the array. A

variant of this problem forms an important subprocess of parallel QuickSort. We can solve it using the algorithm of Zhang and Rao [141]. Perform a prefix sum on the array. The result of this operation assigns all 1s a unique index for them to move to that packs them to the left of the array. To find locations of the 0s, simply take the original array, swap 1s and 0s and perform another prefix operation. Using the result of this operation as well as the total number of 1s, we can find unique locations for each 0 that pack them all to the right of the array. Once this is done, we assign a processor to each element, and move them to their correct location. In total, runtime is dominated by the prefix operations, which take $O(\log n)$ span and $O(n)$ work for the EREW PRAM.

Deterministic Reservations. Deterministic reservations is a design paradigm in which parallel algorithms hew closely to their sequential counterparts [22, 24, 80, 130, 131, 132]. The basic idea is to retain the logic of the sequential algorithm, but have each processor attempt to do each step in parallel. In most algorithms, this will produce conflicts as processors attempt to do operations concurrently that must be done in a certain order. To handle such conflicts, algorithms in this framework use *reservations*, wherein processors write some data related to the operation they intend to do. Operations in conflict may attempt to write to the same place. A processor whose reservation is preserved after each step is permitted to do its operation, ensuring that no conflicting operations are committed at the same time.

The problems which we investigate using this design paradigm are respectively list contraction, tree contraction, and random permutation. In the problem of list contraction, we are given a linked list of n nodes and must determine each node's place in this list. The problem of tree contraction is a natural generalization in which the elements are stored in a tree. In the problem of random permutation, we are given an array of n values which we must uniformly permute.

SampleSort. We also consider SampleSort, a randomized optimal parallel sorting algorithm. The algorithm can be thought of as a generalization of QuickSort. Given an array of n values, rather than a single pivot, we randomly sample several, typically $n^{1/3} - 1$. We sort these pivots via a brute-force method in $O(\log n)$ span and $O(n)$ work. This defines $n^{1/3}$ "buckets," i.e. ranges of key values that lie in between the pivots.

With the remaining elements, we perform binary searches in parallel on the sorted list of pivots to determine the corresponding bucket. Observe that, by properties of random sampling, each element has an equal probability of landing in each bucket. Hence, we can apply a balls-in-bins argument to bound the size of each bucket.

Lemma 1.4. *If n balls are tossed independently and uniformly into $n^{1/b}$ bins for $b > 1$, each bin will contain at most $5n^{1-1/b}$ balls with probability greater than $1 - (2^{-5n^{1-1/b}} \times n^{1/b})$.*

Proof. Let X_i be a 0-1 indicator random variable that takes 1 iff ball i is thrown into the first bin and a 0 otherwise. Then $X = \sum_{i=1}^{n} X_i$ is a random variable for the number of balls thrown in the first bin. $E[X_i] = 1/n^{1/b}$, then $\mu = E[X] = n/n^{1/b} = n^{1-1/b}$.

We apply Theorem 1.1 to bound the size of X. Setting $x = 1$ and $R = 5\mu > (2^1 e - 1)\mu$, we have

$$\Pr(X > 5n^{1-1/b}) = \Pr(X > R)$$

$$< 2^{-xR} = 2^{-5n^{1-1/b}}.$$

The proof follows from a union bound over all bins. □

It remains to distribute each remaining element into the appropriate bucket, which reduces to an application of parallel integer sorting. This can be performed on the EREW PRAM in $O(\log n)$ span and $O(n \log n)$ work (see, e.g., JáJá [91, Chapter 9.6.4]).

We also consider an alternate algorithm described by Blelloch, Fineman, Gu, and Sun which can be implemented on the CRCW PRAM [23]. First, allocate arrays of size $10n^{2/3}$ for each bucket, i.e., twice the max size. Then perform the following randomized reservations procedure. Each processor takes an element and attempts to write it to its respective bucket by selecting a uniformly random location in the corresponding array. If that location already has an element written to it or another write to the same location took precedence, the processor tries again in the following round. Assuming that each bucket will contain no more than $5n^{2/3}$ elements, the arrays will be at least half-empty by the end of the algorithm. Hence, each processor has at least a $1/2$ chance of succeeding in each round, and so the analysis reduces to the properties of flipping fair coins, where a heads indicates a processor successfully writing their element. The following lemmas bound the span and work of this procedure to $O(\log n)$ and $O(n)$ w.h.p. in n.

Lemma 1.5. *Consider n agents that are each independently flipping a fair coin until they see a heads for the first time. The last agent to find a heads will have flipped their coin $O(\log n)$ times w.h.p. in n.*

Proof. The probability that an agent makes more than $\beta \log n$ flips to find their first heads is $< 2^{-\beta \log n} = n^{-\beta}$. The result follows from a union bound over all agents, for any $\beta > 2$. □

Lemma 1.6. *If $8n$ fair coins are flipped, at least n return heads with probability $1 - 2^{-2n}$.*

Proof. Let X_i be a 0-1 indicator random variable that takes 1 iff coin i returns a heads and a 0 otherwise. Then $X = \sum_{i=1}^{8n} X_i$ is a random variable for the number of heads and $\mu = 4n$.

Theorem 11, Equation (23) of Dillencourt, Goodrich, and Mitzenmacher [51] states

$$\Pr(X < (1-\delta)\mu) < 2^{-\mu/2}, \text{ for } 1 > \delta \geq 0.7064.$$

Setting $\delta = 3/4$, we have $\Pr(X < n) < 2^{-2n}$. □

Once all elements have been distributed to their appropriate bin, we recursively solve each subproblem. To analyze this algorithm, we first observe that Lemma 1.4 only returns a high probability bound for subproblems of size $\Omega(\log^c n)$ for some constant $c > 5/2$. Hence, when subproblems are sufficiently small, we switch to any deterministic parallel algorithm with $O(\text{polylog}\, n)$ span and $O(n\, \text{polylog}\, n)$ work. The runtime associated with these small subproblems will be no more than $O(\log n)$ span and $O(n \log n)$ work.

Observe that each step of the algorithm takes at most $O(\log n)$ span and $O(n \log n)$ work respectively w.h.p. in the current problem size, n. Since subproblem sizes decrease polynomially at each level of recursion, it has been shown that the total span of this algorithm is at most $O(\log n)$ w.h.p. (see, e.g., Blelloch *et al.* [23, Theorem 4.2]). Since the algorithm requires no more than n processors at any step, an $O(n \log n)$ work bound w.h.p. naturally follows.

1.4.2 Computational Geometry

Computational geometry concerns the design of efficient algorithms which solve geometric problems. We will now review the problems and algorithmic techniques in sequential computational geometry necessary to understand the contributions of this thesis. In the following subsection, we review parallel algorithms that compute convex hulls, which we extend in Chapters 3 and 5.

General Position. General position assumptions are a set of simplifying assumptions applied to a finite set of input points in order to avoid degenerate configurations. The assumptions made throughout this thesis are as follows. We assume no three points are collinear and no four points are cocircular. In the case of 3D convex hulls, we assume no

four points are coplanar. In the case of trapezoidal decompositions, we assume that no two distinct segment endpoints share the same x-coordinate.

With careful analysis, these assumptions may be lifted (see, e.g., de Berg, Cheong, Kreveld, and Overmars [46, Chapters 6, 9, and 11]). There are also perturbation methods which automatically eliminate input degeneracies, such as the construction of Mehlhorn, Osbild, and Sagraloff [110]. Nevertheless, we make these assumptions in this work for ease of expression.

Random Incremental Construction. The random incremental construction (RIC) is a general algorithm design technique often used to solve geometric problems. It was first developed by Clarkson and Shor [42]. See also Mulmuley [112] for a general review of randomness in geometric algorithms.

At the most basic level, RICs do the following: (1) uniformly permute the input ordering and (2) repeatedly insert objects using this random ordering, maintaining an intermediate structure at each step. To illustrate this, consider the problem of sorting n distinct values. In a random incremental construction, we randomly permute the n values, repeatedly insert them into a binary search tree, and afterwards perform an in-order traversal to retrieve the sorted ordering.

Without randomly permuting the input, existing sortedness may disrupt the balance of the search tree. Indeed, if the input array were already in sorted order, the tree would devolve into a linked list and the repeated insertions would take $O(n^2)$ time. Fortunately, randomness destroys this pre-existing order and finds an input sequence that balances the tree automatically. Devroye and Reed have shown that the height of such a random binary search tree is $O(\log n)$ with high probability in n [50], so performing all insertions takes $O(n \log n)$ time with high probability.

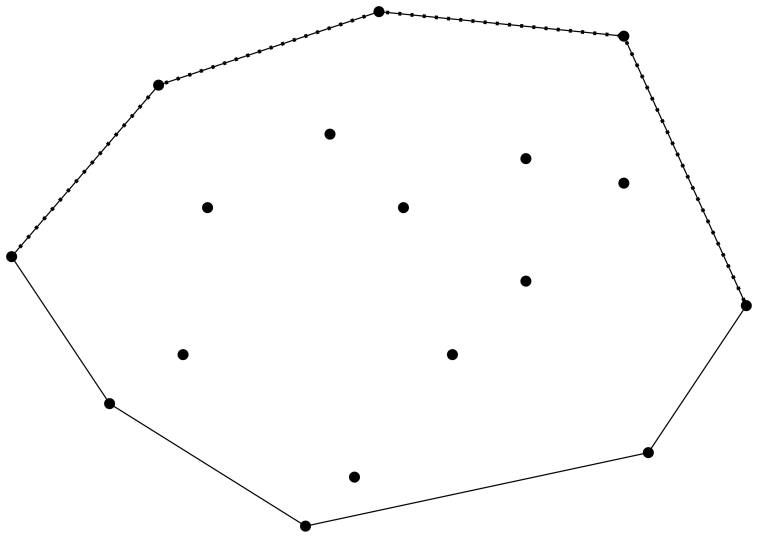

Figure 1.1: The convex hull of a point set in 2D. The upper hull has been indicated with dotted lines.

This algorithmic paradigm has a natural extension to sequential geometric algorithms, in which we are computing a structure composed of the combination of a set of geometric objects. It also forms an important part of our contribution in Chapter 2. We review non-noisy random incremental constructions below.

2D Convex Hulls. Consider a discrete set of points P in Euclidean space. The convex hull of P is defined as the smallest convex set that contains all points in P. Recall that a convex set is any set S such that, for all points $u, v \in S$, the line segment $\overline{uv}$ is totally contained in S. If P is a set of points in the plane, its convex hull forms a convex polygon whose vertices are points in P. Each point in P that lies on the convex hull is an extremal point in some direction. In general, convex hulls of points in $\mathbb{R}^d$ are convex polytopes in $\mathbb{R}^d$. A common analogy to visualize the convex hull in two dimensions is to consider the shape of a rubber band stretched around the point set P. See also, Figure 1.1.

In the plane, there are a variety of algorithms that compute the boundary of the convex hull in $O(n \log n)$ time, where n is the number of points contained in P. A typical algorithm design paradigm is to compute the "upper" and "lower" hulls of P individually, where the

Algorithm 1 Computing an Upper Hull via Graham Scan [78]

```
Sort points by their x-coordinate and denote the sorted ordering p_1, ..., p_n.
Push p_1 and p_2 onto stack S
for i ← p_3, ..., p_n do                          ▷ t indicates the top index of S
    while |S ≥ 2| and S[t − 1], S[t], p_i make a left-hand turn do
        Pop S
    Push p_i onto S
```

upper hull can be defined as the chain of hull points in clockwise ordering from the leftmost to the rightmost point. The lower hull is defined symmetrically, and in general, whenever an algorithm is defined that computes an upper hull, a symmetric algorithm computes the lower hull.

The 2D convex algorithm we consider in Chapter 2 is Graham scan [78]. It computes the upper convex hull and is described in Algorithm 1.[1] We assume that no two points share the same x coordinate for simplicity. When sweeping the points, we perform repeated *orientation tests* to determine which suffix of our current convex hull to remove after considering each new point p_i. An orientation test on points qrp_i determines whether they make a "right-hand turn" or a "left-hand turn," i.e., what side p_i is with respect to the ray $\overrightarrow{qr}$. When walking from left to right on the upper hull, all consecutive points make right-hand turns. Thus, if we find a left-hand turn, we know that r, the middle point, must be removed. Sorting the points takes $O(n \log n)$ time. Since each point is added and removed from the stack at most once, the remaining work is $O(n)$.

There exist algorithms that can compute the convex hull in time $O(n \log h)$, where h is the number of points of P that are on the boundary of its convex hull, meaning the algorithm could take as little as $O(n)$ time if h is small [3, 36, 76, 99]. However, as we observe in Chapter 2, computing convex hulls in the noisy primitives setting must take time $\Omega(n \log n)$, hence these "output-sensitive" algorithms are not of use to us.

[1] Note that the original algorithm sorts the points radially and computes the entire convex hull. We describe this version for simplicity.

3D Convex Hulls. We now review a random incremental construction originally described by Mulmuley [112] for Voronoi diagrams, and later adapted to 3D convex hulls. As with our sorting RIC above, we begin by randomly permuting the points and then inserting them one-by-one. In the case of a 3D convex hull, we take the first four points and compute an initial tetrahedron by brute-force in constant time. When we insert each new point, p_i, we use *visibility queries* to determine which facets of the current hull p_i "sees." These are the facets that will be removed. New facets are added that join p_i to the boundary of the gap created by the deleted facets.

Rather than a binary search tree, our search structure to determine the facets to remove is as follows. Each node represents a facet of the hull. Root nodes represent each facet of the initial tetrahedron. Each leaf of the structure represents a facet of the current hull. Upon inserting a new point, we delete a set of facets X and insert a new set of facets Y. In the DAG, this is represented by initializing a new node for every $f \in Y$. Additionally, every node representing each $f \in X$ has a pointer to each node representing every $f \in Y$. Observe that the search structure is a directed acyclic graph (DAG) which encodes the changes in the hull that occur at each step. We call it the *history DAG*.

To obtain faster search times, we also organize the new nodes representing Y into a *radial search structure*. Let p be the newly added point. As will be clear soon, we store the new edges incident to p in the radial search structure rather than the faces. These edges are ordered cyclically around p. Convert this cyclic ordering into a linear ordering by arbitrarily picking the "least clockwise" edge and organize them into a search structure (e.g. sorted array, binary search tree, skip list, etc.). This structure is then embedded into the history DAG along with the nodes for Y.

Now we show how to query this structure. After we create the initial tetrahedron, we choose a coordinate system such that its origin lies in the tetrahedron (thus the origin will be located within the current hull at every step of the RIC). Imagine shooting a ray from the origin to

our query point q. We start our query at the node representing the facet of the tetrahedron which the ray pierces. Likewise, we descend to the child that represents the next facet whose interior our ray passes through, and so on until we reach a leaf node. By general position, the ray only passes through the interior of a facet, never the edge or at a vertex. In this way, we have a query that designates a unique path down the history DAG.

To make this process more efficient, we rely on the radial search structures. Say we are at some node v of the history DAG and are querying a radial search structure to find the unique child to descend to. Let p be the point that created the edges in the current radial search structure and let line $\overrightarrow{Op}$ be a ray that starts at the origin and passes through p. From the edges incident to p, we define a system of half-spaces, all of which are bounded by $\overrightarrow{Op}$ (think of them as like wedges of an orange surrounding $\overrightarrow{Op}$ where each plane that slices the wedges is drawn between an edge incident to p and $\overrightarrow{Op}$). The goal of this search is to determine which face our query ray $\overrightarrow{Oq}$ pierces. Each node of the radial search structure represents one of the half-spaces in the system. An orientation test determines which side of the half-plane the query q is, and so determines which direction to recurse in the tree. At the end of the search, we will have found two adjacent half-spaces that bound q on either side. Because the edges incident to p defined these half-spaces, this corresponds exactly to the face that the ray $\overrightarrow{Oq}$ pierces. Thus, we have found the unique child of v that we were looking for.

We use the history DAG to determine the set of facets X_i that conflict with our new point q each iteration. Notice that the above query only finds one such facet. However, given a single conflicting facet, we can simply walk around the hull and perform visibility tests on adjacent facets to recover the set of conflicting facets in $O(|X_i|)$ time in the non-noisy setting. Let ℓ_i be the length of the path that q took in the history DAG to find a conflicting facet. Then it takes $O(\ell_i + |X_i|)$ time to update the history DAG. An analysis by Mulmuley [112] shows that $\sum_{i=1}^{n} \ell_i = O(n \log n)$ and $\sum_{i=1}^{n} |X_i| = O(n)$ in expectation. Then the total runtime of the algorithm in the non-noisy setting is $O(n \log n)$ in expectation.

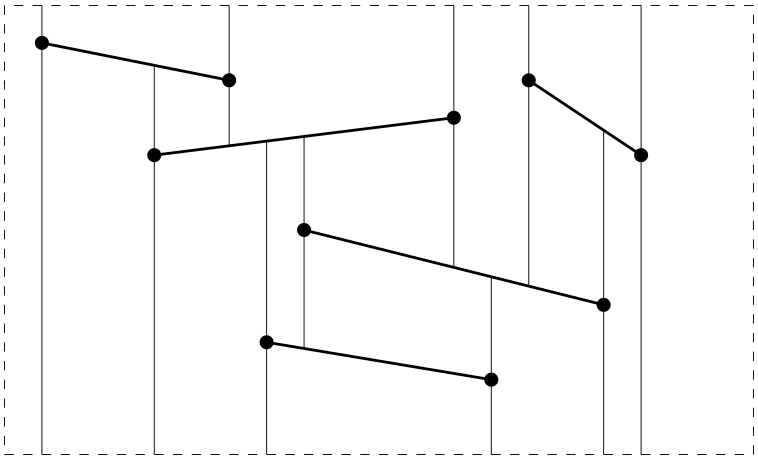

Figure 1.2: A trapezoidal decomposition composed of five line segments. Each segment endpoint has vertical "walls" extended from it, which define the trapezoids.

Trapezoidal Decomposition. An important class of problems in geometry involve *ray-shooting*. Specifically, consider the vertical ray-shooting problem, in which we are given a query point q, and a set of n line segments S in the plane. We want to determine which segments lie directly above and below q, i.e., which segments are hit first by vertical rays emanating from q. We can solve this problem by defining a planar decomposition as follows. Draw vertical "walls" extending from each endpoint of each segment which stop when they touch another segment. See Figure 1.2. This splits the plane into $O(n)$ trapezoids, thus we call this planar subdivision the *trapezoidal decomposition*. If we associate each trapezoid with the segments that bound it from above and below, then determining which trapezoid contains q answers the vertical ray-shooting problem.

We review a random incremental construction which computes the trapezoidal decomposition and an associated point-location structure in expected $O(n \log n)$ time (see also, e.g., [46, Chapter 6]), assuming that no pair of segments cross. The algorithm has a similar structure to above. We begin by randomly permuting the segments and inserting them one-by-one. Each node of our history DAG represents a trapezoid. Leaves are trapezoids of the current decomposition and internal nodes are trapezoids that have been destroyed in previous iterations.

At each step, we query the history DAG to determine which leaf nodes contain the two endpoints of the new segment we wish to insert. Querying the structure with some point q is straightforward. Start at the root, which is a large trapezoid representing a bounding box of the algorithm, and choose the unique child node whose trapezoid contains q. Repeat until a leaf node is found.

Similarly to 3D hulls, once we have navigated through the DAG, we perform a fix-up step to update the decomposition. We walk from one segment endpoint to the other destroying and creating new trapezoids as needed, which takes time linear in the number of destroyed trapezoids. Via a standard backwards analysis, total point-location and fix-up cost are respectively $O(n \log n)$ and $O(n)$ in expectation [46, Chapter 6.2].

In Chapter 2, we also utilize a well-known plane-sweep algorithm by Bentley and Ottmann that finds the trapezoidal decomposition in $O((n+k) \log n)$ time when there exist k crossings between the input segments [16].

Voronoi Diagrams and Delaunay Triangulations. Consider a set of points, S, in the plane. We call the points in S *sites*. For each p in S, let its *Voronoi cell* $V(p)$ be defined as the following subset of the plane.

$$V(p) = \{q \in \mathbb{R}^2 : d(q,p) \leq d(q,p'), \forall p' \in S\}.$$

The *Voronoi diagram* $\mathcal{V}(S)$ is the cell complex of the plane composed of all $V(p)$ for all $p \in S$. The *Delaunay triangulation*, $DT(P)$, is a straight-line embedding of the dual graph of $\mathcal{V}(S)$.

The Delaunay triangulation can also be characterized independently of Voronoi diagrams. Consider a triangle of sites $\triangle pqr$ in S. We say that $\triangle pqr$ is *Delaunay* iff its circumscribing

circle contains no other sites in S. Then a Delaunay triangulation is a triangulation of S such that all triangles are Delaunay. Recall that a triangulation of S is a *maximal planar subdivision* of S, i.e., a subdivision formed by adding edges between pairs of sites in S. It is maximal in the sense that no more edges can be added without violating planarity. Assuming general position, this triangulation is unique.

We now review an RIC that computes the Delaunay triangulation of n sites in general position in expected $O(n \log n)$ time. As before, we randomly permute the set of sites and insert them sequentially. Each node of the history DAG represents a triangle, with leaves representing the triangles in the current Delaunay triangulation. We query the history DAG in a similar way to our trapezoidal decomposition history DAG. Once we determine which triangle, Δ, of our current decomposition contains our new site, p, we insert p by drawing edges between p and the vertices of Δ. This splits Δ into three triangles, but these triangles may not be Delaunay. As a result, we perform a local edge-flipping process as a fix-up step. It has been shown that this flipping process terminates at the Delaunay triangulation. Furthermore, the total runtime for point-location and edge-flipping is respectively $O(n \log n)$ and $O(n)$ in expectation [46, Chapter 9.4].

1.4.3 Parallel Convex Hull Algorithms

2D Convex Hull Algorithm for CREW PRAM. We consider the $O(\log n)$-span, $O(n \log n)$-work algorithm that computes the upper hull of a point set, independently discovered by Aggarwal, Chazelle, Guibas, Ó'Dúnlaing and by Goodrich and Atallah [5, 74].

We first sort all n points by their x-coordinate in $O(\log n)$ span and $O(n \log n)$ work, e.g., using Cole's merge sort [44]. Then we partition them into $\sqrt{n}$ groups of $\sqrt{n}$ points. This partition is done by their index, e.g., points 1 through $\sqrt{n}$ in the sorted order belong in group 1, and so on. We recursively solve each group in parallel, and each group ultimately returns with an upper hull of their $\sqrt{n}$ points.

Once we have $\sqrt{n}$ upper hulls, we can combine them in parallel. We first compute an upper tangent for every pair of upper hulls using the double binary search of Overmars and van Leeuwen [115].

We next determine the contribution of each upper hull U_j to the combined upper hull of all n points. To do this, we compute V_j, the tangent of smallest slope between U_j and some other U_i where $i < j$. Likewise, we compute W_j, the tangent of largest slope between U_j and some other U_k where $k > j$. Let v_j and w_j be the corresponding tangent points on U_j. If the angle between V_j and W_j around their intersection point pointing upward is $\geq 180°$, then points from v_j to w_j on U_j are on the combined upper hull. Otherwise, U_j contributes no points to the upper hull. The authors' rationale for choosing V_j and W_j is that U_i and U_k are the two hulls that are best able to "cover up" U_j such that it does not contribute to the combined upper hull. Their respective tangents with U_j indicate that they are the "highest" hulls on each side of U_j. Finally, a parallel prefix operation is used to combine contributions from each U_j into the combined upper hull.

Now we discuss the span and work performed at each level of recursion. There are $\binom{\sqrt{n}}{2} = O(n)$ pairs of upper hulls. The double binary search method of Overmars and van Leeuwen [115] takes $O(\log n)$ time sequentially. If we perform each tangent computation in parallel, this takes $O(\log n)$ span and $O(n \log n)$ work. To compute W_j, we perform a max-find operation on the $O(\sqrt{n})$ tangents that touch U_j. Doing this in parallel over all U_j takes $O(\log n)$ span and $O(n \log n)$ work. The same cost applies to computing V_j. It then takes constant span and $O(\sqrt{n})$ work to determine the convex chain each U_j contributes to the combined hull. The final parallel prefix operation of the $O(\sqrt{n})$ convex chains takes $O(\log n)$ span and $O(\sqrt{n})$ work.

For $n \leq 2$, span and work are both constant as the convex hull is simply a line segment between the two points. We conclude that the total span and work for the algorithm are

given by the following two recurrences:

$$T(n) = T(\sqrt{n}) + O(\log n)$$

$$W(n) = \sqrt{n}W(\sqrt{n}) + O(n \log n)$$

With $T(2), W(2) = O(1)$. Observe that both recurrences can be bounded by a geometric series.

$$T(n) < O(\log n) \times \sum_{i=0}^{\infty} 1/2^i = O(\log n)$$

$$W(n) < O(n \log n) \times \sum_{i=0}^{\infty} 1/2^i = O(n \log n)$$

The initial sorting takes $O(\log n)$ span and $O(n \log n)$ work [44], so the algorithm takes $O(\log n)$ span and $O(n \log n)$ work overall in the CREW PRAM model [5, 74].

3D Convex Hull Algorithm for CREW PRAM. We now review the $O(\log n)$-span and $O(n \log n)$-work randomized 3D convex hull algorithm of Reif and Sen [123]. First, we note that Reif and Sen solve the dual problem of halfspace-intersection of n halfspaces in 3D. This allows them to take advantage of an earlier result of Clarkson [42], who showed that a random sample, R, from a set of halfspaces, H, can be used to divide the remaining $H \setminus R$ halfspaces into subproblems of roughly equal size, admitting a natural randomized divide-and-conquer algorithm. Specifically, these good samples divide the remaining halfspaces into groups with the following two properties: (1) the largest group has size at most $O(|H| \log |R|/|R|)$ and (2) the sum over all groups is of size $O(|H|)$. However, Clarkson only showed that this is the case with constant probability $> 1/2$.

To turn this sampling probability into a high-probability bound, Reif and Sen introduce the notion of *polling*. Rather than one sample R, they take $O(\log n)$ samples R_j and test if any

are "good" as defined by Clarkson. It would be too expensive to evaluate each R_j, so instead of sampling each from H, they first sample $n/\log^d n$ elements from H into H_j, for $d > 2$. Then they sample R_j from H_j. By shaving this polylog factor, Reif and Sen show that we can evaluate whether each R_j is a good sample for H_j in $O(\log n)$ span and $O(n \log n)$ work. Through applications of Chernoff bounds, they show that (1) at least one R_j is good with respect to H_j, and (2) R_j is a good sample for the original H as well, despite the polylog factor decrease. Both occur with high probability in n. Note that if a good sample was not found, we could simply spend the same amount of span and work to repeat the process.

Say that we have found a good sample R. In the process to determine whether R was good, Reif and Sen use a non-optimal halfspace intersection algorithm to compute the polyhedron $\mathcal{R}$ in $O(\log n)$ span and $O(n \log n)$ work. We can use this structure to define our divide-and-conquer. Observe that the halfspace intersection $\mathcal{C}$ of H can only get smaller than $\mathcal{R}$, thus the subproblems which we recursively solve must involve some partition of the interior of $\mathcal{R}$. Specifically, this is done by considering some point o assumed to be in the halfspace intersection of H. We draw lines from each vertex of $\mathcal{R}$ to o and triangulate the faces of $\mathcal{R}$ such that we have a set of at most $2|R|$ "cones", their base a triple of vertices of $\mathcal{R}$ and their apex o. Then each recursive problem would involve computing a halfspace intersection of the subset of halfspaces in $H \setminus R$ that intersect some cone C_i.

To determine the cones each halfspace in $H \setminus R$ stabs, the authors consider the dual problem. Taking the dual transform of each vertex of $\mathcal{R}$ gives us an arrangement of hyperspaces. Each cell of the arrangement corresponds to a combination of cones C_i that some halfplane in the primal could intersect. By parallelizing the multidimensional search method of Dobkin and Lipton [52], they show that, as long as R is a sufficiently small sample, $|R| \leq n^{1/8}$, we can construct a search structure in $O(\log n)$ span and $O(n \log n)$ work that allows for fast queries to return the set of cones of $\mathcal{R}$ that a hyperspace h intersects in the primal in. Note that this sample guarantees that a subproblem can be as large as $O(n^{7/8} \log n)$.

Lastly, Reif and Sen observe that, despite our sample R being "good", it is not good enough to prevent a blowup in problem size as we go down recursive levels. This is because halfspaces may intersect with multiple cones. If X_i is the number of halfspaces of $H \setminus R$ that intersect cone C_i, then Clarkson showed that $\sum_i X_i = O(n)$ [42], meaning the total problem size may increase by a constant factor every level of recursion. There are $O(\log \log n)$ levels of recursion, so this constant factor blow-up implies total problem size may expand to $O(n \log^{O(1)} n)$, increasing work by a polylog factor. Thus, their final step before recursion involves pruning redundant halfspaces to ensure that total problem size at each level of recursion is no more than $O(n)$.

In their work, Reif and Sen classify different types of redundancies and describe a two-step approach to pruning the halfspaces of cone C_i, first by applying a 3D maxima algorithm and then by computing a simplified 3D hull they call a skeletal hull. Information from these outputs can be used to narrow down only the halfspaces that both intersect C_i and contribute a vertex to the final halfspace intersection $\mathcal{C}$ that lies in C_i.

However, a later work by Amato, Goodrich, and Ramos [9] provides a simpler way to prune. They show that we can perform a sufficient amount of pruning from information given to us by the *contours* of C_i. A contour is the result of projecting the halfspaces that intersect C_i onto the three faces of the cone incident to the apex o and then computing the 2D convex chain of the projection on each face. Given these contours, they show that a constant number of binary searches per halfspace and one sorting step per cone can be used to prune a sufficient number of halfspaces. By the fact that the total problem size among all cones is $O(n)$, we can compute contours for all cones with $O(\log n)$ span and $O(n \log n)$ work. In addition, Amato, Goodrich, and Ramos show that it takes $O(\log n)$ span and $O(n \log n)$ work to prune given these contours.

Finally, we recursively solve each subproblem defined by each cone C_i. Each subproblem returns with the subset of vertices of the halfspace intersection $\mathcal{C}$ located in that subproblem,

from which an adjacency structure can be computed by an application of sorting [123, Lemma 2.2]. Once the subproblem size is smaller than $O(\log^k n)$ for some constant k, the authors can apply a non-optimal time algorithm (e.g., the $O(\log^3 n)$-span, $O(n \log^3 n)$-work algorithm of Aggarwal, Chazelle, Guibas, Ó'Dúnlaing, and Yap [5]) to solve the problem directly without increasing total asymptotic complexity. Reif and Sen conclude that the span for their algorithm is bounded by following recurrence:

$$T(n) \leq T(n^{9/10}) + O(\log n)$$

The $n^{9/10}$ comes from Clarkson, whose analysis showed that, with a good sample, no cone has more than $O(n^{7/8} \log n) = O(n^{9/10})$ halfspaces that stab it [42]. Furthermore, we note that the span is logarithmic w.h.p. in the current subproblem size, so the analysis is slightly more involved than solving the above recurrence. Nevertheless, Reif and Sen are able to show that such a sequence of runtimes converges to $O(\log n)$ total span w.h.p. in n [123, Theorem 2.1]. To bound the algorithm's work, we simply observe that the pruning step guarantees that each level of recursion has total problem size $O(n_0)$, where n_0 is the initial problem size. Then total processor count is bounded by $O(n_0)$ at each step, and so the algorithm must have work $O(n_0 \log n_0)$ w.h.p. in n_0. We conclude that, given a set of n halfplanes in 3D, Reif and Sen's algorithm computes their halfspace intersection in $O(\log n)$ span and $O(n \log n)$ work w.h.p. for the CREW PRAM model [123].

1.4.4 Failure Sweeping

In Chapter 3, we introduce a novel generalization of the *failure sweeping* algorithm paradigm in order to develop parallel algorithms that optimally construct convex hulls in 2D and 3D while being resilient to noisy primitives.

Failure sweeping, formalized by Ghouse and Goodrich [71], is a general technique used to improve confidence bounds on algorithms that either fail or exceed a desired span or work bound with some failure probability $p(n)$, given input size n. Say that our desired bounds for span and work done at a subproblem of size n are $T(n)$ and $W(n)$. Our goal is for the entire algorithm to fail with probability at most $p(n)$, but when we recurse to sufficiently small subproblems of size s, e.g., $s = O(\log n)$, $p(s)$ may be exponentially larger than $p(n)$. If unchecked, errors in small subproblems propagate to the returned answer, meaning the algorithm can only guarantee success probability $1 - p(s)$, rather than the intended $1 - p(n)$. This is exactly the case in the noisy primitives setting. Algorithms resilient to noise run in span $T(s)$ if we wish for them to fail with probability at most $p(s)$. Attempting to have all subproblems fail with probability at most $p(n)$ would cause each subproblem to take span proportional to $T(n)$, leading to an inefficient algorithm. Failure sweeping gives us a way to improve our confidence bound without incurring this span blow-up.

Let M be the size of a parent subproblem and m the size of its child subproblems. The general failure sweeping framework introduces two algorithms: a failure sweeping algorithm that can determine whether one of M's child subproblems of size m is incorrect and a "brute-force" algorithm that can recompute an incorrect subproblem w.h.p. in M. We must sweep all of M's subproblems at every level of recursion, so failure sweeping over all subproblems must not take more than $T(M)$ span and $W(M)$ work. On the other hand, our brute-force algorithm only applies to incorrect subproblems. Thus, brute-force recomputation of a single problem of size m may exceed $W(m)$, though brute-force recomputation over all failed subproblems should not exceed $T(M)$ and $W(M)$ w.h.p. in M.

To complete the argument, one must show that not too many subproblems fail with probability $p(M)$. Specifically, if each subproblem of size m fails with probability $p(M)$, we must show that no more than q subproblems, determined in the analysis, fail with probability $1 - p(M)$. Then, we apply our brute-force algorithm to compute the q subproblems in

span $T(M)$ and total work $\leq W(M)$. This way, all subproblems succeed with probability $1 - p(M)$. Applied inductively, we "upgrade" our failure probabilities at each level of recursion without asymptotically increasing span or work. This ultimately allows the algorithm to achieve confidence bound $1 - p(n)$, where n is the initial problem size.

1.4.5 Entropy-Sensitive Sorting

In Chapter 4, we investigate the first strictly in-place sorting algorithms that are optimal with respect to the run-based entropy of the array. Here, we define the notion of run-based entropy. Consider an array $A = [x_1, x_2, \ldots, x_n]$, of elements that come from a total order. If we let $\mathcal{R} = \{R_1, R_2, \ldots, R_{\rho(A)}\}$ be a partition of A into a set of maximal increasing or decreasing *runs* (i.e., subsequences of consecutive elements), and we let r_i denote the size, $|R_i|$, of the i-th run in A (so $\sum_{i=1}^{\rho(A)} r_i = n$), then the *run-based entropy*, $\mathcal{H}(A)$, for A is defined as follows:

$$\mathcal{H}(A) = \sum_{i=1}^{\rho(A)} (r_i/n) \log(n/r_i).$$

Researchers have proven that any comparison-based sorting algorithm has a lower bound for its running time that is $\Omega(n(1 + \mathcal{H}(A)))$, and there are several sorting algorithms that run in $O(n(1 + \mathcal{H}(A)))$ time; see, e.g., [10, 15, 34, 69, 93, 113, 128, 135]. Interestingly, all of these algorithms are *stack-based natural mergesort* algorithms. As their description suggests, these algorithms maintain a stack of runs and decide when to merge them based on their lengths and/or position in order to take advantage of the input data's pre-sortedness. Two of these algorithms, TimSort and PowerSort, have become widely used in practice, e.g., as the default sorting algorithms in Python, Java, and other major programming languages.

1.5 Our Contributions

1.5.1 Computational Geometry under Noisy Primitives

In Chapters 2 and 3, we consider computational geometry algorithms in the noisy primitives model. We do this by extending the notion of noisy comparisons to that of *Boolean geometric primitives*. Geometric algorithms typically rely on one or more Boolean geometric primitive operations that are assumed to be computable in $O(1)$ time. For example, in a Delaunay triangulation algorithm, this may be determining if some point p is located in some triangle Δ; in a 2D convex hull algorithm, this may be an orientation test; etc. Here we assume that a geometric algorithm relies on primitive operations that each outputs a Boolean value and has a fixed probability $p < 1/2$ of outputting the wrong answer. As in earlier work for the sorting problem by Feige, Raghavan, Peleg, and Upfal [61], we assume non-persistent errors, in which each primitive test can be viewed as an independent weighted coin flip.

If we consider our random incremental sorting algorithm described above, observe that by replacing the binary search required for each insertion step with the optimal noise-tolerant binary search of Feige *et al.* [61], we immediately have a noise-tolerant sorting algorithm that operates in $O(n \log n)$ time w.h.p. In Chapter 2, we generalize this insight to apply to various geometric random incremental constructions using a new technique we call *path-guided pushdown random walks*. In particular, we prove the following central theorem, which supports several of our results.

Theorem 2.1. *Given an error tolerance, $\varepsilon = n^{-c}$ for $c > 0$, and a DAG, G, with a path, P, from a vertex, s, to a distinct vertex, t, the path-guided pushdown random walk in G starting from s will terminate at t with probability at least $(1 - \varepsilon)$ after $N = \Theta(|P| + \log(1/\varepsilon))$ steps, for a transition oracle, T, with error probability $p_e < 1/15$.*

In Chapter 3, we consider noise-tolerant parallel algorithms that compute convex hulls in 2D and 3D. We do this by applying the failure-sweep paradigm to existing parallel divide-

and-conquer algorithms for 2D and 3D convex hulls. Recall that in failure sweeping, we incorporate new algorithms that detect and fix incorrect subproblems using two new failure sweeping and brute-force algorithms. However, failure sweeping alone is not sufficient as subproblems in our algorithms may be too large to be recomputed by brute-force. Thus, we augment the failure sweeping paradigm with a new technique we call *size reduction*. This technique selectively applies the divide step more than once such that subproblems can now be efficiently recomputed in case they are faulty. Using this new technique, we give algorithms for 2D and 3D convex hulls that run in optimal $O(\log n)$ span and $O(n \log n)$ work w.h.p. in n for the CREW PRAM. As an additional demonstration of our new techniques, we give a much simpler optimal noisy sorting algorithm than the one recently presented by Goodrich and Jacob [75].

1.5.2 New In-Place Algorithm Design Techniques

In Chapters 4 and 5, we present new results in memory-efficient algorithms. In Chapter 4, we propose two simple paradigms that enable stack-based natural mergesorts to be in-place. We observe that the barrier to making these algorithms in-place involves eliminating the space required to maintain their stack of runs S, which may be as large as $\Omega(\log n)$ in all known stack-based mergesorts. Our first contribution is called the *walk-back* algorithm. In this algorithm, we only maintain the lengths of a constant number of runs on the stack plus at most $O(1)$ additional metadata. If we ever need to determine the length of a run that occurs deeper in the stack, we simply walk backwards down the array to recover it. We show that several stack-based mergesorts can be implemented to run efficiently and in-place using the walk-back algorithm, including PowerSort. Notably, we show that there exist input sequences in which TimSort is *not* efficient when implemented with the walk-back algorithm.

This motivates our second contribution, which we call the *jump-back algorithm*, that can also be used to implement virtually all stack-based mergesorts in-place. In this algorithm,

we only maintain the lengths of a constant number of runs on the stack plus at most $O(1)$ metadata, and we encode the lengths of runs in-place in the runs themselves using simple bit-encoding methods. If we ever need to determine the length of a run that occurs deeper in the stack, we decode its length from its bit encoding, which then allows us to jump to the beginning of the run.

In Chapter 5, we propose a new parallel in-place model called Synchronous Strict PIP. Our model is inspired by the recent work of Gu, Obeya, and Shun on Parallel In-Place models [80]. Recall that they propose the Strong and Relaxed PIP models and present several algorithms in these models. However, their models only provide strong parallel in-place guarantees when either the number of processors is unbounded or the algorithm is run sequentially (i.e., the number of processors is one). This is a restrictive form of parallelism which fails to apply, for example, in the setting of big data bulk synchronous processing frameworks, where the number of processors is fixed and much smaller than the total input size, n.

This motivates our new Synchronous Strict PIP model, which is a specialization of the PRAM model of computation. Let $1 \leq p \leq n$ be the number of processors available. A PRAM algorithm is Synchronous Strict PIP if it allocates no shared memory aside from the input, uses $\Theta(1)$ private memory per processor, and uses at most $\Theta(1)$ additional space, i.e., is strictly in-place, when run sequentially. Thus, our model provides strictly in-place parallelism at all processor counts, irrespective of the input size.

We develop several efficient parallel algorithms in this model by introducing two new techniques. First is the *IJ swap*, a family of methods to strategically swap data between shared and private memory in order to avoid allocating auxiliary memory. Second, we introduce *parallel-augmented sweep*, an insight that allows us to schedule our in-place algorithms on $p < n$ processors by interpolating between sequential in-place sweep algorithms and their parallel counterparts.

Chapter 2

Computational Geometry with Probabilistically Noisy Primitive Operations

2.1 Background

There is considerable prior work on sorting and searching with noisy comparison errors, i.e., comparisons that fail randomly and independently with some fixed probability $p < 1/2$. For example, Feige, Raghavan, Peleg, and Upfal [61] show that one can sort in $O(n \log n)$ time w.h.p. with probabilistically noisy comparisons. Dereniowski, Lukasiewicz, and Uznanski [48] study noisy binary searching, deriving time bounds for constant factors involved. A similar study has also been done by Wang, Ghaddar, Zhu, and Wang [137]. Klein, Penninger, Sohler, and Woodruff solve linear programming in 2D under a different model of primitive errors [100], in which errors persist regardless of whether primitives are recomputed.

The only prior work we are aware of on computational geometry algorithms tolerant to such non-persistent errors is work by Groz, Mallmann-Trenn, Mathieu, and Verdugo [79] on computing a d-dimensional skyline of size k and probability $1 - \delta$ in $O(nd \log(dk/\delta))$ time.

Nevertheless, considerable prior work has designed algorithms that can deal with input degeneracies, data imprecision, and arithmetic round-off errors. For example, several researchers have studied general methods for dealing with degeneracies in inputs to geometric algorithms, e.g., see [56, 140]. Researchers have designed algorithms for geometric objects with imprecise positions, e.g., see [107, 108]. In addition, significant prior work has dealt with arithmetic round-off errors and/or performing geometric primitive operations using exact arithmetic, e.g., see [63, 111]. While these prior works have made important contributions to algorithm implementation in computational geometry, they are orthogonal to the probabilistic noise model we consider in this chapter.

Emamjomeh-Zadeh, Kempe, and Singhal [58] and Dereniowski, Tiegel, Uznański, and Wolleb-Graf [49] explore a generalization of noisy binary search to graphs, where one vertex in an undirected, positively weighted graph is a target. Their algorithm identifies the target by adaptively querying vertices. A query to a node v either determines that v is the target or produces an edge out of v that lies on a shortest path from v to the target. As in our model, the response to each query is wrong independently with probability $p < 1/2$. This problem is different than the graph search we study in this chapter, however, which is better suited to applications in computational geometry. For example, in computational geometry applications, there is typically a search path, P, that needs to be traversed to a target vertex, but the search path P need not be a shortest path. Furthermore, in such applications, if one queries using a node, v, that is not on P, it may not even be possible to identify a node adjacent to v that is closer to the target vertex.

Viola [136] uses a technique similar to ours to handle errors in a communication protocol. In one problem studied by Viola, two participants with n-bit values seek to determine which of

their two values is largest. This can be done by a noisy binary search for the highest-order differing bit position. Each search step performs a noisy equality test on two prefixes of the inputs, by exchanging single-bit hash values. The result is an $O(\log n)$ bound on the randomized communication complexity of the problem. Viola uses similar protocols for other problems including testing whether the sum of participant values is above some threshold. The noisy binary search protocol used by Viola directs the participants down a decision tree, with an efficient method to test whether the protocol has navigated down the wrong path in order to backtrack. One can think of our main technical lemma as a generalization of this work to apply to any DAG.

2.1.1 Our Results

This chapter centers around our novel technique, *path-guided pushdown random walks*, described in Section 2.2. It extends noisy binary search in two ways: it can handle searches where the decision structure of comparisons is in general a DAG, not a binary tree, and it also correctly returns a non-answer in the case that the query value is not found. These two traits allow us to implement various geometric algorithms in the noisy comparison setting.

However, to apply path-guided pushdown random walks, one must design an oracle that, given a sequence of comparisons, can determine if one of them is incorrect in constant time. Because different geometric algorithms use different data structures and have different underlying geometry, we must develop a unique oracle for each one. The remainder of this chapter describes noise-tolerant implementations with optimal running times for plane-sweep algorithms, point location, convex hulls in 2D and 3D, and Delaunay triangulations. We also present a dynamic 2D hull construction that runs in $O(\log^2 n)$ time w.h.p. per operation. See Table 2.1 for a summary of our results.

Our algorithms to construct the trapezoidal decomposition, Delaunay triangulation, and 3D convex hull adapt classic randomized incremental constructions. A recent paper by

Algorithm	Runtime	Section
Trapezoidal Map	$\Theta(n \log n)^\star$	Section 2.4
Trapezoidal Map with k Crossings	$O((n+k)\log n)$	Section 2.5.2
2D Closest Points	$\Theta(n \log n)^\dagger$	Section 2.5.3
2D Convex Hull	$\Theta(n \log n)^\dagger$	Section 2.6.1
Dynamic 2D Convex Hull	$O(\log^2 n)$ per update	Section 2.6.2
3D Convex Hull	$\Theta(n \log n)^{\star\dagger}$	Section 2.6.3
Delaunay Triangulation	$\Theta(n \log n)^\star$	Section 2.7

Table 2.1: Our main results in the noisy setting. All algorithms succeed w.h.p. in n. Runtimes marked with a * are in expectation. Runtimes marked with a † are optimal despite having faster algorithms in the non-noisy setting.

Gudmundsson and Seybold [83] published at SODA 2022 claimed that such constructions produce search structures of depth $O(\log n)$ and size $O(n)$ w.h.p. in n, implying that the algorithms run in $O(n \log n)$ time w.h.p. Unfortunately, the authors have contacted us privately to say that their analysis of the size of such structures contains a bug. In this chapter, we continue to use the standard expected time analysis.

2.2 Path-Guided Pushdown Random Walks

In this section, we provide an analysis tool that we use repeatedly in this chapter and which may be of independent interest (e.g., to analyze randomized routing protocols). Specifically, we introduce *path-guided pushdown random walks*, which are related to biased random walks on a graph (e.g., see [14, 65]) and generalize the noisy binary search problem [61, 67]. See Figure 2.1 for a depiction of path-guided pushdown random walks.

A path-guided pushdown random walk is defined in terms of a directed acyclic graph (DAG), G, that has a starting vertex, s, a target vertex, t, and a path, $P = (s, v_1, v_2, \ldots, t)$, from s to t, in G (we assume $s \neq t$). We start our walk from the start vertex, $v \leftarrow s$, and we use a stack, S, which initially contains only s, and a transition oracle, $T(v)$, to determine our walk. For each vertex, v, during our walk, we consult the transition oracle, $T(v)$, which first

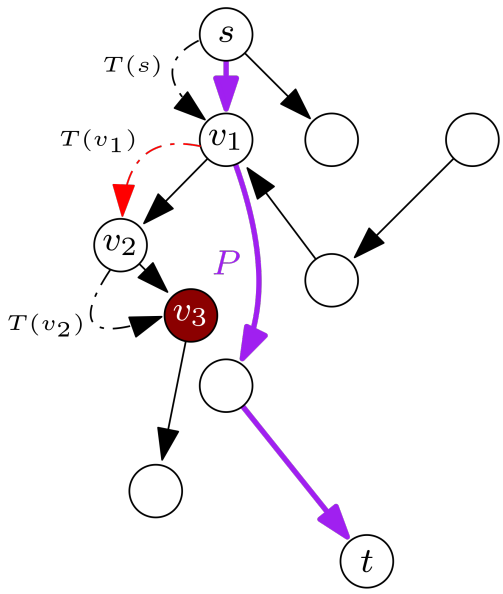

Figure 2.1: Here we show a sample execution of path-guided pushdown random walks in some DAG G. The transition oracle acts *arbitrarily* when it lies, so we may end up with the sequence shown here, a correct move prior to our current node but an incorrect move to v_2. To apply path-guided pushdown random walks, we must be able to determine whether we are on P in $O(1)$ comparisons, no matter where we are in G.

tells us whether $v \in P$ and if so, then $T(v)$ tells us the next vertex in P to visit to make progress towards t. $T(v)$ can return v, which means we should stay at v, e.g., if $v = t$.

Our model allows T to "lie." We assume a fixed error probability,[1] $p_e < 1/15$, such that T gives the correct answer with probability $1 - p_e$, independently each time we query T. With probability p_e, $T(v)$ can lie, i.e., $T(v)$ can indicate "$v \in P$" when $v \notin P$, $T(v)$ can indicate "$v \notin P$" when $v \in P$, or $T(v)$ can return a "next" vertex that is not an actual next vertex in P (including returning v itself even though $v \neq t$). Importantly, this next vertex must be an outgoing neighbor of v. This allows us to maintain the invariant that S holds an actual path in G (with repeated vertices). Our traversal step, for current vertex v, is as follows:

- If $T(v)$ indicates that $v \notin P$ (and $v \neq s$), then we set $v \leftarrow S.\mathsf{pop}()$, which may be v again, for the next iteration. This is a backtracking step.

[1]The threshold of 1/15 simplifies our proof. We can tolerate any higher error probability bounded below 1/2 via an application of Theorem 1.3.

- If $T(v)$ indicates that $v \in P$, then let w be the vertex indicated by $T(v)$ as next in P, such that $v = w$ or (v, w) is an edge in G.[2]

  - If $v = w$, then we call $S.\mathsf{push}(v)$ and repeat the iteration with this same v, since this is evidence we have reached the target, t.

  - Else ($v \neq w$) if $v = S.\mathsf{top}()$, then we set $v \leftarrow S.\mathsf{pop}()$. That is, in this case, we don't immediately transition to w, but we take this as evidence that we should not stay at v, as we did in the previous iteration. This is another type of backtracking step.

  - Otherwise, we call $S.\mathsf{push}(v)$ and set $v \leftarrow w$ for the next iteration.

We repeat this step until we are confident we can stop, which occurs when enough copies of the same vertex occur at the top of the stack. The following is the central theorem of the chapter, which we prove in the next section.

Theorem 2.1. *Given an error tolerance, $\varepsilon = n^{-c}$ for $c > 0$, and a DAG, G, with a path, P, from a vertex, s, to a distinct vertex, t, the path-guided pushdown random walk in G starting from s will terminate at t with probability at least $(1 - \varepsilon)$ after $N = \Theta(|P| + \log(1/\varepsilon))$ steps, for a transition oracle, T, with error probability $p_e < 1/15$.*

2.2.1 Correctness and analysis

We now prove that w.h.p. the path-guided random walk terminates with the correct answer to its search problem. To do so we first need to define a termination condition for the walk. It is not adequate to terminate merely after an appropriate number of steps: with constant probability, the final step of the walk will be taken after an erroneous oracle result, and may be incorrect. On the stack used to guide the algorithm, we store along with each vertex a *repetition count*, equal to one if the vertex is different from the previous vertex on the stack,

[2] If $w \neq v$ and (v, w) is not an edge in G, we immediately reject this call to T and repeat the call to T.

and equal to the previous repetition count otherwise. We terminate the algorithm when this repetition count reaches an appropriately large value, $\Theta(\log(1/\varepsilon))$.

Lemma 2.2. *If it does not terminate earlier, the path-guided random walk will reach the correct goal vertex t, within $\Theta(|P| + \log(1/\varepsilon))$ steps, with high probability, with a constant factor determined by the analysis below.*

Proof. Define a call to the transition oracle, T, in our random walk as "good" if it returns the correct answer (i.e., T does not lie) and "bad" otherwise, so that each call is bad independently with probability at most p_e. Note that if we are at a node, $v \in P$, and $v \neq t$, then a good call with either undo a previous bad call to stay at v or it will move to the next vertex in P. Also, if we are already at t, then a good call will keep us at t, adding another copy of t to the top of the stack, S. Alternatively, if we are a node, $v \notin P$, then a good call with either undo a previous bad call to stay at v or it will move back to a previous node in our traversal, which undoes a previous bad call to move to v. Moreover, if we repeat this latter case, we will eventually return back to a node in P. Thus, every good call either undoes a previous bad call or makes progress towards the target vertex, t. Admittedly, if we are at a node, $v \in P$, then a bad call can undo a previous good call (e.g., which was to move to v or stay at v if $v = t$). Also, a sequence of bad calls can even deviate from P and return back to it—but because G is a DAG, a series of bad calls cannot return back to a previously visited vertex of P. Thus, a path-guided pushdown random walk will successfully reach the target vertex, t, if the difference between the number of good calls and bad calls is at least $|P|$, the length of the path, P. Let X_i be an indicator random variable that is 1 iff the ith call to the transition oracle is bad, and let $X = \sum_{i=1}^{N} X_i$. Since each call is bad independent of all other calls, we can apply a Chernoff bound to determine an upper bound on the probability that the difference between the number of good calls and bad calls is less than $|P|$, i.e., if $(N - X) - X < |P|$, that is, if $X > (N - |P|)/2$. Further, note that

$\mu = E[X] = p_e N \leq (1/15)N$. Then, for $N = 3(|P| + \log(1/\varepsilon))$, the failure probability is

$$\begin{aligned}
\Pr\left(X > \frac{N - |P|}{2}\right) &= \Pr\left(X > |P| + \frac{3}{2}\log(1/\varepsilon)\right) \\
&= \Pr\left(X > \frac{1}{3}\left(3|P| + \frac{9}{2}\log(1/\varepsilon)\right)\right) \\
&\leq \Pr\left(X > \frac{1}{3}N\right).
\end{aligned}$$

Further, by a Chernoff bound from Dillencourt, Goodrich, and Mitzenmacher [51, Theorem 7],

$$\Pr\left(X > \frac{1}{3}N\right) < 2^{-N/3} \leq 2^{-\log(1/\varepsilon)} = \varepsilon.$$

This establishes the proof. □

As mentioned above, the condition $p_e < 1/15$ can, with the same asymptotic performance as in Theorem 2.1 be replaced with any error probability bounded away from 1/2. We caution, however, that while Theorem 2.1 provides a high-probability guarantee we reach the target vertex, t, it does not provide a high-probability guarantee we visit every vertex of the path, P. There is an exception to this, which we describe now.

Corollary 2.3. *If G is a tree, then the path-guided pushdown random walk in G will visit every node in P.*

Proof. By definition, there is a unique path P^* from the root of G to the target vertex t. Thus, according to Theorem 2.1, with high probability after N steps, the path represented by the stack S must equal the intended path P^*, which also must equal P. □

Lemma 2.4. *If it does not terminate earlier, the path-guided pushdown random walk will accumulate* $\Theta(\log(1/\varepsilon))$ *copies of the goal vertex* t *on its stack, after* $\Theta(|P| + \log(1/\varepsilon))$ *steps, and therefore terminate, with high probability.*

Proof. This follows by applying Lemma 2.2 to a modified DAG G' in which we expand each vertex v of G into a chain of copies $(v, 1)$, $(v, 2)$, $(v, 3)$, ... of v with a repetition count, as used in the termination condition of the algorithm. □

Lemma 2.5. *The path-guided pushdown random walk will not terminate with any other vertex than* t*, with high probability.*

Proof. Consider some vertex $t' \neq t$. For path-guided pushdown random walks to terminate at t', the repetition count at t' would need to reach $\alpha(\log(1/\varepsilon))$ for some constant $\alpha > 0$ determined below. Recall that $\varepsilon = n^{-c}$ for some $c > 0$, so $\alpha(\log(1/\varepsilon)) = c\alpha \log n$.

Let us assume for simplicity that, when we come across t' during our random walk, all bad calls at t' add to our repetition count and all good calls subtract from the repetition count. For us to terminate erroneously at t', we would need a sequence of calls to T such that the number of bad calls exceeds the number of good calls by $c\alpha \log n$. However, since $p_e < 1/15$, the proof of Theorem 1.2 shows that, so long as the number of calls at t' is at least $c\alpha \log n$, a majority of them will be good with probability $1 - n^{-c\alpha/2}$. Then for $\alpha \geq 2$, the good calls will subtract all repetitions at t' and cause us to leave t' with probability at least $1 - \varepsilon$. For $\alpha > 2/c$, this is a high-probability bound.

By choosing α appropriately, we can make the probability of termination at any fixed step smaller than ε/N, where N is the number of steps taken by the walk in Lemma 2.4. The result follows by applying a union bound to all the steps of the walk. □

Proof of Theorem 2.1. Termination of the algorithm w.h.p. follows from Lemma 2.4, and correctness once terminated follows from Lemma 2.5. For a given high-probability bound $1 - \varepsilon$, we apply Lemma 2.4 and Lemma 2.5 with $\varepsilon/2$ and then apply a union bound to get a bound on the probability that the algorithm terminates with a correct answer. □

2.2.2 Counterexample to Generalizations

It is very tempting to consider a generalization of the path-guided pushdown random walk defined by a set of *valid nodes* and *goal nodes*, with the property that a non-noisy search starting from any valid node will reach a goal node and then stop. With a noisy oracle that determines whether a node is valid and if so follows a noisy version of the same search, one could hope that this generalized algorithm would quickly reach a goal state. The path-guided pushdown random walk would then be a special case where the valid nodes are exactly the nodes on the non-noisy search path from the root node. However, as we show in this section, this generalization does not work with the same fast high-probability termination bounds, unless additional assumptions are made (such as the assumptions giving the special case of the path-guided pushdown random walk).

Consider the following search problem: the DAG to be searched is simply a complete binary tree with height $\log_2 n$. One root-to-leaf path on this DAG is marked as valid, with the nodes on this path alternating between non-goal and goal nodes, so that the non-noisy search from the root stops after one step but other later stops would also produce a valid result. We have a noisy oracle with the following behavior:

- At a leaf node of the binary tree, or an invalid node, it always produces the correct result.
- At a valid non-leaf node, it follows the same behavior as a non-noisy oracle with probability 14/15: that is, it correctly identifies whether the node is a goal, and if not

returns the unique valid child. With probability $K \log\log n / \log n$ (for some suitably large constant K) it returns the valid child regardless of whether the node is a goal. And with the remaining probability $1/15 - K \log\log n / \log n$ it returns the invalid child.

Now consider the behavior on this problem of a path-guided random walk, as defined above with a repetition-count termination condition. It will follow the valid path in the binary tree, with brief diversions whenever the noisy oracle causes it to walk to a node not on the path. At any goal node, it will wait at that node, accumulating more repetitions of the node, until either it achieves $C \log n$ repetitions (with a constant factor C determined by the desired high probability bound) or the noisy oracle tells it to take one more step along the valid path. The probability of reaching $C \log n$ repetitions without taking one more step is $(1 - K \log\log n / \log n)^{C \log n} \approx \exp -(K/C) \log\log n$, and by adjusting K relative to C we can make this probability less than $1/(2 \log n)$. With this choice, by the union bound, the random walk will eventually take one more step from each goal node until finally reaching the leaf goal node. The expected number of steps that it waits at each goal node will be $\Theta(\log n / \log\log n)$, so the expected number of steps in the overall random walk, before reaching the leaf goal node and then accumulating enough repetitions to terminate, will be $\Theta(\log^2 n / \log\log n)$.

It also would not work to use a termination condition of making enough steps to have high probability of reaching a goal node, and then stopping. If this number of steps is not enough to reach the leaf goal node, the probability of stopping at a non-goal node would be too high.

Thus, without either a change of algorithm or more assumptions on the valid and goal node sets, this generalized random walk can be made to take a number of steps that is longer than $|P| + \log 1/\varepsilon$ by a non-constant factor.

2.3 Transition Oracle for Binary Search Trees

While path-guided pushdown random walks applies to many DAGs, many fundamental computational geometry algorithms rely on binary search trees. In this section, we present a transition oracle for BSTs.

Say that we are attempting to find some value x in a binary search tree T. By properties of BSTs, there is a unique path from the root to the node containing x. If there is no such node, there is still a unique path to the nonexistent leaf that would contain x. When a call to the transition oracle T is good, we expect it to correctly determine if we are on said valid path only using a constant number of comparisons. To do so, we add a constant amount of extra data to each node.

For each node v of the BST, let P_v be the unique path from the root to v. Let $v.l$ be the lowest ancestor of v whose right child is in P_v (or a sentinel representing $-\infty$). Likewise, let $v.r$ be the lowest ancestor of v whose left child is in P_v (or a sentinel representing ∞). One of $\{v.l, v.r\}$ is v's parent. Observe that, if we have correctly navigated to v, x must be in between the values held by $v.l$ and $v.r$. Thus, by adding two pointers to each node, the transition oracle can determine if we are on the correct path using two comparisons. See Figure 2.2 for a visual depiction of the pointers $v.l$ and $v.r$ and their effect. Augmenting the tree with these pointers can be done in $O(\log n)$ extra time per insertion and deletion. With slight modification, we can also support inexact queries such as for predecessors or successors.

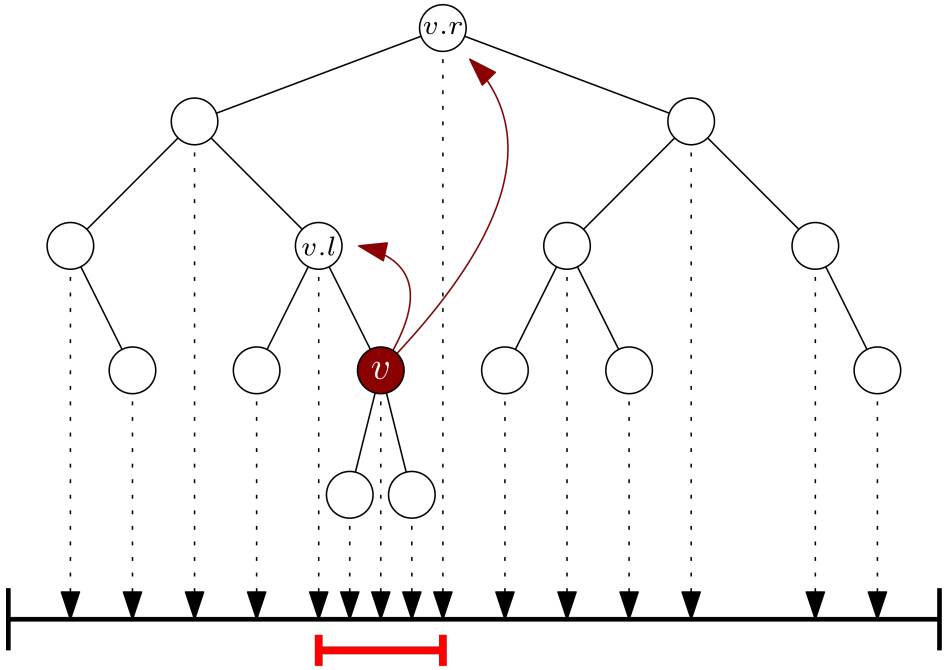

Figure 2.2: Here we have depicted the $v.l$ and $v.r$ pointers for the node v. Notice that $v.l$ is v's first ancestor whose value is smaller than v's. Likewise, $v.r$ is its first ancestor whose value is larger than v's. The dotted lines map nodes to their relative ordering on a number line. It is clear that the values held by $v.l$ and $v.r$ form the interval of possible values that could be held by nodes in v's subtree. If the query is located strictly in this interval and if it is in the tree at all, then it must be within v's subtree. If the query is not in this interval, then it cannot be in v's subtree and we have made an errant comparison earlier.

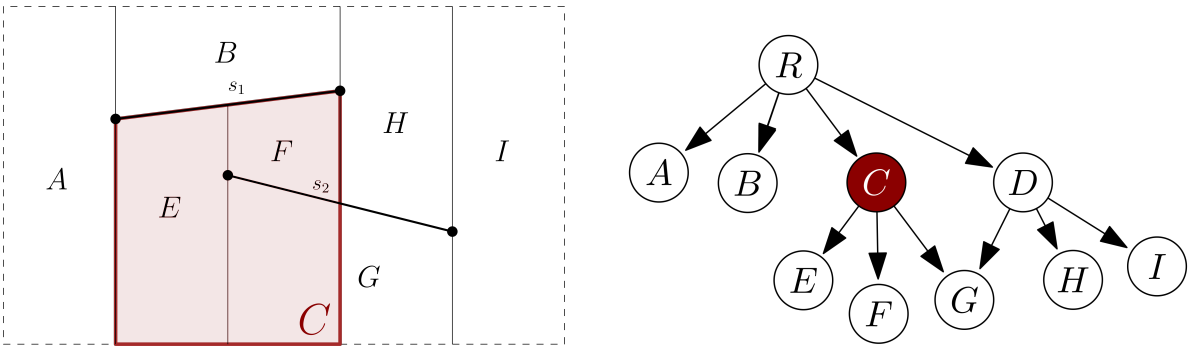

Figure 2.3: An instance of a path-guided pushdown random walk in the trapezoidal map history DAG. R represents the initial bounding box, the leaf nodes represent currently visible trapezoids. The remaining nodes represent destroyed trapezoids. We are at node C in a path-guided pushdown random walk, so we test if our query point lies in trapezoid C, the shaded region in the decomposition.

2.4 Noisy Randomized Incremental Construction for Trapezoidal Maps

In this section, we show how a history DAG in a randomized incremental construction (RIC) algorithm can be used as the DAG of Theorem 2.1. Suppose we are given a set of n non-crossing line segments in the plane and wish to construct their trapezoidal decomposition, $\mathcal{D}$. See Section 1.4.2 for a review of the RIC that computes the trapezoidal decomposition. We note that there exists another variant of a history DAG that represents the segments themselves as nodes in the DAG [46, Chapter 6.2], which seems less usable when primitives are noisy. To make this RIC algorithm noise-tolerant, we must solve two issues. The first is to navigate the history DAG in $O(\log n)$ time w.h.p.; the second is to walk along each segment to merge and destroy trapezoids when it is added.

To navigate down the history DAG, we apply a path-guided pushdown random walk. To do so we must test, with a constant number of operations, whether we are on the path that a non-noisy algorithm would search to find the query endpoint q of a new segment, s_i. See Figure 2.3 for an example of this process. Each node of our history DAG represents a trapezoid (either destroyed if an internal node or current if a leaf node). Importantly,

each of the at most four children of a node cover the parent's trapezoid, but do not overlap. Therefore, we can define a unique path, P w.r.t query q, to be a sequence of trapezoids beginning at the root of the history DAG and ending at the leaf-node trapezoid containing q. Each node on the path is the unique child of the previous node that contains q. Thus, all our transition oracle must do to determine if we are on the correct path is to check whether q is contained by the trapezoid mapped to our current node v. If this test succeeds, the oracle determines (rightly or wrongly) that v is on a valid path, and it proceeds to compare q against the segment whose addition split trapezoid v in order to return the next node of the walk. If one or more of these tests fails, the oracle says that v is not on a valid path. Let ℓ_i be the sum of the lengths of the two unique paths corresponding to the endpoints of s_i. If we set our error probability for path-guided pushdown random walks to be w.h.p. in n, the point-location cost to insert the ith segment is $O(\ell_i + \log n)$.

For the second issue, suppose there are d_i trapezoids between the left and right endpoint of the ith segment to be inserted, and that we need to walk left-to-right in the current subdivision to find them. To find the next trapezoid in this walk, we simply test if the segment endpoint that defines the right boundary of the current trapezoid lies above or below segment s_i, e.g., determining whether to choose its upper-right or lower-right neighbor. Combining this above-below test with the trivial repetition strategy, we can compute the correct sequence of trapezoids in $O(d_i \log n)$ time w.h.p.

Via the standard backwards analysis, $\sum_{i=1}^{n} \ell_i = O(n \log n)$ and $\sum_{i=1}^{n} d_i = O(n)$ in expectation [46]. It has also been shown [46, 129] that the search depth of the final history DAG is $O(\log n)$ w.h.p. Therefore, after constructing the decomposition, we can use path-guided pushdown random walks to answer planar point-location queries in $O(\log n)$ time w.h.p. We conclude the following.

Theorem 2.6. *We can successfully compute a trapezoidal decomposition map of n non-crossing line segments w.h.p. in expected $O(n \log n)$ time, even with noisy primitives, and*

thereby construct a data structure (the history DAG) that answers point location queries in $O(\log n)$ *time w.h.p.*

It is natural to hope that this can be extended to line segment arrangements with crossings, for which the best non-noisy time bounds are $O(n \log n + k)$, achieved with a similar randomized incremental approach. However, as we explain in Section 2.5.2 and Section 2.8.2, an extra logarithmic factor may be necessary.

2.5 Plane-Sweep Algorithms

In this section, we show that many plane-sweep algorithms can be adapted to our noisy-primitive model.

2.5.1 Noisy Balanced Binary Search Trees

We begin by noting that we can implement binary search trees storing geometric objects to support searches and updates in $O(\log n)$ time w.h.p. by adapting the noisy balanced binary search trees recently developed for numbers in quantum applications by Khadiev, Savelyev, Ziatdinov, and Melnikov [97] to our noisy-primitive framework for geometric objects. Plane-sweep algorithms require us to store an ordered sequence of events that are visited by the sweep-line throughout the course of the algorithm while maintaining an active set of geometric objects for the sweep line in a balanced binary tree. Some plane-sweep algorithms, such as the one of Lee and Preparata that divides a polygon into monotone pieces [103], have a static set of events that can simply be maintained as a sorted array that is constructed using an existing noisy sorting algorithm [61]. Others, however, add or change events to the event queue as the algorithm progresses. In addition, the algorithms must also maintain already-processed data in a way that allows for new data swept over to be efficiently incorporated. The two most common dynamic data structures in plane-sweep

algorithms are dynamic binary search trees and priority queues. Throughout this section, noise is associated with the comparison "$a \leq b$?" where a and b are geometric objects.

Once a data structure is built, its underlying structure can be manipulated without noise. For example, we need no knowledge of the values held in a tree to recognize that it is imbalanced and to perform a rotation, allowing implementation of self-balancing binary search trees. Khadiev, Savelyev, Ziatdinov, and Melnikov [97] show how to implement red-black trees [84] in the noisy comparison model for numbers. We observe here that the same method can be used for ordered geometric objects compared with noisy geometric primitives, with a binary search tree serving as the search DAG and a root-to-leaf search path as the path. See Section 2.3 for an explanation for how to instantiate path-guided pushdown random walks on a binary search tree.

2.5.2 Noisy Trapezoidal Decomposition with Segment Crossings

We can implement an event queue as a balanced binary search tree with a pointer to the smallest element to perform priority queue operations in the noisy setting.

Theorem 2.7. *Given a set, S, of n x-monotone pseudo-segments[3] in the plane, we can construct a trapezoidal map of S in $O((n+k)\log n)$ time w.h.p., where k is the number of pairs of crossing segments, even with noisy primitive operations.*

Proof. Bentley and Ottmann [16] compute a trapezoidal map of line segments S via a now well-known plane-sweep algorithm, which translates directly to x-monotone pseudo-segments. Pseudo-segment endpoints and intersection points are kept in an event queue ordered by x-coordinates and line segments intersecting the sweep line are kept in a balanced binary search tree, both of which can be implemented to support searches and updates in $O(\log n)$ time w.h.p. in the noisy model, as described above. Given the $O((n+k)\log n)$-

[3] A set of x-monotone psuedo-segments is a set of x-monotone curve segments that do not self-intersect and such that any two of them intersect at most once [4, 37].

time performance of the Bentley-Ottmann algorithm, this implies that we can construct the trapezoidal map of S in $O((n+k)\log n)$ time w.h.p. □

We note that this running time matches the construction time in Theorem 2.6 for non-crossing line segments, but it does not give us a point-location data structure as in Theorem 2.6. Further, the upper bound of Theorem 2.7 is optimal with noisy primitive operations for $k = \Theta(n^2)$ via a reduction from computing line arrangements to computing trapezoidal decompositions. We show that computing an arrangement of n lines takes $\Omega(n^2 \log n)$ time in Section 2.8.2.

2.5.3 Noisy Closest Pair

Here, we present another plane-sweep application in the noisy setting.

Theorem 2.8. *We can find a closest pair of points, from n points in the plane, with noisy primitives, in time $O(n \log n)$ w.h.p.*

Proof. Hinrichs, Nievergelt, and Schorn show how to find a pair of closest points in the plane in $O(n \log n)$ time [86] by sorting the points by their x-coordinate and then plane-sweeping the points in that order. They maintain the minimum distance δ seen so far, and a *y-table* of *active points*. A point is active if has been processed and its x-coordinate is within δ of the current point; once this condition stops being true, it is removed from the y-table. The y-table may be implemented using a balanced binary tree. When a point is processed, the y-table is updated to remove points that have stopped being active, by checking the previously-active points sequentially according to the sorted order until finding a point that remains active. The new point is inserted into the y-table, and a bounded number of its nearby points in the table are selected. The distances between the new point and these selected points are compared to δ, and δ is updated if a smaller distance is found.

In our noisy model, sorting the points takes $O(n \log n)$ time w.h.p. [61]. Checking whether a point has stopped being active and is ready to be removed from the y-table may be done using the trivial repetition strategy; its removal is a non-noisy operation. Inserting each point into the y-table takes $O(\log n)$ time w.h.p. Selecting a fixed number of nearby neighbors is a non-noisy operation, and comparing their distances to δ may be done using the trivial repetition strategy with a noisy primitive that compares two distances determined by two pairs of points. In this way, we perform $O(\log n)$ work for each point when it is processed, and $O(\log n)$ work again later when it is removed from the y-table. Overall, the time is $O(n \log n)$ w.h.p. □

We give this result mostly as a demonstration for performing plane sweep in the noisy model, since a closest pair (or all nearest neighbors) may be found by constructing the Delaunay triangulation (Section 2.7) and then performing min-finding operations on its edges.

An algorithm by Rabin [121] finds 2D closest points in $O(n)$ time, beating the $O(n \log n)$ plane sweep implementation that our solution is based on, in a model of computation allowing integer rounding of numerical values derived from input coordinates. However, computing the minimum of n elements takes $O(n \log n)$ time for noisy comparison trees [61], and Rabin's algorithm includes steps that find the minimum among $O(n)$ distances, so it is not faster than our algorithm in our noisy model. In Section 2.8.1, we prove that if data is accessed only through noisy primitives that combine information from $O(1)$ data points (allowing integer rounding), then finding the closest pair w.h.p. requires time $\Omega(n \log n)$. Thus, the plane-sweep closest-pair algorithm is optimal in this model.

2.6 Convex Hulls

In this section, we describe algorithms for constructing convex hulls that can tolerate probabilistically noisy primitive operations. Here, primitive operations are orientation tests and

visibility tests for 2D and 3D convex hulls respectively. Sorting is also used throughout, so comparing the x or y coordinates of two points is also a noisy primitive.

2.6.1 Static Convex Hulls in 2D

We begin by observing that it is easy to construct two-dimensional convex hulls in $O(n \log n)$ time w.h.p. Namely, we simply sort the points using a noise-tolerant sorting algorithm [61] and then run Graham scan [46]. The Graham scan phase of the algorithm uses $O(n)$ calls to primitives, so we can simply use the trivial repetition strategy to implement this phase in $O(n \log n)$ time w.h.p. We note that this is the best possible, however, since just computing the minimum or maximum of n values has an $\Omega(n \log n)$-time lower bound for a high-probability bound in the noisy primitive model [61].

2.6.2 Dynamic Convex Hulls in 2D

Overmars and van Leeuwen [115] show how to maintain a dynamic 2D convex hull with $O(\log^2 n)$ insertion and deletion times, $O(\log n)$ query time, and $O(n)$ space. They maintain a binary search tree, T^*, of points stored in the leaves ordered by y-coordinates and augment each internal node with a representation of half the convex hull of the points in that subtree. They define an lc-hull of P as the convex hull of $P \cup \{(\infty, 0)\}$, which is the left half of the hull of P (rc-hull is defined symmetrically). It is easy to see that performing the same query on both the lc- and rc-hull of P allows us to determine if a query point is inside, outside, or on the complete convex hull. We follow this same approach, showing how their approach can be implemented in the noisy-primitive model using path-guided pushdown random walks.

Overmars and van Leeuwen show that two lc-hulls A and C separated by a horizontal line can be merged in $O(\log n)$ time. Rather than being a single binary search, however, their method involves a joint binary search on the two trees that represent A and C with the aim of finding a tangent line that joins them. At each step, we follow one of ten cases to determine how to

navigate the trees of A and/or C. Because each decision may only advance in one tree, noisy binary search is not sufficient. Fortunately, as we show, the path-guided pushdown random walk framework is powerful enough to solve this problem in the noisy model. The challenge, of course, is to define an appropriate transition oracle. Say that the transition oracle has the ability to identify which case corresponds to our current position in A and C. This decision determines which direction to navigate in A and/or C and corresponds to choosing a child to navigate down an implicit decision DAG wherein each node has ten children. It remains to show how a transition oracle acting truthfully can determine if we are on the correct path.

We have each node of A and C maintain $v.l$ and $v.r$ pointers as described in Section 2.3. The values at $v.l$ and $v.r$ are ancestors of v whose keys are just smaller and just larger than v respectively. They determine the interval of possible values of any node in v's subtree, which corresponds to an interval of points on each hull. See Figure 2.4 for an example.

Say we are currently at node $a \in A$ and node $c \in C$. We can perform four case comparisons: $a.r$ with $c.r$, $a.r$ with $c.l$, $a.l$ with $c.r$, and $a.l$ with $c.l$. Each outputs a region of A and C that the two tangent points must be located in. If the intersection of all four case comparisons contains the regions bounded by $a.r$ and $a.l$ as well as $c.r$ and $c.l$, then we are on a valid path by the correctness of Overmars and van Leeuwen's case analysis [115]. Since the noise-free process takes $O(\log n)$ time, with a path-guided pushdown random walk using the transition oracle described above, this process takes $O(\log n)$ time w.h.p.

To query a hull, we can perform noisy binary search on the lc- and rc-hull structures using the transition oracle of Section 2.3. Updating the convex hull utilizes the technical lemma discussed above along with split and join operations on binary search trees [115]. We conclude the following.

Theorem 2.9. *We can insert and delete points in a planar convex hull in $O(\log^2 n)$ time per update w.h.p., even with noisy primitives.*

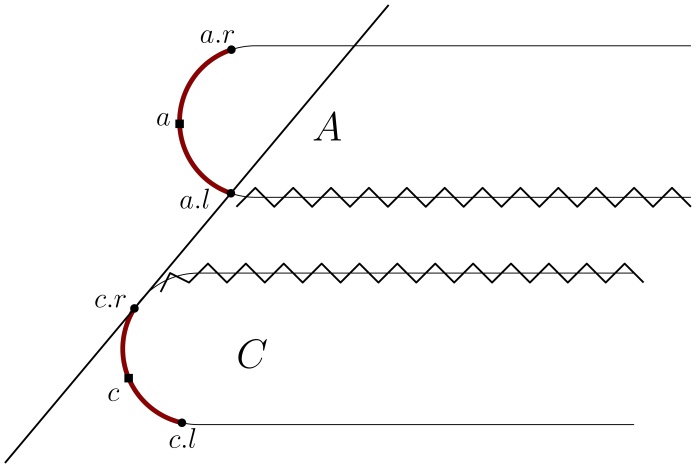

Figure 2.4: Here we show one of four comparisons computed by the transition oracle. The points a and c represent the node we are at in each structure. The highlighted regions represent all possible values in that node's subtree, bordered by points $a.r$ and $a.l$ (resp. $c.r$ and $c.l$). This comparison is valid as it does not eliminate the highlighted regions. If all four are valid, then we must be on the right path in our double binary search.

2.6.3 Convex Hulls in 3D

In this section, we show how to construct 3D convex hulls in $O(n \log n)$ time w.h.p. even with noisy primitive operations. The main challenges in this case are first to define an appropriate algorithm in the noise-free model and then define a good transition oracle for such an algorithm. For example, it does not seem possible to efficiently implement the divide-and-conquer algorithm of Preparata and Hong [120] in the noisy primitive model as each combine step performs $O(n)$ primitive operations. See Section 1.4.2 for an explanation of the 3D hull random incremental construction we base our solution on. We begin by constructing a tetrahedron through any four points; we need the orientation of these four points, which can be found by the trivial repetition strategy in $O(\log n)$ time. The points within this tetrahedron can be discarded, again using the trivial repetition strategy in $O(\log n)$ time per point. The main challenge to designing a transition oracle for the history DAG in this method is its nested nature. We have the nodes in the DAG themselves representing facets and pointing to the facets that replaced them when they were deleted. We also have the

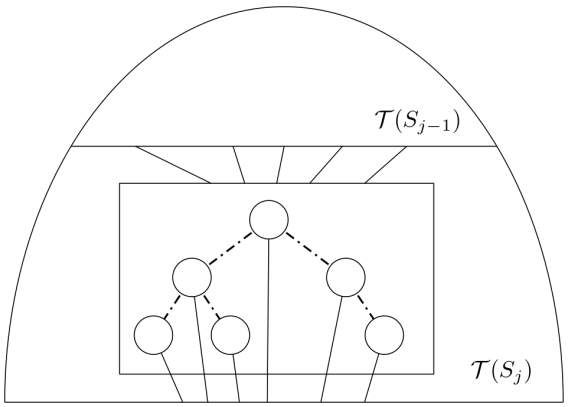

Figure 2.5: A depiction of how the radial-search structure is embedded into the history DAG. $\mathcal{T}(S_{j-1})$ represents the history DAG before the jth point was added to the hull. Every node that was a leaf in $\mathcal{T}(S_{j-1})$ now points to each node created in $\mathcal{T}(S_j)$. Destroyed nodes have pointers to all created nodes. The dashed lines between the nodes in $\mathcal{T}(S_j)$, represent the connections of the radial search structure, represented as a BST.

radial search structures, which maintain the set of new facets produced in a given iteration in sorted clockwise order around the point that generated them. See Figure 2.5.

Our transition oracle needs to be able to determine if we are on the correct path. Say we are given a query point, q_i, being inserted in the ith iteration of the RIC algorithm and are at node v of the history DAG. To see if we are performing the right radial search, we first check if the ray from the origin to q passes through v's parent (in the history DAG, not the radial search structure). Then we must determine if we are performing radial search correctly. To do so, we can augment the radial search tree as described in Section 2.3 and perform two more orientation tests to determine if v is on the right path in our current radial search structure. If a comparison returns false, then we use our stack to backtrack up the radial search structure or, if we are at its root, back to the parent node in the history DAG and its respective radial search tree. Thus, in just three comparisons, we can determine if we are on the right path in the history DAG. Similar to our trapezoidal decomposition analysis, we observe that the point-location cost in each iteration is $O(\ell_i + \log n)$, where ℓ_i is the length of the unique path in the history DAG corresponding to our query q. An analysis by

Mulmuley [112] shows that, over all n iterations, $\sum_{i=1}^{n} \ell_i = O(n \log n)$ in expectation. As a result, total point-location cost is $O(n \log n)$ in expectation.

Once a conflicting facet is found through the random walk, we can again walk around the hull and perform the trivial repetition strategy to determine the set of all conflicting facets, X_i, in $O(|X_i| \log n)$ time. Each facet must be created before it is destroyed, so $\sum_{i=1}^{n} |X_i|$ is upper-bounded by the total size of the history DAG. The same analysis by Mulmuley [112] shows that the expected size of the history DAG is $O(n)$. We conclude the following.

Theorem 2.10. *We can successfully compute a 3D convex hull of n points w.h.p. in expected $O(n \log n)$ time, even with noisy primitives.*

2.7 Delaunay Triangulations and Voronoi Diagrams

In this section, we describe an algorithm that computes the Delaunay triangulation (DT) of a set of points in the plane in the noisy-primitives model. Noise is associated with determining whether a point p_i lies within some triangle Δ and whether a point p_i lies in the circle defined by some Delaunay edge e. Here, we describe the algorithm under the Euclidean metric. We later show how to generalize our algorithm for other metrics.

Once again, we use a history DAG RIC similar to what we described in the previous section. This time, each node in the DAG represents the triangles that exist throughout the construction of the triangulation (see Mulmuley [112] for details). A leaf node is a triangle that exists in the current version, and an internal node is a triangle that was destroyed in a previous iteration. When we wish to insert a new point p_i, we first use the history DAG to locate where p_i is in the current DT. If p_i is within a triangle Δ, we split Δ into three triangles and add them as children of Δ. Otherwise p_i is on some edge e, and we split the two adjacent triangles Δ_1 and Δ_2 into two triangles each. Once this is done, we repeatedly flip edges that violate the empty circle property so that the triangulation becomes a DT again

(see [46] Chapter 9.3). As described in [112], we store the new triangles in a radial search structure sorted CCW around the added point p_i. Using a path-guided pushdown random walk in the history DAG to find the triangle containing p_i is broadly similar to the process used to find conflicting faces in the 3D convex hull RIC. Once again, we use the ray-shooting and radial search queries described in Section 2.6.3. With a similar transition oracle to 3D convex hulls, we can determine where p_i is in the current DT in $O(\ell_i + \log n)$ time, where ℓ_i is the length of p_i's unique path in the history DAG. Unlike for 3D convex hulls, there is a unique triangle or edge that is in conflict with p_i, which simplifies the process of determining a unique path through the history DAG. We observe that if p_i is not located in the triangle Δ represented by the current node in the DAG, then we are not on the right path. Once we have found the triangle containing p_i and added in new edges, we begin edge flipping. Using the trivial repetition strategy, each empty circle test costs $O(\log n)$ time. The number of empty circle tests is proportional to the number of edges flipped, which is less than the total number of triangles created throughout the algorithm. Once again, Mulmuley's [112] analysis shows that $\sum_{i=1}^{n} \ell_i = O(n \log n)$ in expectation and the number of empty circle tests needed to update the DAG after point-location is $O(n)$ in expectation. We conclude the following.

Theorem 2.11. *We can successfully compute a Delaunay triangulation of n points w.h.p. in expected $O(n \log n)$ time, even with noisy primitives.*

We remark that this also leads to an algorithm for constructing a Euclidean minimum spanning tree, in which we construct the Delaunay triangulation, apply a noisy sorting algorithm to its edge lengths, and then (with no more need for noisy primitives) apply Kruskal's algorithm to the sorted edge sequence. In the following section, we generalize our Delaunay triangulation algorithm to work for a wider class of metrics, including all L_p metrics for $1 < p < \infty$.

2.7.1 Generalized Delaunay Triangulations

In this section, we show the following:

Theorem 2.12. *Given n points in the plane, we can successfully compute a generalized Delaunay triangulation generated by homothets of any smooth shape w.h.p. in expected time $O(n \log n)$, even with noisy primitives.*

A *shape Delaunay tessellation* is a generalization of a Delaunay triangulation first defined by Drysdale [53]. Rather than a circle, we instantiate some convex compact set C in $\mathbf{R}^2$ and redefine the empty circle property on an edge $\overline{pq}$ like so: given a set of points S and some shape C, the edge $\overline{pq}$ exists in the shape Delaunay tessellation $DT_C(S)$ iff there exists some homothet of C with p and q on its boundary that contains no other points of S. We will consider smooth shapes, as arbitrary non-smooth shapes may cause $DT_C(S)$ to not be a triangulation under certain point sets [12]. Note that all L_p metrics for $1 < p < \infty$ correspond to smooth shapes (rather than a unit circle, they are unit rounded squares) and furthermore, all smooth shapes produce triangulations of the convex hull of S [12, 133]. Similar to above, we use the general position assumption that no four points lie on a homothet of C.

After inserting a point, p_r, in the incremental algorithm, we draw new edges that connect p_r to the points that comprise the triangle(s) that p landed in. These new edges are Delaunay as either they are fully enclosed by the homothets of C that covered the triangle(s) p was located in or they are enclosed by the union of previously-empty shapes that covered each edge of the triangles. After this, the algorithm finds adjacent edges, tests if they obey the empty shape property, and flips them if not. Aurenhammer and Paulini have shown that there exists a lexicographical ordering to these triangulations, similar to how the Delaunay triangulation in L_2 maximizes the angle vector [12, Theorem 3]. Thus, for any convex shape C, local Delaunay flips lead to $DT_C(S)$ and so the flipping algorithm used after an

incremental insertion will terminate at a triangulation that is $DT_C(P_r)$, where P_r is the set $\{p_1, ..., p_r\}$.

Because the flipping process works as before, the point location structure can be easily adapted to this setting. As stated above, using a smooth shape C guarantees that $DT_C(S)$ is an actual triangulation as opposed to a tree. Thus, in every iteration, there exists a triangle that the new point is enclosed by. The history DAG data structure is agnostic to the specific edge flips being made during the course of the algorithm and so works as expected. Our transition oracle also behaves the same.

In the Euclidean case, the standard backwards analysis (e.g., de Berg *et al.* [46]) uses the fact that a Delaunay triangulation of r points has $O(r)$ edges, any new triangle created in iteration r is incident to the newly inserted point p_r, and triangles are defined by at most three points. By planarity and by construction of the algorithm, all three hold true in this generalized case. Thus, we have the desired result.

2.8 Lower Bounds

In this section, we prove lower bounds on computing with noisy primitives showing that for finding the closest pair of points (which takes $O(n)$ time in the non-noisy setting in a model of computation allowing integer rounding) and for constructing line arrangements (which takes time $O(n^2)$ in the non-noisy setting) it is not possible to avoid the logarithmic time penalty imposed by the trivial repetition strategy.

2.8.1 Closest Points

In this section we prove that computing the closest point w.h.p. with noisy primitives requires $\Omega(n \log n)$ time; thus, our $O(n \log n)$ time algorithm is optimal, despite the existence of non-noisy closest point algorithms taking time $O(n)$. For this section, we make no assumption

that the computation can be modeled as a comparison tree or decision tree (unlike past lower bounds for noisy minimum-finding). Instead, we assume merely that the only direct access to the input data is through noisy primitives. We assume that each primitive takes as input $O(1)$ data points and produces an arbitrary value (not necessarily Boolean) as output, which is correct with probability $1 - p$ and incorrect with probability p, for some constant error probability $p < \frac{1}{2}$. For our lower bound, we restrict the incorrect values to be values that could have been produced by a non-noisy version of the primitive on different data points; restricting the model in this way only makes the lower bound stronger. Once a value has been returned from a noisy primitive, the algorithm is free to perform arbitrary computation on it. However, with this restriction, incorrect values may be chosen adversarially.

Theorem 2.13. *Let $c > 0$. Then in the model of computation described above with constant error probability p, computing a closest point among n 2D points, with probability $\geq 1 - n^{-c}$, requires calls to $\Omega(n \log n)$ noisy primitives, even for the expected number of calls made by a randomized algorithm, and even for noisy primitives that (when erroneous) produce an answer that would be valid for an arbitrarily small perturbation of the input relative to the closest pair distance.*

Proof. Let n be any multiple of four, and consider any point set S in general position, grouped into $n/2$ pairs of points, each pair approximately at unit distance (as our general position assumption allows), with all other distances larger. We define a hard distribution on random instances R to the closest pair problem by choosing one of the $n/2$ pairs of points uniformly at random and perturbing this pair to be closer than all other pairs. Additionally, we specify a noisy primitive that, with probability p, answers with the result that would be correct for S instead of for the perturbed input R. By choosing the perturbations appropriately, this erronous result will be a result that would be valid for an arbitrarily small perturbation of the input. We will first show that, for this input distribution and this adversary, a deterministic

algorithm requires $\Omega(n \log n)$ noisy primitives to solve the problem correctly with the given probability.

Let t denote the maximum number of data points that participate in a single primitive. For some suitably small constant κ, we have $p^{\kappa \log n} > 2n^{-c}$: that is, if we make $\kappa \log n$ calls to noisy primitives, there is a somewhat large probability that all of them produce erroneous outputs. Now suppose that a deterministic algorithm makes at most $\frac{\kappa}{4t} n \log n$ calls to noisy primitives for any n-point input, and consider the sequence of calls that it would make for point set S; this sequence is deterministic as our noise model will not affect the result of any primitive on input S. Among the $n/2$ unit-distance pairs of points in S, let F be the set of pairs whose two points are involved in at most $\kappa \log n$ calls to primitives in this sequence of calls, and such that this pair is not the output of the algorithm on input S. The average number of calls that involve at least one point from a pair is at most $\frac{\kappa}{2} \log n$ per pair, and by Markov's theorem more than half of the pairs have a number of calls that is at most twice the average. After removing the output of the algorithm on S from this set of pairs with few calls, we have that $|F| \geq n/4$.

For the two events such that (1) our hard distribution chooses a pair in F to make closest and (2) the first $\kappa \log n$ calls involving the two points from this pair are all noisy, the first event happens with probability $\geq 1/2$. After conditioning on the choice of pair, the second event happens with probability $> 2n^{-c}$. Therefore, both events happen with probability $> n^{-c}$. For all such combinations of events, the deterministic algorithm will receive the same results of primitives as it would with input S, and will produce the same output as it would with input S, but this output is not in F. Thus, the algorithm will be mistaken with probability $> n^{-c}$. Because the algorithm was arbitrary, every deterministic algorithm fails to have high probability of correctness on this input distribution.

Yao's principle [139] allows us to convert this lower bound on random inputs and deterministic algorithms into a lower bound on random algorithms, as stated in the theorem.

This principle is closely related to the minimax principle for zero-sum games and the duality principle for linear programs. It states that, for arbitrary definitions of the cost of an algorithm, the minimum expected cost for a worst-case random distribution on inputs and a resource-bounded deterministic algorithm chosen for that distribution, equals the minimum expected cost for a random algorithm with the same resource bound against a deterministic input chosen for that algorithm. It follows that any valid limit on the performance that can be achieved, proven by finding an input distribution that prevents deterministic algorithms from achieving any better performance, applies as well to the performance that can be achieved by a random algorithm on its worst-case input.

The probability of an incorrect result of an algorithm with noisy primitives is the expected cost, for a cost that is 0 when the algorithm is correct and 1 when it is incorrect. By Yao's principle, applied to the input distribution described above, any randomized algorithm that always makes at most $\frac{\kappa}{4t} n \log n$ calls to noisy primitives has a worst-case input causing its probability of an incorrect result to be too high. We can extend this limitation to algorithms whose number of calls to primitives is itself random, as follows. If a randomized algorithm has an expected number of calls that is at most $\frac{\kappa}{8t} n \log n$, then by Markov's inequality it has probability $\geq 1/2$ of making a number of calls that is at most $\frac{\kappa}{4t} n \log n$, and therefore of making too few calls to achieve high-probability correctness. By adjusting the parameters of the argument we can ensure that it has probability $\geq 1/2$ of making so few calls that it is incorrect with probability $\geq 2n^{-c}$, so that even if it is perfectly correct in the cases where it makes more than twice its expected number of calls, it will still fail to achieve the given high-probability bound. □

The same lower bound applies to the problem of detecting whether a system of unit disks has a pair of disks that intersect, by scaling the same hard distribution so that, for unit disks centered at its points, the only pair that can intersect is the randomly selected closest pair.

2.8.2 Line Arrangements

In this section we prove a lower bound on the construction of line arrangements. The standard primitive needed for this construction, in the non-noisy case, is an *orientation test*, which for three lines determines whether they meet at a single point or, if not, how they are ordered around the triangle that they form. (This is the projective dual to an operation that takes as input three points and determines whether they are collinear or, if not, how they are ordered around the triangle that they form, and constructing the arrangement is equivalent by projective duality to the problem of determining the order type of a set of n points [59].) Unlike the lower bound for closest pairs, which was agnostic about the primitives used, our lower bound for arrangements will be specific to this primitive.

Theorem 2.14. *Constructing an arrangement of n lines with high probability, using a noisy three-line orientation test, requires $\Omega(n^2 \log n)$ calls to this test, even for the expected number of calls made by a randomized algorithm.*

Proof. Erickson and Seidel [60] construct an arrangement A_n of n lines, no three having a common intersection, for which there are $\frac{1}{9}n^3 + O(n)$ *collapsible triangles*, triples of lines whose orientation can be changed so that they have a common intersection or so that their triangle is reversed without changing the results of any other orientation test. We define a hard input distribution for the problem of constructing arrangements by choosing one of the collapsible triangles in A_n and reversing it, and using a noisy orientation test that, when erroneous, returns the orientation in A_n instead of in this perturbed orientation.

Then, as in the proof of Theorem 2.13, consider any deterministic algorithm running on this input distribution, and let Δ_0 be the collapsible triangle (if one exists) that this algorithm would determine to have been reversed when all its queries are answered correctly for A_n instead of for the perturbed input. For any constant c, if the algorithm performs at most $\frac{1}{18}cn^2 \log n$ calls to the noisy orientation test, then at least half of the collapsible triangles

in A_n other than Δ_0 will be tested at most $c \log n$ times. This leads to probability $\Omega(1/n^c)$, that all of the tests to the reversed triangle are erroneous and that the algorithm will fail to correctly identify the reversed triangle. By adjusting c this failure probability can be made greater than any high-probability bound; thus, no deterministic algorithm can succeed with high probability against this input distribution. An application of Yao's principle extends this impossibility result from deterministic to randomized algorithms. We omit the detailed calculations as they are the same in essential respects as for Theorem 2.13. □

Theorem 2.15. *Determining whether a given set of n points has three collinear points w.h.p., using a noisy three-point orientation test, requires $\Omega(n^2 \log n)$ calls to this test, even for the expected number of calls made by a randomized algorithm.*

Proof. The proof is essentially the same as Theorem 2.14 except that we collapse one collapsible triangle rather than reversing it, and then construct the projective dual system of points. In its most basic form, the dual of the construction of Erickson and Seidel [60] has many collinearities separate from the collapsible triangles (it consists of $n/3$ points on each of three parallel lines, with the collapsible triangles formed by triples of one point from each of these three lines). However, as Erickson and Seidel describe, it can be modified by replacing these lines by shallow convex curves to eliminate these extra collinearities while still preserving the collapsibility of the collapsible triangles. □

For constructing line segment arrangements, additional primitives are required to determine whether two line segments cross or whether one ends before reaching the point where it would cross another segment. For instance, this may be done with orientation tests for triples of endpoints of the segments, as well as orientation tests of triples of lines passing through segments. The same lower bound applies, to a line segment arrangement formed by intersecting Erickson and Seidel's arrangement A_n with a large bounding disk, as long as these additional primitives are *uninformative*: they do not supply any information about

whether any collapsible triangle has been reversed. For instance, orientation tests on line segment endpoints are uninformative: both A_n and its perturbation, after intersection with a large bounding disk, have segment endpoints in the same convex position, so the result of an endpoint orientation test does not change when any collapsible triangle is reversed. Therefore, constructing line segment arrangements with both line and point orientation tests may require $\Omega(n^2 \log n)$ time in the noisy setting.

Chapter 3

Optimal Parallel Algorithms for Convex Hulls in 2D and 3D under Noisy Primitive Operations

3.1 Background

Fault tolerance is an important issue in parallel algorithm design, since parallelism naturally introduces more opportunities for faults to occur. As noted earlier, the noisy primitives model for fault tolerance has been extensively studied for binary comparisons of 1-dimensional values. This is also true in the parallel algorithms setting. In 1994, Feige, Raghavan, Peleg, and Upfal [61] showed how to sort n items in $O(\log n)$ span with $O(n \log n)$ work in the CRCW PRAM model, with high probability in the same noisy comparison model, and Leighton, Ma, and Plaxton [104] show how to achieve these bounds in the EREW PRAM model. Both algorithms make extensive use of the AKS sorting network [6, 7], however, which implies that their asymptotic performance bounds have very large constant factors. Recently, in a SPAA 2023 paper, Goodrich and Jacob [75] provide a parallel sorting algorithm

where comparisons are either persistently or non-persistently faulty without using the AKS sorting network. Unfortunately, their algorithm is quite complex, involving several rounds of parallel algorithms and a final clean-up process. It also only guarantees correctness for $p < 1/16$ due to a reliance on WindowSort [67, 68].

In Chapter 2, we use a technique based on random walks in DAGs to develop optimal sequential algorithms for several computational geometry problems resilient to Boolean geometric primitives. These methods seem to be inherently sequential, however, in that our 2D convex hull algorithm uses a noise-tolerant version of the Graham scan algorithm, and our sequential 3D convex hull algorithm applies path-guided pushdown random walks to an RIC, which seems to be a poor candidate for the RIC parallelization techniques of Blelloch, Gu, Shun, and Sun [26, 27], due to their methods needing correct primitive operations. Thus, new approaches are needed.

Since computations cannot be fully correct with persistent errors, we again focus in this chapter on the design of efficient parallel computational geometry algorithms using non-persistent Boolean geometric primitives, such as orientation queries or sidedness tests, that randomly return the wrong answer independently of previous queries with probability at most $p < 1/2$ (where p is known).

3.1.1 Our Results

In this chapter, we give efficient parallel algorithms for constructing convex hulls of n points in 2D and 3D when primitive orientation tests return the wrong answer with probability $p < 1/2$. We give algorithms in the CREW PRAM model that have optimal $O(\log n)$ span and $O(n \log n)$ work, w.h.p. The lower bounds follow from lower bounds on computing the minimum/maximum of a set of n points [61, 85]. We also give a simple parallel noisy sorting algorithm that runs in the same time bounds, which are also optimal [75].

Our algorithms involve parallel divide-and-conquer and novel versions of the *failure sweeping* technique [61, 71, 75, 109], which is potentially of independent interest. The failure sweeping technique is a paradigm for increasing the success probability of a divide-and-conquer parallel algorithm. Unfortunately, when a success probability depends on the input size, the success probability of recursive calls can be too low. The solution, then, is to design an oracle that can test w.h.p. the solutions to the recursive subproblems, and then "sweep" failed subproblems into a memory space for which we can then apply additional parallel resources to solve. The trick is to show that the total size of all such failure subproblems is relatively small such that we can then recompute them all efficiently. In this chapter, we show how to adapt the failure sweeping technique to the case of probabilistically noisy geometric primitives.

We believe that our techniques can be adapted to develop parallel algorithms for other computational geometry problems, such as visibility, point-location, etc., in the noisy primitive setting, and more generally to develop parallel divide-and-conquer algorithms where errors are associated with both construction and validation steps.

3.2 Failure Sweeping with Noisy Primitives

We review the concept of failure sweeping in Section 1.4.4. Recall that in this technique, we augmented parallel divide-and-conquer by detecting and recomputing failed subproblems during the combine step. We denote m as the size of a subproblem of the current parent problem of size M. Also, $T(n)$ and $W(n)$ denote the span and work of an algorithm given an input size of n, and $p(n)$ represents the algorithm's probability of failure as a function of its input size. One challenge in designing failure sweep algorithms is in ensuring that m is not too small and not too large compared to M. If m is too small, then many subproblems are likely to fail, forcing us to recompute too many subproblems. If m is too large, then the cost to recompute even a few subproblems may exceed $W(M)$. To solve the latter issue, we introduce a novel technique called *size reduction*. The basic idea is to selectively repeat the

divide step of our parallel divide and conquer algorithm to break large subproblems of size m into sufficiently small problems of total size $O(m)$. We are then able to show that not too many of these smaller subproblems fail, guaranteeing that our total brute-force work is no more than $W(M)$. How small these subproblems must be depends both on $W(M)$, the brute-force algorithm, and $p(m)$.

3.3 Simple, Efficient Noisy Sorting for CREW PRAM

As stated above, while the recent noisy sorting algorithm of Goodrich and Jacob [75] avoids the large constant factors incurred by prior work based on the AKS sorting network, it is quite complex. In this section, we demonstrate our novel size reduction technique by giving a simple and efficient parallel sorting algorithm tolerant to non-persistent comparison errors. Recall that, in this setting, every comparison "is $a < b$?" is independently incorrect with fixed probability $p < 1/2$.

Our algorithm is adapted from the classic SampleSort parallel sorting algorithm. We review the (non-noisy) algorithm in Section 1.4.1. Our first step to make this algorithm noise-tolerant is to replace all operations subject to noise with noise-tolerant replacements.

Lemma 3.1. *We can sort a list A of n values in $O(\log n)$ span and $O(n^2 \log n)$ work w.h.p. in n under noisy comparisons.*

Proof. Initialize an $n \times n$ table T and assign a processor to each index of T in parallel. The processor assigned to index $T[i][j]$ determines whether $A[i] < A[j]$ by applying the trivial repetition strategy. For all n^2 comparisons to be correct w.h.p. in n, each comparison must be repeated $O(\log n)$ times, setting constant factors as appropriate. Then this step takes $O(\log n)$ span and $O(n^2 \log n)$ work. We now have a lookup table of all pairwise comparisons that is correct w.h.p. in n. Thus, subsequent steps are not associated with noise. A series

of prefix operations can sort A using the lookup table in $O(\log n)$ span and $O(n^2)$ work [91]. This correctly sorts A w.h.p. in n. □

The above algorithm sorts a choice of $n^{1/3} - 1$ pivots in $O(\log n)$ span and $O(n^{2/3} \log n) = o(n \log n)$ work w.h.p. in n. Furthermore, any binary search operation can be replaced with the noisy binary search process of Feige, Raghavan, Peleg, and Upfal [61], with parameters tuned such that it succeeds w.h.p. in n. This replacement increases the runtime of binary search by at most a constant factor and therefore also does not affect the asymptotic runtime of the algorithm. As noted in Section 1.2, comparisons on metadata are non-noisy. Hence, once the binary searches retrieve each element's destination bucket, the non-noisy integer sorting method of JáJá [91] is used to pack the elements in their respective buckets in $O(\log n)$ span and $O(n \log n)$ work. Determining whether our distribution step is good, i.e., whether the size of each bucket is sufficiently small, can also be performed without noise.

However, as we described in Section 1.4.4, it is not enough to replace each noisy operation with a noise-tolerant algorithm. As subproblems get smaller, failure probabilities will increase, introducing errors. As a result, we must apply failure sweeping to detect and fix errors over time.

3.3.1 Failure Sweeping and Size Reduction

In this section, we will use failure sweeping to show that we can certify and recompute each subproblem with high probability in the size of its parent problem, "upgrading" failure probabilities as we return from recursion. Let M be the size of the parent subproblem. We first observe that, for the problem of sorting, certifying that all subproblems are correct w.h.p. in M is easy. Just assign a processor to each of the M values and verify that it is less than or equal to the value following it in the list. This can be performed in $O(\log M)$ span and $O(M \log M)$ work via the trivial repetition strategy such that it is correct w.h.p. in M.

The issue arises if we determine that one or more subproblems is incorrect. We can recompute them by brute-force via Lemma 3.1, however recomputing just a single subproblem takes $O((M^{2/3})^2 \log(M)) = O(M^{4/3} \log M)$ work. This exceeds the $O(M \log M)$ work bound needed at each recursive level to achieve $O(n \log n)$ total work. We can apply *size reduction* to fix this issue.

Recall that, in the original algorithm, we used $M^{1/3} - 1$ pivots to induce $M^{1/3}$ subproblems of size $O(M^{2/3})$. Instead of immediately recursing, repeat the divide step to further reduce the size of our child subproblems. For each of the $M^{1/3}$ buckets in parallel, sample an additional $M^{1/3} - 1$ pivots, sort them via Lemma 3.1, and perform the split step again. Runtime is dominated by sorting all pivots in parallel, which takes $O(\log M)$ span and $M^{1/3} \times O(M^{2/3} \log M) = O(M \log M)$ work. A simple extension of Lemma 1.4 shows that each bucket contains $O(M^{1/3})$ elements w.h.p. in M. Then we now have $M^{2/3}$ subproblems of size $O(M^{1/3})$ w.h.p. in M. We recurse on them and tune parameters such that they individually succeed w.h.p. in $M^{1/3}$.

With this modified algorithm, when returning to parent problem M, recomputing a failed subproblem by brute-force now only takes $O((M^{1/3})^2 \log M) = O(M^{2/3} \log M)$ work. Thus, we can now recompute up to $O(M^{1/3})$ failed subproblems in $O(\log M)$ span and $O(M \log M)$ work.

With the addition of failure sweeping, we now have an algorithm that is correct with high probability in n. We are now ready to show that our algorithm remains efficient even with the addition of failure sweeping.

Theorem 3.2. *We can sort n values in $O(\log n)$ span and $O(n \log n)$ work w.h.p. in n under noisy comparisons.*

Proof. We prove this by induction. Consider a parent subproblem of size M. Our inductive hypothesis assumes that we can compute each of its child subproblems of size $O(M^{1/3})$ in $O(\log M^{1/3})$ span, $O(M^{1/3} \log M^{1/3})$ work with failure probability less than $M^{-c'/3}$. Our base case occurs for subproblems of size $O(\log^k n)$ for some fixed $k > 4$ (note the higher bound due to the additional round of splitting). These can be solved w.h.p. in $O(\log^k n)$ via the trivial repetition applied to any existing non-noisy algorithm without affecting asymptotic performance.

To complete the proof, we show that the parent subproblem can be computed with high probability in M in $O(\log M)$ span and $O(M \log M)$ work. From our discussion above, the split step with size reduction and failure sweeping cost $O(\log M)$ span and $O(M \log M)$ work. Then it remains to analyze the cost of brute-force reconstruction. We expect that $M^{2/3}/M^{c'/3} = M^{(2-c')/3}$ child subproblems will fail by our inductive hypothesis. Recall that at most $O(M^{1/3})$ can fail such that we can recompute them all in $O(\log M)$ span and $O(M \log M)$ work.

Letting X be a random variable representing the number of failed subproblems, Markov's inequality states that $\Pr[X \geq M^{1/3}] \leq M^{(2-c')/3}/M^{1/3} = M^{(1-c')/3}$. To be a high-probability bound in M, $(1 - c')/3 < -1$, so $c' > 4$.

Thus, for $c' > 4$, we can solve the parent subproblem of size M in $O(\log M)$ span and $O(M \log M)$ work. To achieve this correctness bound in a given subproblem of size M, all noisy operations (e.g., invocations of noisy binary search, the trivial repetition strategy, etc.) must succeed with probability more than $1 - M^{-c'}$. As a loose bound, we certainly perform no more than $O(M^2)$ noisy operations at a subproblem of size M. Hence, by a union bound, we parameterize individual noisy operations to fail with probability less than M^{-c} for $c = c' + 2$.

To compute the total span and work, we appeal to an analysis by Reif and Sen [123] (see also [124]). In Theorem 2.1 of [123], they state that, if each subproblem at recursive depth i takes T_i parallel steps such that

$$\Pr[T_i \geq k\alpha \log n(\epsilon_0)^i] \leq 2^{-(\epsilon_0)^i \alpha \log n} = (n^{-\alpha})^{(\epsilon_0)^i}$$

for $k, \alpha \in O(1)$ and $0 < \epsilon_0 < 1$, then all subproblems can be completed in $O(\log n)$ parallel steps w.h.p. in n. Our algorithm satisfies this for $\epsilon_0 = 1/3$, $\alpha = 1$, and some constant k proportional to c, hence the span of our algorithm is $O(\log n)$ w.h.p. in n.

Now we give a bound on total work. By construction, total problem size is at most n at all steps since all subproblems at the same recursive level are disjoint. Thus, we can define our algorithm such that at most $O(n)$ work is done per parallel step. In particular, we simply modify brute-force reconstruction to operate in rounds of $M^{1/3}$ failed subproblems rather than all at once. Our analysis above implies that a constant number of rounds is sufficient w.h.p. in M, so we still satisfy Theorem 2.1 of Reif and Sen. Then if our algorithm takes $O(\log n)$ span w.h.p. and performs $O(n)$ operations per parallel step, it must take $O(n \log n)$ work w.h.p.

□

3.4 A 2D Convex Hull Algorithm for CREW PRAM

In this section, we apply size reduction in a slightly different manner to efficiently compute 2D convex hulls when primitives such as orientation tests, sidedness tests, and comparisons are noisy. Our algorithm closely follows the $O(\log n)$-span, $O(n \log n)$-work algorithm of Aggarwal, Chazelle, Guibas, Ó'Dúnlaing [5] and Goodrich and Atallah [74], which we review in Section 1.4.3. Again, we begin by swapping all operations subject to noise with noise-tolerant replacements. The initial sorting operation is replaced with the parallel noisy sorting

algorithm described above. Upper tangents are computed using the application of path-guided pushdown random walks described in Section 2.6.2. The max find operation used to compute the V_j's and W_j's for each upper hull is replaced with the noise-tolerant version described by Feige, Raghavan, Peleg, and Upfal [61] for EREW PRAM. Note that the final parallel prefix operation is not a noisy operation because it only requires us to compare the indices 1 through $\sqrt{n}$ that we assigned each upper hull after sorting. Also, we note that the base case, when $n \leq 2$, can be computed without noisy comparisons as we simply link the two points with an edge.

Each replacement matches the work and span of its non-noisy counterparts. However, they are only correct with high probability in the size of the current subproblem. Thus, we will now develop a failure sweeping and brute-force reconstruction algorithm to detect and fix incorrect subproblems.

3.4.1 Failure Sweeping

Let M be the size of a parent subproblem. Then by definition, it induces $\sqrt{M}$ subproblems of size $\sqrt{M}$.

Consider one such child subproblem. To perform failure sweeping, we can do the following. Consider each point in parallel. If the point is a hull point, perform the trivial repetition strategy on an orientation test with its neighbors to verify they are convex. If the point is not a hull point, perform a noisy binary search to determine which hull points it lies between. Then perform the trivial repetition strategy on a sidedness test to determine whether it lies above or below the hull. See Figure 3.1. It takes $O(\log M)$ span and $O(\sqrt{M} \log M)$ work to certify a single child problem w.h.p. in M. Certifying all takes $O(\log M)$ span and $O(M \log M)$ work.

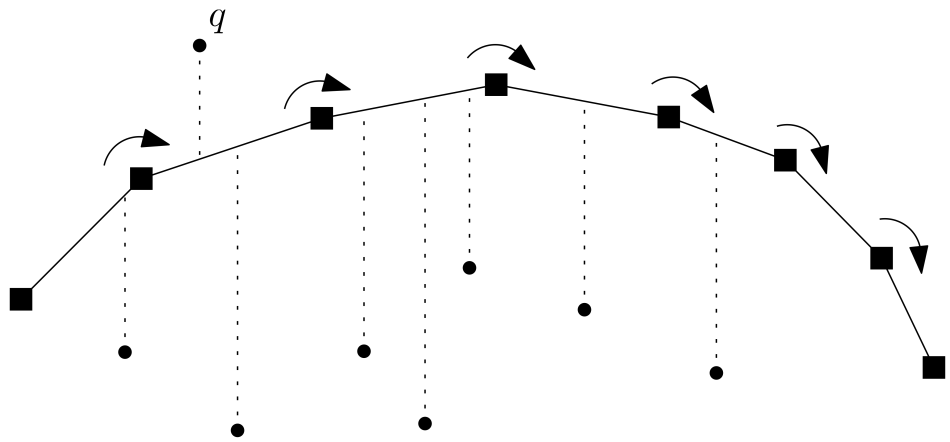

Figure 3.1: Failure sweeping an upper hull. To verify that each (square) hull point is correct, we use the trivial repetition strategy to determine that it is convex with respect to its neighbors. For non-hull points, we perform a noisy binary search on the hull to determine whether they are above or below it. This is true for point q, so this subproblem is flagged for reconstruction.

Now we consider brute-force reconstruction. Our brute-force algorithm is a parallelization of the classic Jarvis march convex hull algorithm [92]. To describe it, we first prove the following technical lemma.

Lemma 3.3. *Consider a point set P and a point $p \in P$. Let $q, r \in P$ be the counterclockwise and clockwise neighbors of p w.r.t. a coordinate system centered at p. If $[q, p, r]$ makes a right-hand turn, then p is an upper hull point.*

Proof. See Figure 3.2. We first describe a forbidden region defined by $\{q, p, r\}$. By definition of CCW and CW neighbors, no points in P exist in the region to the left of $\overrightarrow{pr}$ and the right of the vertical line through p. Likewise, no points in P exist in the region to the right of $\overrightarrow{pq}$ and the left of the vertical line through p.

By the fact that $[q, p, r]$ is a right-hand turn and by definition of the forbidden region, all points in $P \setminus \{q, p\}$ are to one side of the open halfplane extending $\overline{qp}$. The same is true for $\overline{pr}$, meaning that both are convex hull edges of P. Then p must be a hull point.

Now we show that p is an upper hull point. If p is the leftmost or rightmost point, we are done. Otherwise, assume towards a contradiction that p is a lower hull point yet $[q, p, r]$ makes a right-hand turn. Let L and R be the leftmost and rightmost points of P. Then p lies below $\overline{LR}$, so $[L, p, R]$ makes a left-hand turn. But by definition of CCW and CW neighbors, $[q, p, r]$ makes the same or a tighter angle. Then it makes a left-hand turn, which is a contradiction. □

Lemma 3.4. *We can compute the 2D upper convex hull of a set of n points pre-sorted by their x coordinate under noisy primitives in $O(\log n)$ span and $O(n^2 \log n)$ work w.h.p. in n.*

Proof. Assign $n-1$ processors to each point p and perform two noisy max-finds to determine q and r. This takes $O(\log n)$ span and $O(n^2 \log n)$ work over all n points. Then, for each point, apply the trivial repetition strategy to determine whether $[q, p, r]$ makes a right-hand turn in $O(\log n)$ span and $O(n \log n)$ total work w.h.p. in n. A final prefix operation recovers the hull points. □

Unfortunately, this brute-force algorithm is too expensive. If each child subproblem is of size $\sqrt{M}$, then recomputing just one already takes $O(M \log M)$ time. Then if $\omega(1)$ child subproblems fail, we exceed our work bound for the parent subproblem. We now show how to apply size reduction in a different capacity to produce a more efficient brute-force algorithm.

Lemma 3.5. *We can find the 2D upper convex hull of a set of n points pre-sorted by their x coordinate under noisy primitives in $O(\log n)$ span and $O(n^{3/2} \log n)$ work w.h.p. in n.*

Proof. Divide our n points into $\sqrt{n}$ groups of size $\sqrt{n}$. Compute the upper hull of each group by Lemma 3.4 in span $O(\log n)$ and work $O(\sqrt{n} \times (\sqrt{n})^2 \log n) = O(n^{3/2} \log n)$ such that each is successfully constructed w.h.p. in n. Merge each group in $O(\log n)$ span and

$O(n \log n)$ work w.h.p. in n using a noise-tolerant version of the original algorithm's merge step [74]. □

Now if a child subproblem fails, it only takes $O((\sqrt{M})^{3/2} \log M) = O(M^{3/4} \log M)$ work to recompute it. Thus, at most $O(M^{1/4})$ child subproblems can fail such that we can recompute them all in $O(\log M)$ span and $O(M \log M)$ work.

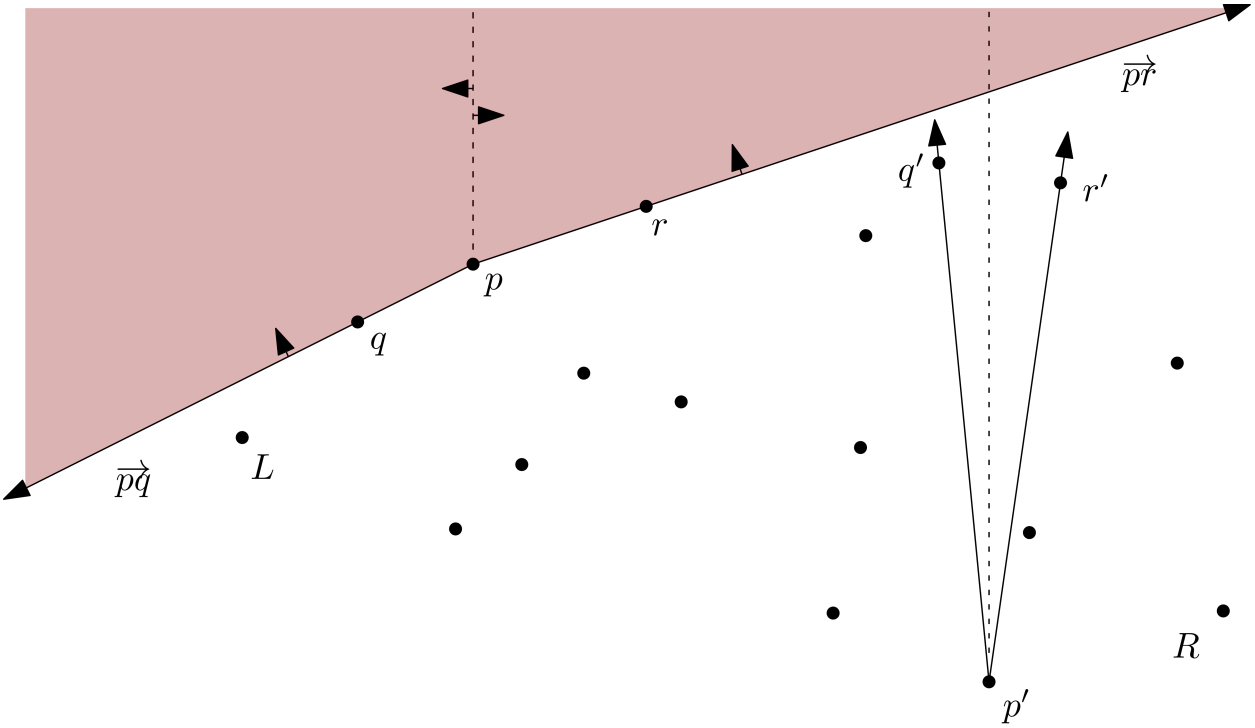

Figure 3.2: The forbidden region defined by p's CCW and CW neighbors q and r as described in Lemma 3.3. We also depict lower hull point p' with left-hand turn $[q', p', r']$.

Theorem 3.6. *We can compute the convex hull of n points in the plane in $O(\log n)$ span and $O(n \log n)$ work w.h.p. in n under noisy primitive operations.*

Proof. Again, we prove the theorem by induction. Consider a parent subproblem of size M. Our inductive hypothesis assumes that we can compute each of its child subproblems of size $\sqrt{M}$ in $O(\log \sqrt{M})$ span and $O(\sqrt{M} \log \sqrt{M})$ work with failure probability less than $M^{-c'/2}$. The base case holds since no noisy comparisons are required to solve a subproblem of size $n = 2$.

As in Theorem 3.2, we now show that the parent subproblem can be computed with high probability in M in $O(\log M)$ span and $O(M \log M)$ work. By our analysis above, we know that the original algorithm and failure sweeping step can be performed in $O(\log M)$ span and $O(M \log M)$ w.h.p. in M. It remains to analyze the cost of brute-force construction. Recall that at most $O(M^{1/4})$ subproblems can fail such that we can recompute them all efficiently. We expect $M^{1/2}/M^{c'/2} = M^{(1-c')/2}$ to fail by our inductive hypothesis.

Letting X be a random variable representing the number of failed subproblems, Markov's inequality states that $\Pr[X \geq M^{1/4}] \leq M^{(1-c')/2}/M^{1/4} = M^{(1-2c')/4}$. To be a high-probability bound in M, $(1-2c')/4 < -1$, so $c' > 5/2$. Thus, for $c' > 5/2$, we can solve the parent subproblem of size M in $O(\log M)$ span and $O(M \log M)$ work w.h.p. in M. We observe that certainly no more than $O(M^2)$ noisy operations are performed at each subproblem, so each noisy operation can be parameterized with failure probability less than M^{-c}, for $c = c' + 2$.

As we did in Theorem 3.2, we can appeal to Reif and Sen's Theorem 2.1 for $\epsilon_0 = 1/2$, $\alpha = 1$, and k dependent on c [123, 124]. Once again, because total problem size at any level of recursion is n, we can easily define the algorithm to operate using $O(n)$ processors by restricting brute-force recomputation to operate in rounds of $M^{1/4}$ failed subproblems. This implies that our algorithm runs in $O(\log n)$ span and $O(n \log n)$ work w.h.p. in n. □

3.5 A 3D Convex Hull Algorithm for CREW PRAM

See Section 1.4.3 for a review of the non-noisy algorithm of Reif and Sen [123]. As before, there are three steps to our analysis. We first show how to adapt all operations of the original algorithm such that they are noise-tolerant w.r.t. their input size. Then we show how to efficiently perform failure-sweeping and, lastly, brute-force reconstruction.

From an application of the trivial repetition strategy, we immediately have the following algorithms to compute a halfspace intersection. Respectively, these are used to compute the halfspace intersection of our sample R and to directly solve sufficiently small subproblems.

Lemma 3.7. *The intersection of a set of n halfspaces in 3D can be computed in $O(\log n)$ span and $O(n^4 \log n)$ work with high probability in n under noisy comparisons.*

Proof. This follows from Lemma 2.2 of Reif and Sen [123]. There are $O(n^3)$ candidate vertices in a 3D arrangement and $O(n)$ halfspaces, so there are $O(n^4)$ vertex-halfspace pairs. For each vertex-halfspace pair, we can determine whether the vertex satisfies the halfspace's equation in a single noisy operation. For each vertex, we perform a parallel prefix operation to compute the conjunction of the $O(n)$ results.

There are $O(n^4)$ pairs, so testing each with the trivial repetition strategy takes $O(n^4 \log n)$ work. Each computation can be done in parallel, so this step takes $O(\log n)$ span. The final prefix operations done at each vertex are not noisy. They take $O(\log n)$ span and $O(n^4)$ work overall.

We are left with a set of vertices in the arrangement of the given halfspaces that satisfy every halfspace. This means these vertices must bound the halfspace intersection. An application of noisy sorting [75] can be used to compute an adjacency structure for the polytope from this set in $O(\log n)$ span and $O(n \log n)$ work [123].

We perform $O(n^4)$ noisy operations in this step. If each individually fails with probability no more than $1/n^c$ for some constant c, then by a union bound at least one fails with probability $1/n^{c-4}$. Therefore, to guarantee a failure probability of at most $1/n^{c'}$, we set $c > 5 + c'$. This only affects constant factors in the work and span of these operations and so does not change their overall complexity. □

Lemma 3.8. *The intersection of a set of n halfspaces can be computed in $O(\log^4 n)$ span and $O(n \log^4 n)$ work with high probability in n under noisy comparisons.*

Proof. This follows from the $O(\log^3 n)$-span, $O(n \log^3 n)$-work algorithm of Aggarwal, Chazelle, Guibas, Ó'Dúnlaing, and Yap [5] and the trivial repetition strategy applied to every noisy primitive. □

We next observe that Reif and Sen's *polling* method [123] in and of itself is not subject to noise as randomly selecting halfspaces requires no primitive operations. However, determining whether a sample is good does involve noisy operations. We now show that computing $\mathcal{R}$, the halfspace intersection of R, and the point-location structure that determines what cones a halfspace in $H \setminus R$ intersects can be performed in $O(\log n)$ span and $O(n \log n)$ work w.h.p.

Lemma 3.9. *Given a point o in the halfspace intersection of H, we can compute the halfspace intersection of R and its cones using $O(|R|^4 \log n)$ work and $O(\log n)$ span under noisy comparisons w.h.p.*

Proof. This follows from Lemma 3.7. The adjacency structure of R that we compute is correct with high probability in n. Because we have computed this structure, any topological changes we make to produce the cones are not noisy. Thus, we can partition $\mathcal{R}$ with planes such that each face is a trapezoid and partition each trapezoid further into a triangle, and draw edges between each face and our origin o as Reif and Sen describe [123] without noise. □

Lemma 3.10. *We can preprocess n planes into an arrangement such that, given an arbitrary query point, we can report the cell in which the point is contained. Preprocessing can be done under noisy comparisons in $O(n^7 \log n)$ work and $O(\log n)$ span w.h.p. using $O(n^7)$ noisy operations. Queries can be performed sequentially in $O(\log n)$ time w.h.p. using $O(1)$ noisy operations.*

Proof. This follows from Reif and Sen [123, Lemma 5.1] as well as by the work of Dobkin and Lipton [52]. Each preprocessing step involves projection of points down to a lower dimension and a constant number of parallel sorting steps per cell. Projection is not a noisy operation as we are manipulating a geometric object's equations, which are its labels. Each sorting step can be swapped with our noisy sorting algorithm. Likewise, each query involves a constant number of binary search steps, which can be replaced with the noisy binary search of Feige *et al.* [61]. □

Corollary 3.11. *We can compute a preprocessing structure from $\mathcal{R}$ such that, given a query halfspace h, we can determine which cones in $\mathcal{R}$ h stabs. Preprocessing can be performed in $O(|R|^8 \log n)$ work and $O(\log n)$ span w.h.p. using $O(|R|^8)$ noisy operations. Queries can be performed sequentially in $O(\log n)$ time w.h.p. using $O(1)$ noisy operations.*

Proof. This follows from our Lemma 3.10 and [123, Lemma 5.2] applied to the dual of each vertex of $\mathcal{R}$. There are $O(|R|^7)$ cells of our arrangement and $O(|R|)$ cones in $\mathcal{R}$. We have each of the $O(|R|^8)$ cone-cell pairs compute whether the cell overlaps the cone in $O(1)$ noisy comparisons. $O(|R|^7)$ parallel prefix operations can be used to compile a list of cones each cell is incident to.

Testing each cone-cell pair takes $O(|R|^8 \log n)$ work and $O(\log n)$ span via the trivial repetition strategy. The parallel prefix operations take $O(|R|^8)$ work and $O(\log |R|) = O(\log n)$ span.

As described in the previous lemma, each query requires a constant number of noisy binary searches and can be accomplished in $O(\log n)$ time sequentially w.h.p. □

Setting $|R| \leq n^{1/8}$ allows us to perform the previous steps in $O(\log n)$ span and $O(n \log n)$ work w.h.p. Once the structure is built, we perform one query per halfspace $h \in H \setminus R$ to determine which cones it stabs. All queries can be done in parallel and there are $O(n)$

halfspaces, so this step takes $O(\log n)$ span and $O(n \log n)$ work. Having done this work, we can use this data structure to determine if our current sample R is "good", i.e. if it distributes halfplanes roughly evenly between cones [42, 123].

3.5.1 Pruning halfspaces

To ensure total problem size among subproblems at a given level of recursion does not exceed some constant multiple of our initial problem size, we rely on a method of Amato, Goodrich, and Ramos [9].

Lemma 3.12. *In a problem size of n, where H is the total set of halfplanes, we can prune enough redundant halfspaces such that total problem size among all subproblems at this level is at most $O(n_0)$, where n_0 is the initial problem size. This can be done in $O(\log n)$ span and $O(n \log n)$ work w.h.p. in n using $O(n)$ noisy operations.*

Proof. This follows from the pruning strategy and corresponding lemmas described in [9, Section 4.1.1]. Their solution first involves projecting the halfspaces associated with C_i onto the three faces of the cone, which is not a noisy operation. Then they compute the 2D convex hull of each projection, which we can do in $O(\log n)$ span and $O(n \log n)$ work w.h.p. in n using our construction in Section 3.4. They call these convex chains *contours*.

They consider separately the halfspaces that contribute an edge to a contour and those that do not. We denote the set of halfspaces that stab C_i but do not contribute to any of its contours $H^{nc}_{|C_i}$. Let H^{nc} be the union of all such halfspaces over all cones. The authors show that each halfspace $h \in H^{nc}$ may contribute a vertex to the halfspace intersection of at most one cone [9, Lemma 4.2]. They prove that this halfspace h can be found by locating the closest point p on each of the three contours of C_i and shooting a ray from each p through C_i towards h. If all three such rays pierce h without being stopped first by their respective

halfspaces h', h'' that contributed p to the contour, then h may contribute a vertex to the halfspace intersection of H.

Each halfspace $h \in H^{nc}_{|C_i}$ can determine whether it should be pruned in $O(\log n)$ time w.h.p. by projecting itself onto each face and performing a noisy binary searches on each contour to determine the closest hull point to the projected line representing h (recall that we maintain the upper and lower hulls of each contour as a binary search tree, something we took advantage of to perform failure sweeping in Section 3.4.1). We can associate each contour point p with the two halfspaces that define it so that they can be recovered in constant time. We can then perform a constant number of noisy comparisons using the trivial repetition strategy to determine which halfspace the ray passes through first in $O(\log n)$ time w.h.p. By the fact that our sample R is good, $|H^{nc}| = O(n)$. Then we can perform each test in parallel in $O(\log n)$ span and $O(n \log n)$ work.

Now we consider the halfspaces that contribute to contours, which we denote as $H^{c}_{|C_i}$. Amato, Goodrich, and Ramos [9] show that, if a halfspace in $h \in H^{c}_{|C_i}$ contributes an edge to at least two of the contours, then it can be pruned. To determine which halfspaces should be pruned, we simply label each contour edge with the corresponding halfspace that generated it, which can be done when constructing the contours, and sort the labels lexicographically. Again, by our use of a good sample, the total number of contour halfspaces $|H^c| = O(n)$, thus we can perform noisy sorting in parallel over all C_i in $O(\log n)$ span and $O(n \log n)$ work. In the lexicographically sorted array, if a halfspace is adjacent to itself, then it must contribute an edge to at least two contours, and so should be pruned. In [9], the authors show that after pruning, the set of all contour halfspaces across all cones and all subproblems also has size $O(n_0)$. □

Finally, to complete the halfspace intersection before returning from a subproblem, Amato, Goodrich, and Ramos note that we must add back a certain subset of halfspaces in $H^{c}_{C_i}$

that contributed an edge to the halfspace intersection but not a vertex. Fortunately, they show that at most three such halfspaces exist and can be identified when pruning. We can incorporate these halfspaces like so.

Let $\mathcal{C}_i$ be the halfspace intersection of cone C_i. Process each halfspace one at a time. Have each vertex in $\mathcal{C}_i$ check if it satisfies the halfspace's equation in $O(\log n)$ time using the trivial repetition strategy. Label the vertex based one whether it succeeded or failed. Then examine each edge. For edges that have one vertex included in h and the other vertex not included in h, compute the intersection between h and the edge to produce a new set of vertices that are incident to h.

For each of the three halfspaces processed, this takes $O(\log n)$ span and $O(|\mathcal{C}_i| \log n)$ work. Because 3D hulls have complexity linear in their input and $|H^{nc}|, |H^c| = O(n)$, this takes $O(n \log n)$ work over all cones. Because each operation is noisy, pruning is successful w.h.p. in n rather than deterministically. We can verify that pruning is correct via a non-noisy prefix operation to compute total size and redo pruning if total size exceeds some constant multiple of current problem size. It can be shown via the same analysis as polling that this does not increase span or work asymptotically.

We conclude that each operation in a non-noisy 3D hull algorithm can be replaced with noisy operations without increasing the asymptotic complexity of each subproblem. We observe that, for a subproblem of size n, all operations discussed prior use at most $O(n)$ noisy operations, including sorting, searching, and trivial repetitions.

3.5.2 Failure Sweeping

We have shown that each step of Reif and Sen's algorithm [123], with Amato, Goodrich, and Ramos' pruning method [9], can be replaced with noise-tolerant versions that individually succeed with high probability in the current problem size. However, as we did for 2D convex

hulls, we must negotiate the fact that our success probabilities get weaker as problem size gets smaller. To compensate for this, we once again perform failure sweeping. Unlike for 2D convex hulls, subproblems may not be evenly sized. We will denote m as the size of the problem at recursive level l and M the size of its parent subproblem at recursive level $l-1$.

It will be simpler to return to the primal problem: given a 3D convex hull, verify that all points on the hull are convex with respect to their neighbors and all other points lie within the hull. To do the former, assign a processor to each of the edges of the hull and verify that the two adjacent faces to each edge are convex. This takes $O(\log M)$ span and $O(m \log M)$ work and succeeds w.h.p. in M by the trivial repetition strategy. Now we perform the latter task. When we performed failure sweeping on a 2D hull, we were able to quickly determine which edge of the hull lies directly above each point through noisy binary search on the tree-based structure that represented the hull. In this algorithm, however, no equivalent structure is built. As a result, we will show how to reduce the problem to planar point location (PPL) and then describe an algorithm that can construct a PPL data structure efficiently under noisy comparisons.

Consider a coordinate system in which the origin lies outside of the hull, and consider the plane p formed by the x- and y-axes. Let the "upper hull" be the portion of the hull that can see this plane and the "lower hull" be the portion that cannot. This test can be done via a constant number of primitive operations per face, and so can be performed in parallel in $O(\log M)$ span and $O(m \log M)$ work via the trivial repetition strategy.

We focus on the upper hull, as the case for the lower hull is symmetric. We can project each point of the upper hull simply by ignoring its z-coordinate. By general position, each face of our hull is a triangle. Therefore, this projection is a triangulation of a convex polygon. We can use this projection to determine if some non-hull point u lies below the upper hull. First, we project u onto p, creating the projected point u^*. We use planar point-location to determine the triangle Δ^* it appears in. This triangle maps to a face of the upper hull, Δ,

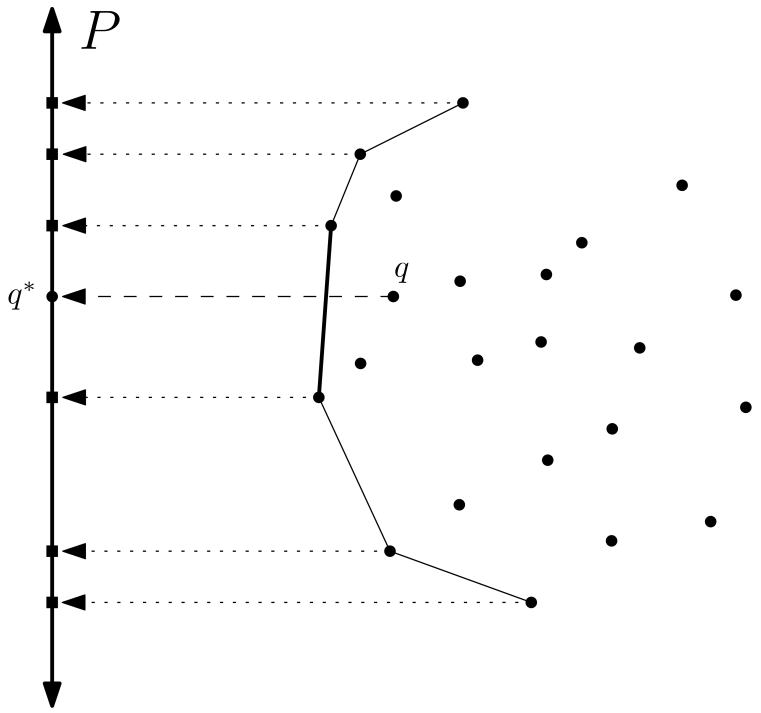

Figure 3.3: A 2D representation of our failure sweeping algorithm. We define a coordinate system with its origin outside of the given hull and let P be the y-axis (the x-y plane in 3D). We define the "upper" hull as the hull points that can see P. We then perform an orthogonal projection of the upper hull points onto P, inducing a set of intervals. In 3D, this would induce a triangulated polygon onto P. We can project a query point q onto P and determine what region of P q^* lies in. This tells us what portion of the hull $\overrightarrow{qq^*}$ stabs. In 2D, this requires a single binary search. However, in 3D we require a planar point-location structure to determine the triangle q^* lies in.

that intersects the ray $\overrightarrow{uu^*}$. Next, we simply perform an orientation test between u and Δ to determine if u is inside or outside of the upper hull. We can repeat this point location test symmetrically on the lower hull. If any u fails either test, then we conclude that our convex hull is invalid.

3.5.3 Constructing a Planar Point-Location Data Structure on a Triangulated Convex Polygon

While the projections done to produce our triangulated convex polygon on plane p are without noise, the work needed to construct the planar point location data structure does involve noisy primitive operations.

To build a PPL data structure, we modify the randomized parallel algorithm of Reif and Sen [124] that constructs a Kirkpatrick decomposition [98] of a triangulated polygon of n points taking $O(\log n)$ span and $O(n \log n)$ work with high probability in n. Their basic strategy is to, at each step, have each vertex of degree $\leq d = O(1)$ independently flip a coin. Vertices that flip heads and have no neighbors that also flip heads are pruned. The gaps left behind are re-triangulated. While constructing the decomposition, Reif and Sen at the same time construct a point-location search DAG. Each triangle is represented by a node in the DAG. Triangles destroyed in a given iteration have a constant number of new triangles that overlap them. Nodes for each new triangle are given a pointer to every triangle destroyed at this step that they overlap with. Each of these operations takes $O(1)$ time in the non-noisy setting.

They prove that, over $O(\log n)$ steps, we remove a constant fraction of vertices at each step such that the last level of the decomposition is of size $O(\log n)$ with high probability in n. Because there are $O(\log n)$ steps, each takes $O(1)$ time per point, and there are $O(n)$ points, the algorithm takes $O(n \log n)$ work and $O(\log n)$ span. They also observe that the depth of the resulting search structure is $O(\log n)$ [98, 124], allowing for $O(\log n)$-time sequential queries.

To convert this into a noise-tolerant algorithm, we begin by discussing how to introduce a bounding triangle to our triangulated convex polygon. We introduce our bounding points a, b, c. To triangulate the boundary of our polygon, we assign each hull point of our polygon

a processor. We consider one bounding point at a time. Using $O(1)$ noisy orientation tests, each point on our polygon's hull can determine whether it "sees" bounding point a, and if so can draw an edge to it. It is easy to see that none of these edges cross. This takes $O(\log n)$ time per point using the trivial repetition strategy.

Now we consider b. Say that point p already has an edge to a. If p's neighbor in the direction of b also has an edge to c, then p should not draw an edge to b as it will cross its neighbor's edge to a. Otherwise, p draws an edge to b. Again, this takes $O(\log n)$ time per point using the trivial repetition strategy. Incorporating c involves a similar process. There are $O(n)$ points p and each can be processed independently at each of the three steps, so this takes $O(\log n)$ span and $O(n \log n)$ work.

After this, at each iteration, we have three steps: (1) sampling the independent set, (2) re-triangulating polygons, and (3) updating the search DAG. We first observe that the sampling step is without noise as we are just generating, reading, and comparing labels on points. For the re-triangulating step, we can remove noise via an initial noisy sorting computation on the x-coordinates of all $O(n)$ points in $O(\log n)$ span and $O(n \log n)$ work. If we label each point with its position in the sorted order, a single processor can re-triangulate a polygon of constant size in $O(1)$ time by comparing each point's labels with the others to determine in what order to sweep the polygon.

Lastly, we have to update the search DAG, which requires us to determine whether a destroyed triangle intersects with a constant number of new triangles. Because the triangles are created throughout the algorithm, it does not seem that we can preprocess the points to remove noise from these operations. Thus, if we were to construct the DAG as we compute the decomposition, the trivial repetition strategy would require that each of the $O(\log n)$ steps take $O(\log n)$ span. Instead, we apply a two-pass system.

On the first pass, construct the Kirkpatrick decomposition and create a dummy node for each triangle. We can associate each dummy node with a triangle of the decomposition and with the triangles that replaced it with a constant number of links per node. On the second pass, consider each triangle and their corresponding DAG node in parallel. For a triangle destroyed at step i, it determines which of the at most $d = O(1)$ triangles created in the corresponding gap at step i intersect it. This requires $O(1)$ primitive operations, and so can be performed in $O(\log n)$ time using the trivial repetition strategy. Each triangle then draws edges from the new triangles that overlap it to itself, creating our search structure. This takes $O(\log n)$ time per triangle and there are at most $O(n)$ triangles in the search structure [98], so this takes $O(\log n)$ span and $O(n \log n)$ work.

3.5.4 Querying the Planar Point-Location Data Structure Efficiently

As we described earlier, Reif and Sen's algorithm stops when the decomposition is of size $O(\log n)$ [124]. This is sufficient in the non-noisy case as a query point can simply brute-force search through all triangles at this first level in $O(\log n)$ time to determine which node of the DAG it should start at. Here, however, the trivial repetition strategy would add an extra log factor to this brute-force search. The following lemma will help us construct another point-location structure to help us find the initial triangle in $O(\log n)$ time w.h.p. in n. One can think of this approach as a brute-force parallelization of the planar point location methods of Cole [43] and Sarnak and Tarjan [126].

Lemma 3.13. *Consider a planar subdivision of size $r \leq n$. Using $O(\log n)$ span, $O(r^3 \log n)$ work, and $O(r^2)$ space, we can construct a point-location data structure with high probability in n that allows for $O(\log n)$-time queries w.h.p. under noisy primitives. Construction requires $O(r^3)$ noisy operations and querying requires $O(1)$ noisy operations.*

Proof. We begin by drawing a vertical line at each point, subdividing the plane into $r + 1$ slabs. Each slab has at most $O(r)$ regions in it, so there are at most $O(r^2)$ regions. There were $O(r)$ initial triangles, so there are $O(r^3)$ triangle-region pairs. For each pair, we can determine whether the triangle and region intersect. This requires $O(1)$ noisy comparisons per pair. A parallel prefix operation can be used to combine all $O(r)$ outcomes for each region to determine the unique triangle that corresponds to it. In total, this takes $O(r^3 \log n)$ work and $O(\log n)$ span for each operation to succeed w.h.p. in n.

We can sort the slabs by their x-coordinate in $O(r \log n)$ work and $O(\log n)$ span w.h.p. in n. We can then sort each slab's regions vertically in $O(r^2 \log n)$ total work and $O(\log n)$ span w.h.p. in n.

Given a query point q, we can use two noisy binary searches to locate the region in which q lies. The first determines the appropriate slab and the second determines the appropriate region within the slab. □

In our case, $r = O(\log n)$, so total work is $o(n \log n)$ and extra space usage is $o(n)$. Lastly, it remains to show how a processor can navigate the search DAG in $O(\log n)$ time w.h.p. in n. To do this, we will construct a transition oracle for the Kirkpatrick decompositon DAG such that we can apply path-guided pushdown random walks. First, we observe that a given query point must have a unique valid path in the Kirkpatrick decomposition. Level i of the DAG corresponds to triangulation G_i of the decomposition. By definition of a triangulation, no triangles overlap, meaning that there is a unique triangle at level i that corresponds to q. Therefore, there is a unique node at level i of the DAG that is valid for q. This is true at all levels of the DAG, so each q has a single valid path through the DAG.

It also follows that we can determine whether we are on the correct path using a single noisy primitive. We simply check whether q is located inside the triangle represented by our current node in the DAG. Because the Kirkpatrick decomposition search DAG satisfies the

requirements of our technique, a query point can navigate the DAG in $O(\log n)$ time w.h.p. in n by instantiating path-guided pushdown random walks. We conclude the following:

Lemma 3.14. *Given a triangulation of n points in the plane, we can construct a point-location data structure to determine which triangle a query point lies in under noisy primitives. It takes $O(\log n)$ time to construct the data structure and $O(\log n)$ time to query it w.h.p. in n.*

Corollary 3.15. *We can perform failure sweeping on a halfspace intersection of m points in $O(m \log M)$ work and $O(\log M)$ span w.h.p. in M.*

Proof. This follows from Lemma 3.14 and the above discussion. Using the point-location data structure instantiated with failure probability w.h.p. in M, we can project each point q to the xy-plane, and determine which triangle the ray $\overrightarrow{qq^*}$ stabs in $O(\log M)$ time per point. From there we can perform one noisy orientation test to determine if q lies inside or outside the hull, which takes $O(\log M)$ time using the trivial repetition strategy. There are $O(m)$ points, so this takes $O(m \log M)$ work and $O(\log M)$ span. □

Theorem 3.16. *Any parent subproblem of size M can failure sweep all of its child subproblems in $O(M \log M)$ work and $O(\log M)$ span w.h.p. in M.*

Proof. This follows from above done in parallel over all subproblems. By Lemma 3.12, the total size of all subproblems is $O(M)$. □

Now, we discuss brute-force construction and size reduction. We will see that we must apply *both* the size reduction techniques discussed in the previous sections and make non-trivial modifications to Reif and Sen's original algorithm [123] in order to efficiently implement brute-force reconstruction.

3.5.5 Efficient Brute-Force Reconstruction via Size Reduction

Attempting to bound the number of failed subproblems as we did in previous sections presents a problem. Recall from Reif and Sen's original work [123] as well as Clarkson [42], if the problem at recursive level $l-1$ is size M, then its child problems at recursive level l could be size at most $O(M^{7/8}\log M)$. Indeed, if we were to apply Lemma 3.7 to recompute just a single subproblem by brute-force, it would take $O(M^{7/2}\log^5 M)$ work to solve it w.h.p. in M. To solve this issue, we first apply size reduction as we did in Section 3.4 to produce a more efficient brute-force algorithm.

Lemma 3.17. *The intersection of a set of n halfspaces in 3D can be computed in $O(\log n)$ span and $O(n^{3/2}\log n)$ work with high probability in n under noisy comparisons.*

Proof. Here, we apply size reduction. That is, we selectively apply the divide step of Reif and Sen's original algorithm [123] in order to reduce problem size. Recall that, in their algorithm, we first sample a set of $n^{1/8}$ halfspaces, compute their intersection, and compute a point-location structure for each facet of the solution. In our setting, this is computed in span $O(\log n)$ and work $O((n^{1/8})^8\log n) = O(n\log n)$ w.h.p. in n. Furthermore, this induces $O(n^{1/8})$ subproblems of size at most $O(n^{7/8}\log n)$ w.h.p. in n.

However, using this sample size is too expensive to achieve the stated bounds. Instead, consider a sample size of $O(n^{1/16})$, for which we can solve and compute a point-location structure in $O(\log n)$ span and $O(\sqrt{n}\log n)$ work.

Doing this for one step splits our original problem of size n into $O(n^{1/16})$ subproblems of size $O(n^{15/16}\log n)$. Just one step has not made much of a difference. Nevertheless, for any subproblem of size $\omega(n^{1/8})$, we can repeat this process with the same $O(n^{1/16})$ sample size. Each split step shaves off at least a $(n^{1/16}/\log n)$-factor from the max problem size. After at most 15 iterations of size reduction, all subproblems are of size $O(n^{1/16}\log^{O(1)} n) = O(n^{1/8})$ w.h.p. (with constants adjusted to accommodate a union bound over all $O(n)$ split

operations). After this, we compute all subproblems by brute-force using Lemma 3.7 and merge them.

We now analyze our brute-force algorithm's cost. Each split costs $O(\log n)$ span and $O(\sqrt{n}\log n)$ work. After 15 levels, we certainly induce no more than $O(n)$ subproblems, so the total cost of splitting is at most $15 \times O(\log n) = O(\log n)$ span and $15 \times O(n \times \sqrt{n}\log n) = O(n^{3/2}\log n)$ work.

There are $O(n^{15/16}) = O(n)$ subproblems at the bottom level that must be solved by brute-force, each of size $O(n^{1/8})$. This takes $O(\log n)$ span and $O(n \times (n^{1/8})^4 \log n) = O(n^{3/2}\log n)$ work by Lemma 3.7. We recombine all solved subproblems at once via an application of noisy sorting in $O(\log n)$ span and $O(n \log n)$ work w.h.p. [123]. □

This improved brute-force algorithm, however, is still not enough. If our child subproblems are size $O(M^{7/8}\log M)$, recomputing just one still takes $\omega(M \log M)$ work.

Thus, we also apply size reduction to reduce subproblem size, similar to Section 3.3.

Lemma 3.18. *We can split a problem of size n into at most $O(n^{9/16})$ subproblems of size $O(\sqrt{n})$ in $O(\log n)$ span and $O(n \log n)$ work. Total problem size remains $O(n)$.*

Proof. We apply a similar size reduction process as in Lemma 3.17 using a sample size of $O(n^{1/16})$ at each step. However, descending the full 15 levels would take work $O(n^{3/2}\log n)$. Thus, we stop after just nine levels. This induces at most $O(n^{9/16})$ subproblems of size at most $O(\sqrt{n})$. In the last level of splitting, there are at most $O(n^{8/16}) = O(\sqrt{n})$ subproblems. The total cost of splitting is dominated by the splitting step of this last level, which takes $O(\log n)$ span and $O(\sqrt{n} \times \sqrt{n}\log n) = O(n \log n)$ work.

By the pruning steps of the original algorithm [123], total problem size remains $O(n)$ across all induced subproblems. □

Thus, we have applied size reduction twice: first to give an efficient brute-force method and second to reduce child subproblems to size $O(\sqrt{M})$. Once we have sufficiently reduced the size of our child subproblems, we recurse into them, tuning parameters such that they succeed w.h.p. in $\sqrt{M}$. As in our 2D convex hull algorithm, if all subproblems are size $O(\sqrt{M})$ and our brute-force method takes work $O(M^{3/2} \log M)$, then at most $O(M^{1/4})$ subproblems can fail such that we can recompute all in $O(\log M)$ span and $O(M \log M)$ work w.h.p. in M.

Theorem 3.19. *We can compute the intersection of n halfspaces in 3D in $O(\log n)$ span and $O(n \log n)$ work w.h.p. in n, even with noisy primitive operations.*

Proof. We prove this by induction. Let M be the size of the current subproblem. Our inductive hypothesis assumes that each subproblem can be computed in span $O(\log \sqrt{M})$ and work $O(\sqrt{M} \log \sqrt{M})$ and fails with probability less than $M^{-c'/2}$.

The base case follows from our discussion above. All problems of size smaller than $O(\log^k n)$ for some constant k can be solved by Lemma 3.8 without affecting asymptotic runtime of the algorithm.

It remains to show that we can solve the parent subproblem in $O(\log M)$ span and $O(M \log M)$ work w.h.p. From our previous discussion, the splitting, size reduction, failure sweep, and recombination steps all occur in $O(\log M)$ span and $O(M \log M)$ work. Using the same analysis as in previous sections, no more than $O(M^{1/4})$ child subproblems fail w.h.p. in M for $c' > 21/8$. Then we can tune parameters such that brute-force reconstruction takes span $O(\log M)$ and work $O(M \log M)$ w.h.p. in M.

Applying Theorem 2.1 of Reif and Sen shows that our algorithm takes span $O(\log n)$ w.h.p. in n [123, 124]. By pruning and the arguments used in previous sections, the algorithm takes $O(n \log n)$ work w.h.p.

□

Chapter 4

Simple Entropy-Sensitive Strictly In-Place Sorting Algorithms

4.1 Background

Embedded computer systems often have real-time performance requirements, yet their design usually has severe limits on the amount of memory available. This provides a unique challenge for algorithm design, motivating the in-place model of computation. Recall that an algorithm is strictly in-place if it allocates $O(1)$ words of extra memory aside from the input. Furthermore, fast performance requirements motivate the study of beyond worst-case analysis of algorithms, i.e., the study of algorithms whose running time can beat worst-case bounds depending on parameters or metrics defined for problem instances that capture typical properties for expected inputs. For example, recall the definition of *run-based entropy* from Section 1.4.5. We are given an array, $A = [x_1, x_2, \ldots, x_n]$, of elements that come from a total order which can be partitioned into a set of maximal increasing or decreasing *runs* (i.e., subsequences of consecutive elements), $\mathcal{R} = \{R_1, R_2, \ldots, R_{\rho(A)}\}$. Then the run-based

entropy, $\mathcal{H}(A)$, for A is defined as follows:

$$\mathcal{H}(A) = \sum_{i=1}^{\rho(A)} (r_i/n) \log(n/r_i).$$

Several sorting algorithms have been shown to run in $O(n(1 + \mathcal{H}(A)))$ time; see, e.g., [10, 15, 34, 69, 93, 113, 128]. They are known as *stack-based natural mergesort* algorithms. Recall that a stack-based natural mergesort algorithm maintains a stack of runs and decides when to merge them based on their lengths and/or position in order to take advantage of the input data's pre-sortedness. The existing stack-based natural mergesort algorithms generally consider the top k runs on the stack and merge adjacent runs (using the standard sorted-list merge algorithm) according to some rule, where $k \geq 2$ is a constant that depends on the algorithm. Munro and Wild [113] introduce PeekSort and PowerSort, which use rules based on powers of 2, and achieve a running time of $O(n(1 + \mathcal{H}(A)))$. Auger, Jugé, Nicaud, and Pivoteau [10, 11] study the popular TimSort algorithm [118], whose stack-based combination rule is loosely based on the Fibonacci sequence, and show that it is also instance-optimal.[1] Other stack-based natural mergesort algorithms with this running time include the Adaptive ShiversSort of Jugé [93]; the Multiway PowerSort of Gelling, Nebel, Smith, and Wild [69]; and the α-MergeSort [34] of Buss and Knop. Takaoka's MinimalSort [135], Schou and Wang's PersiSort [128], and adaptive search-tree sorting by Barbay and Navarro [15] take different approaches yet still achieve instance-optimality. Furthermore, TimSort and PowerSort have become the reference sorting algorithms for popular software libraries, including for Python, Java, and Swift [69].

There are two main challenges for implementing stack-based mergesort algorithms to be strictly in-place. The first, which has been extensively studied, is how to merge two sorted subarrays in-place. Fortunately, there are existing methods that perform linear-time merg-

[1] TimSort was recently replaced by PowerSort in the CPython reference implementation of Python [69].

ing strictly in-place; see, e.g., [40, 41, 45, 57, 66, 87, 88, 95, 138]. Even with an in-place merge method, however, we must overcome the second challenge to implementing stack-based mergesorts strictly in-place—*the stack itself*—which can be as deep as $\Omega(\log n)$. Thus, none of these algorithms are suitable for embedded systems where we wish to maintain only an $O(1)$-sized amount of information in addition to the input, which is the subject of this chapter. Alternatively, none of the known strictly in-place sorting algorithms, such as heapsort, are entropy-sensitive.

4.1.1 Our Results

We define a simple *walk-back algorithm*, which can implement any stack-based mergesort algorithm to be strictly in-place. In this algorithm, we only maintain the lengths of a constant number of runs on the stack plus at most $O(1)$ additional metadata. If we ever need to determine the length of a run that occurs deeper in the stack, we simply walk backwards down the array to recover it. Also, if the algorithm used to merge runs is stable [40, 45, 57, 88] (i.e., constructed so that equal elements retain their relative order from the input array, A), then the resulting in-place natural mergesort is also stable.

We call a stack-based natural mergesort algorithm, $\mathcal{S}$, *walkable* if applying the walk-back algorithm to $\mathcal{S}$ increases the running time of $\mathcal{S}$ only by a constant factor. Admittedly, this definition hides some important details about our analysis of the walk-back algorithm. In Section 4.3, we give our main result, showing that PowerSort [113] and c-Adaptive ShiversSort are walkable. We also show that several other stack-based mergesorts are walkable, ShiversSort, α-StackSort, and 2-MergeSort [34, 93]. In Section 4.5, we give a negative result and show that the well-known adaptive sorting algorithms TimSort [10, 11, 118] and α-MergeSort [34] are *not* walkable.

Our negative results motivate our second simple algorithm, which we call the *jump-back algorithm*, that can also be used to implement all known stack-based mergesorts in-place.

In this algorithm, we only maintain the lengths of a constant number of runs on the stack plus at most $O(1)$ words of metadata, and we encode the lengths of runs in-place in the runs themselves using simple bit-encoding methods. If we ever need to determine the length of a run that occurs deeper in the stack, we decode its length from its bit encoding in $O(\log n)$ time, which then allows us to jump to the beginning of the run (and possibly repeat this lookup for the next earlier run). In Section 4.6, we show that we can implement any stack-based natural mergesort algorithm to be in-place using the jump-back algorithm while only increasing its running time by a small constant factor, which is a property we call *jumpable*. Combining our main contributions with known results for the running times of stack-based mergesort algorithms and in-place merge algorithms implies the first strictly in-place sorting algorithms whose running times are instance-optimal with respect to run-based entropy.

We support our theoretical results with experiments found in Section 4.8.

4.2 Preliminaries

Let us begin with some preliminaries for stack-based mergesort algorithms. Let A be an input array of n elements. We define a run as either a non-decreasing or strictly decreasing subsequence of elements in A. All decreasing runs can easily be flipped in-place to be increasing runs in an initial linear-time sweep of A without affecting stability, so, without loss of generality, we only consider non-decreasing runs.

We can separate all stack-based mergesort algorithms into two phases. First is the *main loop*, in which we sweep A from left-to-right, pushing new runs into our stack. Each algorithm defines and is distinguished by a merge policy, $\mathcal{P}$. Whenever we push a new run into our stack and $\mathcal{P}$'s invariant is violated, we perform a sequence of merges of adjacent runs in our stack determined by $\mathcal{P}$ until the invariant is restored. We call this a *merge sequence*. After processing all runs in A, the main loop exits and we perform a *collapse* phase on the

remaining runs in the stack, where we repeatedly merge the top two runs in the stack until only one run remains, our final sorted list. We use the convention that runs closer to the top of the stack are labeled with lower indices, such that the run at the top of the stack takes label R_1. We use r_i to denote the length of run R_i, i.e., $r_i = |R_i|$. We present pseudocode of a generic stack-based mergesort in Algorithm 2. See Figure 4.1.

Algorithm 2 A Generic Stack-Based Mergesort Algorithm with Merge Policy, $\mathcal{P}$

```
Input: array A of size n. Merge policy P.
Output: sorted array A'.
R ← run decomposition of A
S ← ∅                                              ▷ the stack of runs
while R is not empty do                            ▷ main loop
    Push next run from R onto S
    while S violates P's invariant do              ▷ algorithm-specific merge policy
        Merge a pair of adjacent runs in S according to P
while |S| > 1 do                                   ▷ collapse phase
    Pop R_1 and R_2 from S and push MERGE(R_1, R_2) onto S
return pop(S)
```

For ease of expression, we may still use the term "stack" when referring to a given stack-based mergesort, even though our algorithms do not explicitly maintain all of S. We review the merge policies of specific stack-based mergesorts as needed throughout the chapter.

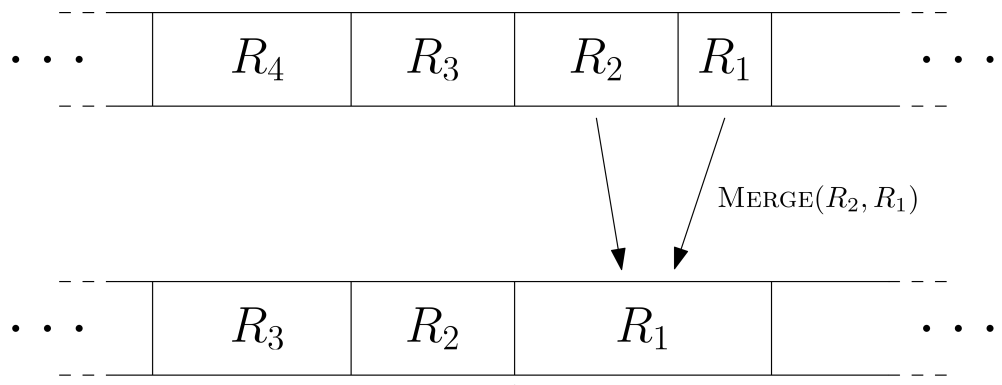

Figure 4.1: A merge step according to a merge policy, $\mathcal{P}$. The merged run takes the label of the higher run. Merging decreases the labels of all runs below the two merged runs.

4.2.1 Shallow Mergesorts

Interestingly, most instance-optimal mergesorts have merge policies that only consider a constant number, k, of runs from the top of the stack. Buss and Knop [34] made a similar observation, which led them to formulate the notion of *k-aware* mergesorts. A stack-based mergesort is *k-aware* if its merge conditions are based on the run lengths of the top k runs in the stack and it only merges runs in the top k entries of the stack. For our purposes, we can also allow merge conditions to operate on other metadata or properties of runs that can be recovered in constant time, such as their position in the array, as well as other global data, provided all of this fits in $O(1)$ memory cells. We call this relaxed definition *almost-k-aware*. Any k-aware algorithm is trivially almost-k-aware.

These definitions suggests a very simple way to make almost-k-aware algorithms in-place: replace our stack S with a "shallow stack" that only maintains the top k runs. Of course, as runs are merged, other runs whose lengths we have forgotten enter the top k entries of S. We must determine the length of these forgotten runs to test whether the merge policy's invariant is still violated. In this chapter, we propose two simple algorithms that efficiently do so, the walk-back algorithm and the jump-back algorithm. They are described respectively in Section 4.3 and Section 4.6. If we can efficiently implement some stack-based mergesort, $\mathcal{S}$, in-place using a shallow stack, then we call $\mathcal{S}$ a *shallow* mergesort. We summarize our results and related properties of known algorithms in Table 4.1.

Algorithm	**Shannon-Optimal**	**Walkable**	**Jumpable**	**Shallow**
PowerSort [113]	✓	✓	✓	✓
c-Adaptive ShiversSort [93]	✓	✓	✓	✓
TimSort [10, 118]	✓	✗	✓	✓
α-StackSort [34]	✗	✓	✓	✓
2-MergeSort [34]	✗	✓	✓	✓
α-MergeSort ($\varphi < \alpha < 2$) [34, 70]	✓	✗	✓	✓
ShiversSort [34]	✗	✓	✓	✓

Table 4.1: A summary of our results applied to selected stack-based mergesorts.

4.3 The Walk-Back Algorithm

In this section, we define the walk-back algorithm and show that a variety of stack-based mergesorts are *walkable*, i.e., applying the walk-back algorithm to them increases runtime by no more than a constant factor. The walk-back algorithm implements a shallow stack like so: whenever we must determine the length of a run for a comparison, we start at the beginning of the run above it in the stack and walk backwards. See Figure 4.2.

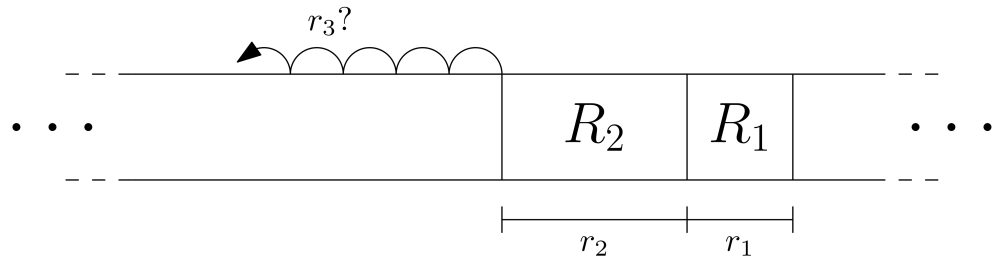

Figure 4.2: The walk-back algorithm applied to an almost-3-aware mergesort. We currently know the lengths of the runs labeled R_1 and R_2. There may be $O(\log n)$ runs to the left of R_2 in the array (our "stack" of runs), but we do not maintain any information about them. Whenever we need to know the length of R_3, we start walking backwards from the start of R_2. Surprisingly, this strategy only increases runtime by a constant factor in several stack-based mergesorts.

4.3.1 The Walk-Back Procedure and Stopping Condition

We keep walking until either (1) we determine the length of the run in question or (2) we walk far enough to confirm whether the given comparison is false. We call (2) our *stopping condition*. It varies by algorithm. If we meet our stopping condition, we forget any information about how far we walked and retry from the beginning the next time we invoke the walk-back algorithm.

In the remainder of this section, we prove that two instance-optimal algorithms, PowerSort and c-Adaptive ShiversSort, are walkable. In combination with an in-place merge algorithm, this produces the first in-place instance-optimal mergesort algorithm.

Let us denote runs from the initial run decomposition $\mathcal{R}$ as *original runs*, and those obtained as the result of merges as *intermediate runs*. Let m be the sum of the sizes of all intermediate runs; the sum of the sizes of all original runs is n by definition. For any stack-based mergesort algorithm $\mathcal{S}$, its runtime $T_{\mathcal{S}}(n) \geq m + n$. This is because, for all intermediate runs i, we must have spent $|i|$ time to merge the two runs that became i. In addition, any sorting algorithm must spend n time just to read the input. If we can show that the walk-back algorithm applied to $\mathcal{S}$ adds no more than $O(m + n)$ time over any input sequence, then we have shown that $\mathcal{S}$ is walkable.

Observation 4.1. *The length of the topmost run, r_1, is always known without need of walking back during the execution of any stack-based mergesort, $\mathcal{S}$.*

Lemma 4.2. *The collapse phase of any stack-based mergesort $\mathcal{S}$ can be implemented in-place using the walk-back algorithm with only a constant-factor increase in runtime.*

The proof follows from the fact that the cost of walking back to recover the last two runs is proportional to the cost to merge them. Because of the above lemma, we only consider the main loop of each stack-based mergesort for the remainder of this work.

4.3.2 PowerSort is Walkable

In this section, we show that PowerSort, an instance-optimal stack-based mergesort, is walkable by seeing how the walk-back cost is 'paid for' by the cost already spent on merges. In particular, we will analyze the walk-back cost of successful merges, determine a stopping condition, and use this stopping condition to compute the walk-back cost of failed merges.

Munro and Wild's PowerSort [113] assigns every run a *power* value, computed jointly with the run who sits above it on the stack. These power values are used to determine whether the second and third runs on the stack should be merged. To understand how runs are assigned powers, imagine superimposing a perfect binary tree on top of the input array,

where each index in the array is associated with a binary tree node.[2] Each index in the array is associated with a power value, where the power of an index is the depth of the corresponding tree node. The center index has a power 0, the first and third quartiles have a power 1, and so on down to $\lfloor \log n \rfloor$. Each run is associated with the smallest power value of any index between its midpoint and the midpoint of the next run, which we refer to as the runs *power interval* I. See Figure 4.3 for an example of how power values appear and how to compute them. Note that [113] define power slightly differently, and in general those power values are one greater than ours. The resulting algorithm is the same, however. We provide a simplified sketch of PowerSort's merge policy in Algorithm 3.

One barrier to applying the walk-back algorithm is that Munro and Wild's original algorithm does not recompute power values when merges occur. Since as many as $\Omega(\log n)$ power values may be stored at a time in the original PowerSort algorithm, maintaining a shallow stack necessitates recomputing power values as runs rise to the top of the stack. It could be that such recomputed power values do not equal their original values, causing the behavior of a shallow PowerSort to differ from standard PowerSort. Fortunately, we show that this is not the case and power values computed on-the-fly are faithful to their original values.

Lemma 4.3. *The merge step of PowerSort (line 4 of Algorithm 3) does not change the power of the second run, nor the power of any existing run.*

[2]We assume for simplicity that the input is of size $n = 2^a - 1$ for some a, although the result holds for general n.

Algorithm 3 PowerSort's Merge Policy (see Line 7 of Algorithm 2)

```
1: if |S| > 1 then
2:     p_2 ← NodePower(R_2, R_1)                ▷ p_3 already known
3:     while |S| > 2 and p_3 > p_2 do
4:         R_2 ← Merge(R_3, R_2)
5:         p_3 ← NodePower(R_3, R_2)
6:     p_3 ← p_2
```

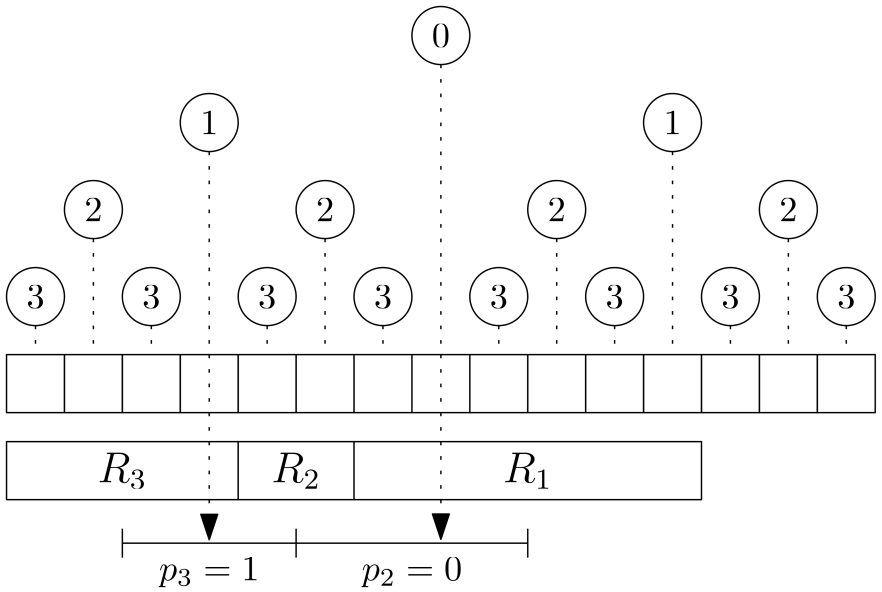

Figure 4.3: Each index in the array is associated with a *power* value, where the power of an index is the depth of the corresponding tree node (depicted above it). Each run is associated with the smallest power value of any index between its midpoint and the midpoint of the next run, i.e. its *power interval* I. The power intervals I_3 and I_2 define powers p_3 and p_2. They are depicted under their respective runs. R_1, being the top run, does not yet have an assigned power.

Proof. See Figure 4.4. A successful merge combines runs R_2 and R_3 where $p_3 > p_2$. Let I_2' denote R_2's power interval after the merge. Since merging only extends I_2 leftward into what was I_3 and all indices in I_3 have power $\geq p_3 > p_2$, the power of R_2 remains p_2. Similarly, R_4's interval extends rightward into I_3, gaining only indices with power $\geq p_3 \geq p_4$, so its power is unchanged. Deeper runs are unaffected. □

From Lemma 4.3, it follows that PowerSort is almost-3-aware since we can correctly recompute all power values with only knowledge of their *current* power intervals. Given the lengths of the top-3 runs and the position of the topmost run, we can compute power intervals in constant time. By Observation 4.1, we always know r_1. It costs at most $r_2 + r_3$ time to recover the lengths of the runs labeled R_2 and R_3 by walking back. If the merge condition is successful, then the merge cost of R_2 and R_3 equals our walk-back cost, so the total walk-back cost in this case is at most m. Thus, we have shown that successful merges "pay" for

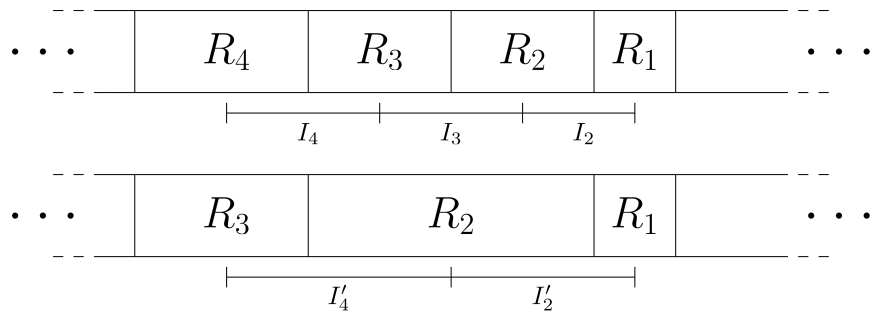

Figure 4.4: Merging runs R_2 and R_3 causes I_2 and I_4 to extend into what used to be I_3. Nevertheless, when R_2 and R_3 are merged, $p_3 > p_2$ and $p_3 \geq p_4$. Therefore, the powers of I'_4 and I'_2 do not change.

their own walk-back cost. It remains to bound the walk-back cost of failed merges. To do so, we next develop a stopping condition for walk-back PowerSort. We begin with the following observations about powers.

Observation 4.4. *Each power p is evenly spaced in the sequence. Two indices with the same power $p > 0$ are separated by at least $n/2^p$ elements.*[3]

Observation 4.5. *The distance between any index with power p and the closest index with power less than p is $n/2^{p+1}$ on either side. Consequently, any range of $n/2^{p+1}$ indices contains either a power p or a power less than p.*

Both follow directly from our description of powers implicitly held at indices.

Observation 4.6. *Any run with size at least $n/2^p$ has a power of at most p.*

This observation follows from the fact that the right half of this run, which is at least size $n/2^{p-1}$, is part of its power interval, and from the second part of Observation 4.5. Rearranging, we obtain the following.

Observation 4.7. *A run of size r must have power at most $p \leq \log \frac{n}{r}$.*

[3]The distance here and in other observations may be off by at most one depending on the exact input size n.

It would be convenient if we could show that long runs always have small powers and short runs always have large powers. However, short runs can have arbitrarily small powers if they appear at indices of small power. Such "lucky" short runs may incur a disproportionate walk-back cost for failed merge comparisons since their eventual merge cost could fail to be proportional to this walk-back cost. Despite this, we show that the overall walk-back cost over the entire main loop is small.

Observation 4.8. *Consider a merge of runs R_3 and R_2 in the merge step of the algorithm. The distance between the midpoints of R_3 and R_1 is at least $n/2^{p_3+1}$, where p_3 was the power of R_3 before the merge.*

Proof. The range of indices between the midpoints of R_3 and R_1 comprises the power intervals I_3 and I_2. Since the merge occurred, $p_3 > p_2$. From the first part of Observation 4.5, the distance between the indices of these two powers is at least $n/2^{p_3+1}$. The proof results from the fact that the nodes containing p_2 and p_3 are in I_2 and I_3 respectively. □

Lemma 4.9. *If $r_3 \geq n/2^{p_2}$, then $p_3 \leq p_2$.*

Proof. This follows directly from Observation 4.6. □

Lemma 4.10. *During PowerSort's main loop, so long as the stack has at least two entries, we always know r_2.*

Proof. Any new R_2 was either previously labeled R_1, whose length is known (Observation 4.1), or is the result of a merge. □

From Observation 4.1 and Lemma 4.10, we always know r_1 and r_2 when testing PowerSort's merge condition. Thus, Lemma 4.9 defines our stopping condition and shows that failed merges incur a walk-back cost of at most $n/2^{p_2}$.

Lemma 4.11. *The total walk-back cost of failed merges in PowerSort is $O(m+n)$.*

Proof. Consider two runs, q and s, such that s appears directly above q in the stack. Their respective powers are p_q and p_s. Say q currently has label R_3, s has label R_2, and we are about to test the merge condition between them. Say we find that $|q| > n/2^{p_s}$; by Lemma 4.9, the merge condition fails, we exit the walk-back algorithm and terminate the merge sequence. What can we charge this walk-back cost of $n/2^{p_s}$ to?

We consider two cases. In case 1, q is never part of a failed merge again. Then we can charge this cost to q itself since $|q| > n/2^{p_s}$. Over all runs, q, this adds at most $(m+n)$ to the walk-back cost as q may be original or intermediate.

In case 2, q will be part of another failed merge. This means that q will have to take the label R_3 again, which implies that s gets merged during the main loop. When s is merged, q has label R_4 and s has label R_3. By Observation 4.8, the distance between the midpoints of R_3 and R_1 is at least $n/2^{p_s+1}$. This implies that $r_1/2 + r_2 + r_3/2 \geq n/2^{p_s+1}$, so $r_1 + 2r_2 + r_3 \geq n/2^{p_s}$. For simplicity, we upper bound this as $2(r_1 + r_2 + r_3)$.

We can charge our cost of $n/2^{p_s}$ to twice the sizes of the current runs with labels R_1, R_2, and R_3. All three runs are charged in this way once. For R_2 and R_3, this is because they are immediately merged and so destroyed. For R_1, recall that we assume that q fails its merge condition, so the merge sequence is subsequently terminated. When a merge sequence finishes, we push a new run onto the stack, shifting the run labeled R_1 to R_2. There is no way for this run to ascend back to R_1, so it is charged as R_1 once.

As a result, this case charges each run seen during the main loop twice its length at most three times (i.e., when labeled R_1, R_2, and R_3). This contributes an additional $O(m+n)$ to the total walk-back cost of failed merges. □

We have shown that, over all merges, the walk-back cost is bounded by $O(m+n)$. We conclude with the following.

Theorem 4.12. *PowerSort is walkable.*

Corollary 4.13. *We can implement PowerSort with the walk-back algorithm to produce an in-place, stable sorting algorithm with running time* $\Theta(n(1+\mathcal{H}(A)))$.

Proof. This follows from properties of PowerSort, stable in-place merge algorithms, and the walk-back algorithm. □

4.3.3 c-Adaptive ShiversSort is Walkable

In this section, we show that the recent instance-optimal stack-based mergesort, c-Adaptive ShiversSort [93], is walkable. See Algorithm 4 for its merge policy. The author defines c as a tunable parameter which allows for a tighter runtime analysis. Such details are not relevant to our work, however.

Algorithm 4 c-Adaptive ShiversSort's Merge Policy (see Line 7 of Algorithm 2)

```
1: ℓ1 ← ⌊log(r1/c)⌋ (resp. ℓ2, ℓ3)
2: while |S| ≥ 3 and ℓ3 ≤ max{ℓ2, ℓ1} do
3:     R2 ← MERGE(R3, R2)                ▷ removes R3 from S
```

We first show that all successful merges during the main loop "pay for" the walk-back cost required to perform them. Assume that the condition on line 2 of Algorithm 4 is true. From Observation 4.1, we already know r_1. The walkback cost to perform this operation is at most $r_2 + r_3$. This is exactly the cost to merge the runs labeled R_2 and R_3. As a result, the walk-back cost over all successful merges is bounded by the total merge cost throughout the main loop, which is at most m. It remains to bound the walk-back cost when the condition fails to hold. We first determine the stopping condition for c-Adaptive ShiversSort.

Lemma 4.14. *If* $r_3 > 2 \times \max\{r_1, r_2\}$, *then* $\ell_3 > \max\{\ell_1, \ell_2\}$.

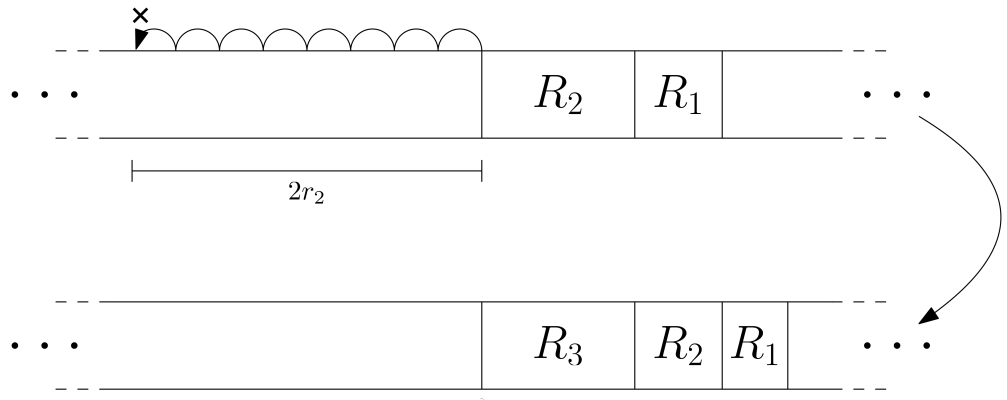

Figure 4.5: In this figure, we illustrate what happens when the merge condition fails in c-Adaptive ShiversSort. Assume w.l.o.g. that $\ell_2 = \max\{\ell_2, \ell_1\}$. We have walked back $2 \times r_2$ steps yet have not found r_3. By Lemma 4.14, the merge condition must be false. Thus, we continue the main loop by pushing a new run onto the stack.

Proof. Without loss of generality, let $r_2 = \max\{r_1, r_2\}$. Then $\ell_2 \geq \ell_1$. Since $r_3 > 2r_2$, the following holds true.

$$\ell_3 \geq \lfloor \log(2r_2/c) \rfloor = \lfloor \log(r_2/c) + 1 \rfloor = \lfloor \log(r_2/c) \rfloor + 1 > \ell_2$$

□

The above lemma gives us the stopping condition for the walk-back algorithm applied to c-Adaptive ShiversSort: if we walk back $2 \times \max\{r_1, r_2\}$ steps and have not found r_3, then we know that the merge condition in line 2 of Algorithm 4 is false. See Figure 4.5.

Lemma 4.15. *The walk-back cost incurred by failed merges is $O(m + n)$.*

Proof. As described above, we spend $2 \times \max\{r_1, r_2\}$ walking back when we fail a merge condition. Since no subsequent merge occurs, we do not immediately "pay back" the cost. For simplicity, we can upper bound this cost to at most $2 \times (r_1 + r_2)$. Thus, every failed merge check incurs a walk-back cost of at most twice the length of the runs labeled R_1 and R_2.

To complete the argument, we show that each run is labeled R_1 and R_2 for at most one merge sequence each throughout the main loop. Consider any run q with label R_1 or R_2 when a failed condition occurs. Assuming the run decomposition is not empty, we push a new run onto the stack and then repeat the main loop. This relabels q to R_2 or R_3, respectively. Since R_1 and R_2 are never merged in this algorithm, it is impossible to ascend from label R_2 to R_1. It is possible to change labels from R_3 to R_2 with a merge. Nevertheless, doing so destroys the constituent runs in the merge and replaces them with a new merged run. As a result, each run is labeled R_1 and R_2 for at most one merge sequence.

The sum of all original and intermediate runs is $m + n$. Each run q contributes a walk-back cost of $2|q|$ at most twice (once as R_1 and again as R_2). Then the total walk-back cost of failed merges is at most $4(m + n)$. □

We have shown that the walk-back cost of successful merges and failed merges during the main loop of c-Adaptive ShiversSort is $O(m + n)$. In combination with Lemma 4.2, we conclude the following.

Theorem 4.16. *c-Adaptive ShiversSort is walkable.*

Upon further inspection, our proof relies on two key properties of c-Adaptive ShiversSort. First, we only ever merge R_3 and R_2 and never examine runs below R_3. Second, the walk-back cost of a failed merge condition is bounded by a constant multiple of our known runs. The first property allows us to easily show that the walk-back cost of successful merges is paid for by the subsequent merge. The second is a relaxed stopping condition that allows us to bound the walk-back cost of failed merges. Indeed, with slight modification, the above proof applies to *any* almost-k-aware mergesort with the following two properties:

1. Only R_k and R_{k-1} are ever merged during the main loop.

2. If $r_k > \alpha \sum_{i=1}^{k-1} r_i$ for some constant α, no merge occurs.

Corollary 4.17. *The original ShiversSort and α-StackSort are walkable.*

Proof. Since both ShiversSort and α-StackSort [34] are almost-2-aware, they trivially satisfy the first property. ShiversSort's merge condition, $2^{\lfloor \log r_2 \rfloor} \leq r_1$, satisfies the second property with $\alpha = 2$. α-StackSort similarly does so with its own parameter α. Thus, both stack-based mergesorts are walkable. □

4.4 Additional Walkable Mergesort Proofs

4.4.1 The Original TimSort

Prior to a 2015 work by de Gouw, Rot, de Boer, Bubel, and Hähnle [47], TimSort's merge policy could be described with three merge conditions, as shown in Algorithm 5. The authors showed that this version may fail to maintain the invariant that run sizes growth at least as fast as the Fibonacci sequence. They proposed a corrected version of TimSort with an extra fourth condition added (see Algorithm 7). We show in this section that the original version of TimSort is walkable, whereas we show in Section 4.5 that the modern TimSort defined by de Gouw *et al.* is not walkable.

Algorithm 5 Original TimSort's Merge Policy (see Line 7 of Algorithm 2)

```
1: while true do
2:     if |S| > 3 and r_1 > r_3 then                    ▷ merge condition #1
3:         R_2 ← MERGE(R_3, R_2)
4:     else if |S| > 2 and r_1 ≥ r_2 then               ▷ merge condition #2
5:         R_1 ← MERGE(R_2, R_1)
6:     else if |S| > 3 and r_1 + r_2 ≥ r_3 then         ▷ merge condition #3
7:         R_1 ← MERGE(R_2, R_1)
8:     else
9:         break
```

As before, we begin by analyzing the walk-back cost of successful merges.

Lemma 4.18. *When we perform a merge, the walk-back cost is paid for by at most 2 times the subsequent merge cost.*

Proof. We prove this by discussing the walk-back and merge costs of each of the three conditions. By Observation 4.1, we always know r_1 throughout the algorithm. For simplicity, assume that we begin each iteration of a merge sequence only knowing r_1. To test merge condition 1, we must spend r_2 time just walking from the end of R_1 to the end of R_2 as we assume we do not know r_2. If the condition is true, we spend an additional r_3 time walking back. As such, the resulting merge of R_2 and R_3 perfectly pays for the walk-back cost.

Now we discuss condition 2. We know that we just tested and failed condition 1. Again we spent r_2 time walking to find the end of R_3. However, because condition 1 failed, we only spent r_1 time walking before stopping. In doing so, we have recovered r_2. Thus, condition 2 can be determined without any additional walk-back cost. We spent $r_1 + r_2$ time walking back and end up merging R_1 and R_2, so the merge again perfectly pays for the walk-back cost.

Lastly, we discuss condition 3. To test and fail conditions 1 and 2 incurs $r_1 + r_2$ walk-back cost. An additional r_3 work is needed to confirm that $r_1+r_2 \geq r_3$. Thus, we spend $r_1+r_2+r_3$ time in total walking back. Since the condition succeeded, we know $r_1 + r_2 \geq r_3$, so the walk-back cost is upper-bounded by twice the merge cost of merging R_1 and R_2. □

Now we study the walk-back cost of failed merges.

Observation 4.19. *Failing all three merge conditions in Original TimSort incurs a walk-back cost of at most* $2(r_1 + r_2)$.

Proof. We once again assume that we are only aware of r_1. As discussed before, failing on condition 1 requires $r_1 + r_2$ time spent walking back and failing on condition 2 incurs no

walk-back cost. Finally, failing on condition 3 requires an additional $r_1 + r_2$ time to walk back on R_3. □

Observation 4.20. *Each unique run can contribute to failing all merge conditions at most once as R_1 and at most once as R_2*

Proof. If the run was R_1, the new run added in the next iteration pushes it to R_2. The same run cannot move back to R_1 without merging first. Likewise, if the run was R_2, the new run pushes it to R_3. It cannot move back to either R_2 or R_1 without merging. □

As a result, each run contributes at most 4 times their size (2 times their size as R_1 and 2 times their size as R_2). This can happen for every original and intermediate run, thus the total walk-back cost of failed merges is $O(m + n)$. Therefore, we conclude the following.

Theorem 4.21. *Original TimSort is walkable.*

4.4.2 The 2-MergeSort

In 2019 Buss and Knop introduced several new natural stack-based mergesorts, which they call the 2-MergeSort and the α-MergeSort [34]. In Section 4.5.1, we show that α-MergeSort (not to be confused with the α-StackSort introduced by Auger *et al.* [11]) is not walkable. Here, we show that the 2-MergeSort is. The merge policy of the 2-MergeSort is described in Algorithm 6.

Algorithm 6 2-MergeSort's Merge Policy (see Line 7 of Algorithm 2)

```
1: while |S| ≥ 3 and r_2 < 2r_1 do
2:     if r_3 < r_1 then
3:         R_2 ← MERGE(R_3, R_2)          ▷ removes R_3 from S
4:     else
5:         R_1 ← MERGE(R_2, R_1)          ▷ removes R_2 from S
```

Theorem 4.22. *2-MergeSort is walkable.*

Proof. As always, we begin by discussing the walk-back costs when a merge occurs. By Observation 4.1, we always know r_1. If a merge occurs, then the condition $r_2 < 2r_1$ is true and we incur walk-back cost r_2 to test this. If $r_3 < r_1$, we incur additional walk-back cost r_3. Since we merge R_3 and R_2, the merge perfectly pays for the walk-back cost of $r_2 + r_3$. The case is symmetric for if $r_3 \not< r_1$.

Now we discuss the cost associated when no merge occurs. This occurs when we fail the initial comparison, so $r_2 \geq 2r_1$. Therefore, we incur walk-back cost $2r_1$. We observe that this can only happen once per unique run during the main loop. Whenever a failure occurs, we push a new run, displacing R_1 to R_2. To return to spot one of the stack, this run will have to merge, destroying it. Thus, the total walk-back cost is $O(m + n)$. □

4.5 TimSort is Not Walkable

It is tempting to try to extend our walk-back analyses to TimSort. Unfortunately, in this section we show that TimSort is not walkable by providing an input instance such that TimSort implemented with the walk-back algorithm performs asymptotically worse than standard, non-in-place TimSort. This motivates our jump-back algorithm, presented in Section 4.6, which works for virtually all stack-based mergesorts. In Section 4.5.1, we use a similar argument to show that α-MergeSort is also not walkable.

Algorithm 7 TimSort's Merge Policy (see Line 7 of Algorithm 2)

```
1:  while true do
2:      if |S| > 3 and r_1 > r_3 then                    ▷ merge condition #1
3:          R_2 ← Merge(R_3, R_2)
4:      else if |S| > 2 and r_1 ≥ r_2 then               ▷ merge condition #2
5:          R_1 ← Merge(R_2, R_1)
6:      else if |S| > 3 and r_1 + r_2 ≥ r_3 then         ▷ merge condition #3
7:          R_1 ← Merge(R_2, R_1)
8:      else if |S| > 4 and r_2 + r_3 ≥ r_4 then         ▷ merge condition #4
9:          R_1 ← Merge(R_2, R_1)
10:     else
11:         break
```

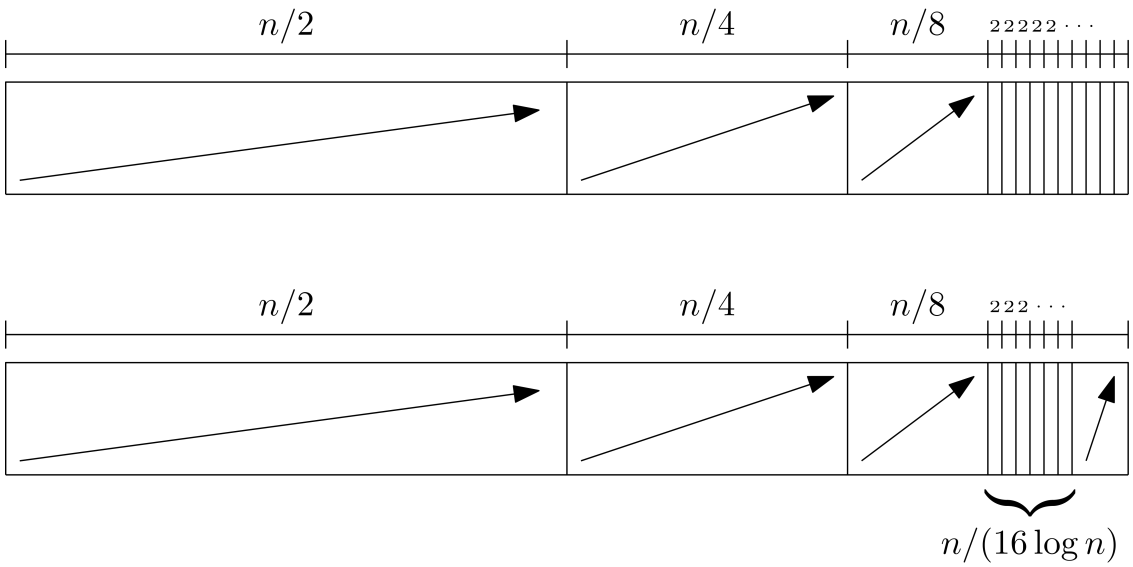

Figure 4.6: The worst-case input sequences we construct for our TimSort walk-back implementation. The top array represents the first version we analyze. The series of $n/16$ runs of size 2 cause $\Theta(\log n)$ *stages* to occur, blowing up walk-back cost to $\Omega(n \log n)$. The bottom array reduces the number of runs of size 2 to $n/(16 \log n)$. The walk-back cost remains $\Omega(n \log n)$ while the Shannon entropy of the array becomes constant. Therefore, TimSort with the walk-back algorithm will perform asymptotically worse than standard TimSort on the second sequence.

Our counterexample considers an array of length n with the following preexisting runs. The first $n/2$ elements form a run, the next $n/4$ elements form a run, and the next $n/8$ elements form a run. The remaining elements form a sequence of $n/16$ runs, each of size 2. For simplicity, assume n evenly divides into these runs. See Figure 4.6. Our proof exploits the following key observations, which follow directly from TimSort's merge policy (see Algorithm 7).

Observation 4.23. *If a merge sequence in TimSort produces a stack of runs whose lengths form a sequence of strictly decreasing powers of 2, then the merge sequence terminates.*

Observation 4.24. *We must know r_2 to validate that all merge conditions are false at the end of a merge sequence.*

Observation 4.23 has two notable effects. First, the beginning three runs, those of lengths $n/2$, $n/4$, and $n/8$, are only ever merged at the very end of the main loop. Second, repeatedly adding runs of size 2 causes them to collapse into powers of 2. For example, adding the first four runs to the stack produces the series of run lengths $\{n/2, n/4, n/8, 2\}$. This produces no merges. However, adding the next run produces a stack like so: $\{n/2, n/4, n/8, 2, 2\}$, which collapses into $\{n/2, n/4, n/8, 4\}$ by merge condition #2 of Algorithm 7. We generalize this with the following definition.

Definition 4.1. *Let* stage i *of TimSort run on our input instance be defined as follows. Consider a set of runs on the stack with sizes*

$$\{n/2, n/4, n/8, 2^{i-1}, 2^{i-2}, \ldots, 8, 4, 2\}.$$

Since their sizes are decreasing powers of 2, no merges occur by Observation 4.23. In the next iteration of the main loop, an additional run is added. We say this is the beginning of stage i. The run lengths are like so:

$$\{n/2, n/4, n/8, 2^{i-1}, 2^{i-2}, \ldots, 8, 4, 2, 2\},$$

which causes merge condition #2 to be repeatedly applied. For $i < \log(n/8)$ and by Observation 4.23, the merge sequence terminates when the stack is of size four, with run lengths like so:

$$\{n/2, n/4, n/8, 2^i\}.$$

Furthermore, let $k = O(1)$ be the size of our shallow stack. As defined by the walk-back algorithm, we only maintain the top k elements of the stack. The rest are forgotten.

Observation 4.25. *At the beginning of stage i, there are $i + 3$ runs in the stack.*

Lemma 4.26. *There are $\Theta(\log n)$ stages that occur throughout TimSort's main loop.*

Proof. The first three runs have total length $7n/8$. Then there are $n/8$ remaining elements across all remaining runs. At the end of stage i, the last run is of size 2^i. This occurs for all i such that $1 \leq i < \log(n/8)$ by definition. Then i is at most $\log(n/8) - 1$. Thus, we have $\Theta(\log n)$ stages throughout the algorithm. □

Corollary 4.27. *There are $\Theta(\log n)$ stages such that $i > k$.*

Proof. This follows from the fact that k is a constant. □

Lemma 4.28. *At all stages i such that $i > k$, applying the walk-back algorithm incurs a walk-back cost of $\Omega(n)$.*

Proof. By Observation 4.25, there are $i + 3$ runs in the stack at the beginning of this stage. By the fact that $i > k$, this means that the bottom three runs of size $n/2$, $n/4$, and $n/8$ have been forgotten. When the merge sequence terminates, there are four runs in the stack. This means that the run of size $n/8$ takes label R_2. By Observation 4.24, we must know r_2 to determine that the merge sequence is indeed finished. The result follows from the fact that $r_2 = n/8 \in \Omega(n)$. □

By Lemma 4.27 and Lemma 4.28, the total walk-back cost of TimSort under this input instance is in $\Omega(n \log n)$. As described, however, our counterexample input instance has entropy $\Omega(\log n)$. Thus, standard TimSort would also run in time $\Omega(n \log n)$.

A small modification to our counterexample gives us the desired result. Instead of $n/16$ runs of size 2, reduce this to $n/(16 \log n)$ runs. Have the remaining elements form a single run of size $n/8 - n/(8 \log n)$. See Figure 4.6. The Shannon entropy of this new array is constant, so standard TimSort would run in linear time. However, a similar analysis

of this variant shows that applying the walk-back algorithm incurs a walk-back cost of $\Omega(n \log(n/\log n)) \subseteq \Omega(n \log n)$. We conclude the following.

Theorem 4.29. *TimSort is not walkable.*

4.5.1 Additional Counterexample

We can use a similar argument to above to show that α-MergeSort of Buss and Knop [34] is not walkable.

First, let us consider the merge policy of α-MergeSort.

Algorithm 8 α-MergeSort's Merge Policy (see Line 7 of Algorithm 2)

```
1: while r_2 < αr_1 or r_3 < αr_2 do
2:     if r_3 < r_1 then
3:         R_2 ← MERGE(R_3, R_2)
4:     else
5:         R_1 ← MERGE(R_2, R_1)
```

Theorem 4.30. *There exist input sequences that can be sorted in $O(n)$ time using standard α-MergeSort that take time $\Omega(n \log n)$ when using α-MergeSort implemented with the walk-back algorithm. Therefore, α-MergeSort is not walkable.*

Proof. Our proof is similar to the proof for TimSort, and in fact, when $\alpha = 2$, the identical counterexample works. Similar sequences work for other values of α as well where the three original runs are of sizes $n/\alpha, n/\alpha^2, n/\alpha^3$. Without loss of generality, we assume $\alpha = 2$ for this proof and use the same counterexample as for TimSort.

We note that, similar to TimSort, only the first while loop condition (on line 1) will ever succeed, while the check in the if statement (on line 2) will always fail, and consequently only the merge on line 5, between runs R_3 and R_2, will ever occur. Therefore the algorithm will perform identical merges to TimSort on this input. And as with TimSort, the problematic walk-back cost occurs after a stage has collapsed, and the stack has just four runs:

$\{n/2, n/4, n/8, 2^i\}$. On the second while loop condition, where we check if $r_3 < 2r_2$, when implementing the walk-back algorithm, we must at least fully load R_3 to confirm that the condition is false. Therefore, the walk-back cost is at least $n/8$ (and will in-fact be $3n/8$), and our TimSort analysis holds. □

4.6 The Jump-Back Algorithm

In this section, we describe our jump-back algorithm, which is a general method for implementing all the known stack-based natural mergesort algorithms to be in-place without a stack, albeit while sacrificing stability. The only assumption required is that a stack-based mergesort algorithm, $\mathcal{S}$, be almost-k-aware. Importantly, if the running time of $\mathcal{S}$ is $O(n(1 + \mathcal{H}(A)))$, then the running time of our in-place jump-back implementation of $\mathcal{S}$ will also run in time $O(n(1 + \mathcal{H}(A)))$.

Suppose we are given an input array, A, of size n, and let $\lambda = \lceil \log n \rceil + 1$ (λ is a parameter we that we use throughout our jump-back algorithm). We begin by moving the elements in *short* runs, those of size at most 3λ, to the end of A. We then sort them via a standard in-place mergesort algorithm; e.g., see [40, 45, 54, 55, 57, 88]. Our method for performing this move in-place can be done in linear time using a partitioning algorithm described in Section 4.6.1. See Figure 4.7.

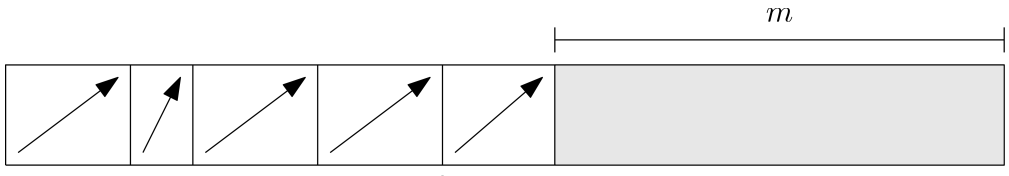

Figure 4.7: The result of performing partitioning to move all short runs to the end of the array. All runs to the left of the gray block are of size $> 3\lambda$. The elements in the gray block belonged to runs of size $\leq 3\lambda$. Their ordering may not be preserved.

We then apply the given stack-based natural mergesort algorithm $\mathcal{S}$ to the remaining, *long*, runs, with one change—rather than using a complete stack to store run sizes, as in the

original way $\mathcal{S}$ was defined, we use a shallow stack of size k and we use bit-encoding to encode the size of each run *in the run itself*.

Lemma 4.31. *The size of any* long *run can be encoded in the run itself using a reversible in-place encoding scheme that can be encoded, decoded, and reversed in* $O(\lambda) = O(\log n)$ *time.*

This lemma is proved by the introduction of two new encoding schemes which we call *pivot-encoding*, which works when there are a sufficient number of distinct elements in the run, and *marker-encoding*, which works in general. Under either scheme, we encode the size of a run within the last $\lambda + 1$ elements of the run. We discuss these encoding schemes in Section 4.7. See Figure 4.8.

Recall that we have already moved all short runs to the end of the array. Using Lemma 4.31, we can implement the almost-k-aware algorithm, $\mathcal{S}$, in-place on the remaining long runs using our jump-back algorithm with only two changes.

1. Whenever a new run, q, is added and the bottommost run, s, in the shallow stack no longer fits, we *encode* s's size using the algorithm of Lemma 4.31. This step takes $O(\lambda)$ time, and can be charged to the run, q, that was just added to the stack, since its size must have been at least $3\lambda + 1$.

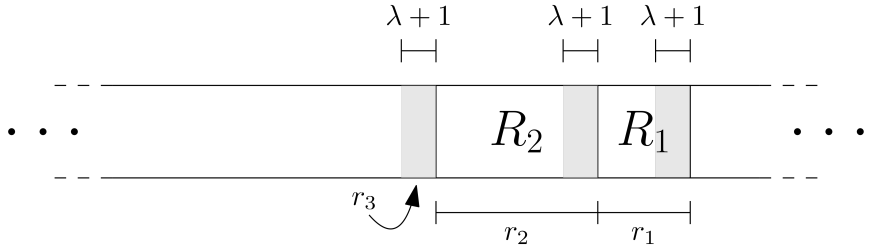

Figure 4.8: We encode run lengths within the last $\lambda + 1$ elements of each run, indicated with shading, allowing us to compute the run's length in $O(\log n)$ time. This eliminates the need for walking back via the walk-back algorithm albeit at the loss of stability.

2. Symmetrically, whenever a run is merged and the shallow stack is no longer full, we add the run deeper in the array to the shallow stack by *decoding* its size and by *reversing* the encoding such that the original run is restored. This step also takes $O(\lambda)$ time, and can be charged to the merge operation.

Using these two changes, we can maintain all the information of our shallow stack efficiently, without asymptotically increasing the running time of our algorithm, $\mathcal{S}$. All that is left is to sort all the remaining *short* runs.

All short runs, of size at most 3λ, were initially moved to the end of the array. Let us denote this portion of the array as A'. By the definition of the Shannon entropy of an array, $n\mathcal{H}(A) = \sum_{i=1}^{\rho(A)} r_i \log(n/r_i)$, and we see that

$$\sum_{i=1}^{\rho(A')} r_i \log(n/r_i) \geq \sum_{i=1}^{\rho(A')} r_i \log(n/3\lambda).$$

But $\lambda = \lceil \log n \rceil + 1$; hence, if we let $m = \sum_{i=1}^{\rho(A')} r_i$, then the sum of their entropy is $\Omega(m \log n)$. As a result, the cost of sorting all runs of size at most 3λ in time $O(m \log m)$ is "paid" for by their contribution to the entropy of the original array. We spend an additional $O(n)$ time moving all such runs to the end of the array. (see Section 4.6.1). As noted above, the time to sort the long runs in A is asymptotically the same as $\mathcal{S}$. Thus, we have the following:

Theorem 4.32. *Given an array, A, of size n, and a stack-based natural mergesort algorithm, $\mathcal{S}$, which is almost-k-aware for constant $k \geq 2$, we can implement A in-place in $O(n(1 + \mathcal{H}(A)) + T_{\mathcal{S}}(n))$ time, where $T_{\mathcal{S}}(n)$ is the running time of $\mathcal{S}$.*

4.6.1 In-Place Run Partitioning

For the sake of completeness, we describe a simple partitioning algorithm that can be used, for example, to move all short runs in an array, A, to the end of A, which also moves the long runs to be before that end region, without destroying the long runs.

We can abstract this problem by representing each element of a long run as 1's, and each element of a short run as 0's. Suppose A is an array of n elements, each of which is either 0 or 1, such that there are $n - m$ 1's and m 0's in A. Our goal of moving the short runs to the end of A is equivalent to segregating the 0's and 1's in A, such that the $n - m$ 1's precede the m 0's. While it is important for us that the 1's are moved stably, i.e., that the long runs are not destroyed, it is not important for us and in general our algorithm will not preserve the stability of the 0's, i.e., the short runs. This can be done in-place in $O(n)$ time by making a single pass through A while maintaining two pointers, one for the next 1 and one for the first preceding 0. See Algorithm 9.

In our setting, we do not know which elements are 0's or 1's *a priori*. However, since these labels are determined by the length of each run and our algorithm performs a single pass through A, we can scan each run as we come across them to determine whether they are a run of 0's or 1's. We may scan each run at most twice, once for each pointer, so this adds an additional $O(n)$ time.

Algorithm 9 In-Place Run Partitioning

```
Let A[i] be the first cell with A[i] = 0
Let A[j] be the first cell with A[j] = 1 and j > i
Swap A[i] and A[j]
while i < n do
    Let A[i] be the next cell with A[i] = 0
    Let A[j] be the next cell with A[j] = 1 and j > i
    Swap A[i] and A[j]
```

Note that this algorithm will move 0's forward in a way that preserves their original order in the array, A. Thus, we have the following:

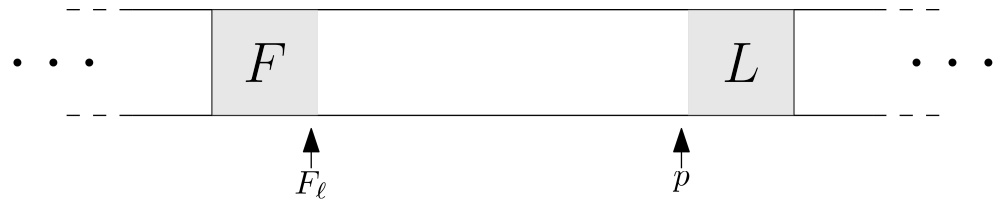

Figure 4.9: Here we depict the test to determine whether a run is *pivotable*. We define F and L as the first and last λ cells of the run. Our pivot p is the element just prior to L and F_ℓ is the last element of F. This run is pivotable if $F_\ell < p$.

Theorem 4.33. *Given an array, A, of n cells, m of which store 1's and $n - m$ of which store 0's, we can stably partition the 0's in-place to the first $n - m$ cells in A in $O(n)$ time.*

4.7 Our Bit Encoding Methods

In this section, we describe our methods for encoding run sizes in-place in the runs themselves. Unlike previous encoding methods, such as those of Kuszmaul and Westover [101], our methods make no assumptions about the elements and their representation themselves, except that they are comparable and copyable. Recall that we define $\lambda = \lceil \log n \rceil + 1$. We do so because the size of each run can be represented using $\lceil \log n \rceil$ bits, along with one extra bit to specify the encoding method used. More specifically, let $b_1 b_2 \ldots b_{\lceil \log n \rceil}$ be a binary encoding of the size, r, of a *long* run, R, and let b_λ indicate the encoding method used, i.e., $b_\lambda = 0$ for the *pivot-encoding* method and $b_\lambda = 1$ for the *marker-encoding* method. Both of our methods efficiently encode $\vec{b}$ in-place within the run, R.

Our Pivot-Encoding Method. Our first encoding method is a simple encoding method we call the *pivot-encoding* method. Let R be a *long* run, and let F and L be the first and last λ elements in R, respectively. Also, let p be the element of R that is immediately before L, which we call the *pivot* (i.e., p is the element $\lambda + 1$ cells from the end of R). We say that the run R is *pivotable* if the last element of F, F_ℓ, is strictly smaller than our pivot, p. See Figure 4.9.

Suppose then that R is pivotable, which implies that each element in F is strictly smaller than every element in L and p. Encoding $\vec{b}$ in our pivot-encoding scheme is easy: We consider each element, $L[i]$, of L as corresponding to the bit, b_i, and, if $b_i = 0$, then we swap $L[i]$ and $F[i]$. Thus, if $L[i] < p$, we know that $b_i = 0$, and if $L[i] \geq p$, then we know that $b_i = 1$. See Figure 4.10. Encoding, reading, and reversing such an encoding can be done in $O(\log n)$ time.

Our Marker-Encoding Method. For a long run, R, not to be pivotable, i.e., $F_\ell \geq p$, it must be true that not only $F_\ell = p$, but also that the entirety of the elements between F and L, denoted M, are equal to p. In this case, our pivot-encoding method does not work, and we use a different encoding method, which we call the *marker-encoding* method.

For our marker-encoding method, we designate the first two distinct elements encountered as *markers*, denoting them m_1 and m_2, where $m_1 < m_2$, and store them using $O(1)$ memory. These markers are in fact derived from the first run itself, since each run must contain at least two distinct elements.

Since R is not pivotable, we know that $F_\ell = p$, and thus all elements in M are equal to p. First, we store a copy of L in the first λ cells of M. Next, we encode $\vec{b}$ in L using only our two markers, letting $L[i]$ correspond to the bit b_i, and, if $b_i = 0$, then we set $L[i] \leftarrow m_1$ and if $b_i = 1$, then we set $L[i] \leftarrow m_2$. Finally, we set the pivot value p to m_2. Reading

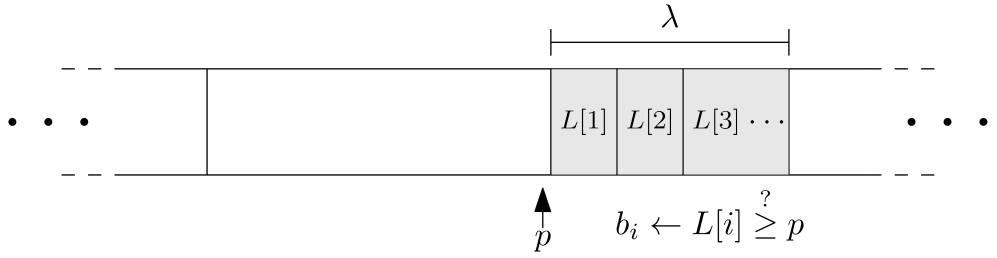

Figure 4.10: The decoding process, which applies to both our pivot-encoding and marker-encoding schemes. The last λ elements encode our bit string $\vec{b}$. Element p, positioned $\lambda + 1$ elements from the end, determines the bit corresponding to each $L[i]$. When decoding, if $L[i] \geq p$, then b_i is 1. Otherwise, it is 0.

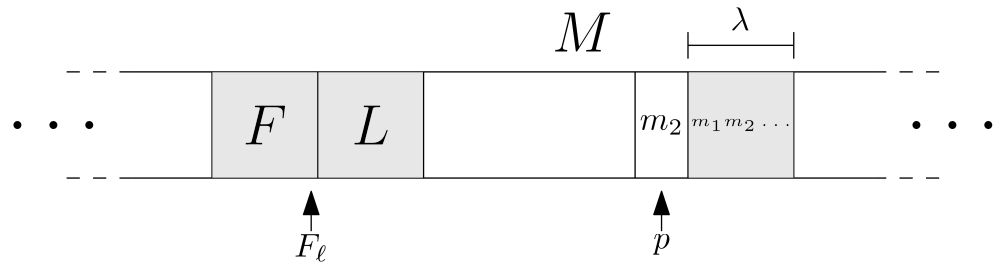

Figure 4.11: Transforming the run to apply marker-encoding. We copy the block L to just after F, with the remaining entries belonging to M. As in pivot-encoding, we use p and the last λ entries of the array to encode $\vec{b}$. However, this time it is done via our markers m_1 and m_2, rather than the elements of F and L. We replace p's element with m_2 to make the decoding process identical to that of pivot-encoding.

this encoding is identical to how the pivot-encoding method is read, since it is still true that if $L[i] < p$, then $b_i = 0$ and if $L[i] \geq p$, then $b_i = 1$. After the encoding is read and we must reverse the encoding, we simply set L back to its original values using its copy in M, and then we clean up M by copying F_ℓ into its first λ elements and into the original pivot location, p. See Figure 4.11.

4.8 Experimentally Evaluating Walk-Back Cost

We support our theoretical results with experiments by comparing the performance of several stack-based mergesorts with their in-place versions produced from the walk-back algorithm. We also compare our algorithms against the performance of the built-in C++ implementation of introsort, `std::sort`, and against QuickXsort, a recent in-place sorting algorithm.

4.8.1 Experimental setup

We ran our C++ experiments on `Ubuntu 22.04.5 LTS` (Linux kernel `5.15.0-135-generic`). The hardware comprised two Intel® Xeon® X5680 processors (@ 3.33 GHz, 24 MiB total L3 cache) and 94 GiB of RAM. The code was compiled using `GCC` version `11.4.0` with the `-O3 -march=native` optimization flags. We ran our benchmarks using `Google Benchmark` version `1.8.3`, which is a microbenchmarking framework for C++.

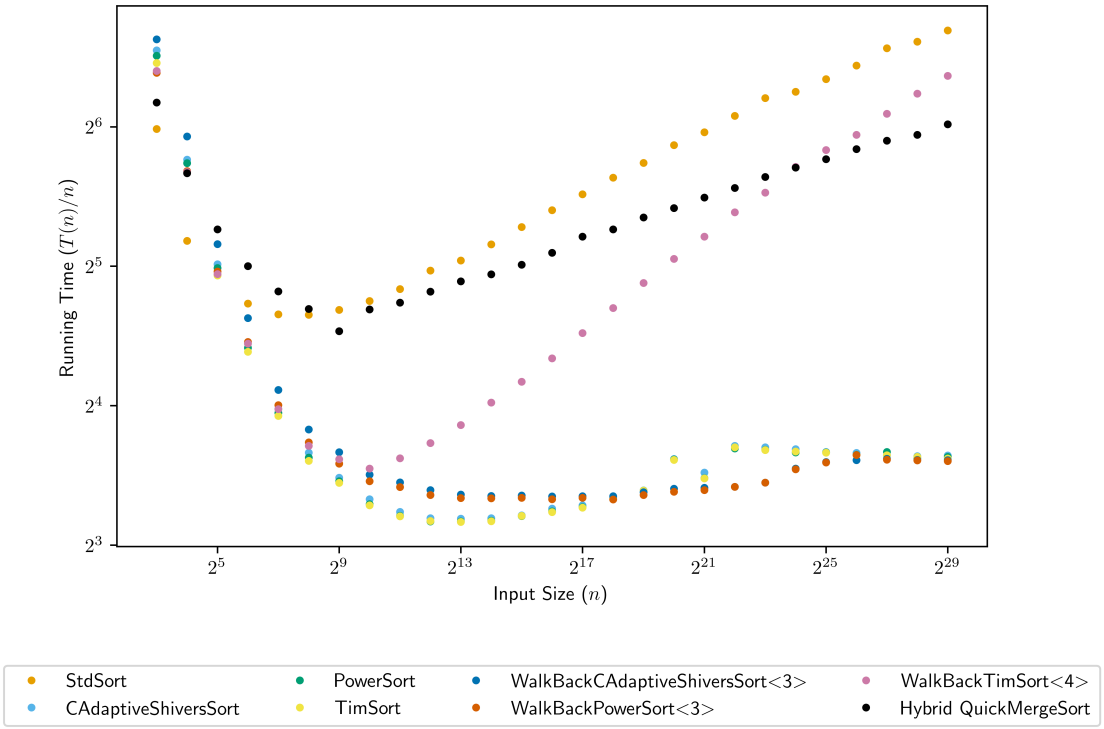

Figure 4.12: A comparison of normalized running times on our TimSort counterexample inputs as described in Section 4.5. Recall that for this input the Shannon entropy is constant. Thus, we scale the running times by n. As opposed to our stack-based mergesorts, `std::sort` and Hybrid QMS are not instance-optimal, so they cannot take advantage of the natural ordering of the input. Further, while all our in-place versions of the *walkable* mergesorts are able to take advantage of the natural ordering of the input, walk-back TimSort is not. This confirms our proof in Section 4.5 that TimSort is not walkable and it requires the encoding techniques described in Section 4.6.

We tested two types of inputs: (1) our TimSort counterexample input as described in Section 4.5 and (2) a uniform input distribution formed by shuffling distinct integers using the Mersenne Twister 19937 random number generator. Many QuickXsort variants were introduced by Edelkamp, Weiß, and Wild [55], and by Edelkamp and Weiß [54]. We picked the Hybrid QuickMergeSort (Hybrid QMS) variant, which is the most competitive variant described in [54]. We implemented TimSort as described in Algorithm 3 of [10].

For simplicity, we used `std::merge` from the C++ standard library for all the canonical stack based mergesorts, and `std::inplace_merge` for the in-place versions. While the former is guaranteed to run in $O(n)$ time, C++'s in-place merge algorithm has a worst case time

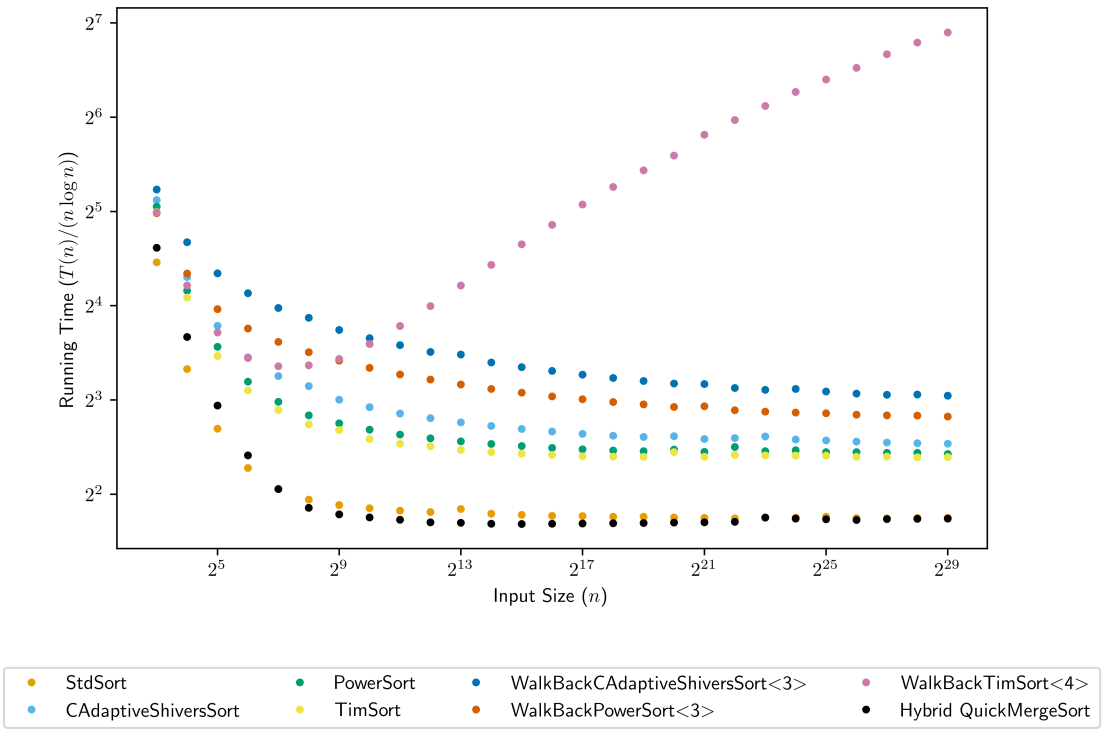

Figure 4.13: A comparison of normalized running times on uniformly random inputs. Since uniformly random inputs are expected to have a Shannon entropy of $\Theta(\log n)$, we scaled the running times by $n \log n$, such that any algorithm that is optimal with respect to Shannon entropy should appear constant. As expected, `std::sort` (StdSort) and Hybrid QMS are optimized well for these inputs. Reassuringly, not only do all the stack-based mergesorts appear constant, but also all our in-place versions of the *walkable* mergesorts appear constant. The only mergesort which does not appear constant is the in-place version of TimSort, which is not walkable, as shown in Section 4.5.

complexity of $O(n \log n)$, although our experiments suggest that this worst-case condition was not reached. Another detail is that none of our inputs allowed duplicate elements, although preliminary experiments suggest that this does not have a significant impact on the running time of the algorithms. The code used for our experiments can be found on GitHub [72].

4.8.2 Results

In Figure 4.12, we compare the algorithms on the TimSort counterexample input as described in Section 4.5 and Section 4.5.1. This input has a constant Shannon entropy, so any algo-

rithm that is instance-optimal with respect to Shannon entropy should run in linear time, and after scaling the running times by n, should appear constant. Indeed, we see that all the standard stack-based mergesorts take advantage of the natural ordering of the input and appear constant, while C++'s `std::sort` function (using introsort) and QuickXsort, which are not instance-optimal, perform poorly. Reassuringly, after applying our simple walk-back algorithm, the in-place versions of the *walkable* mergesorts (PowerSort and c-Adaptive ShiversSort) appear to remain input-sensitive like their non-in-place counterparts. In contrast, the in-place version of TimSort, which as we proved in Section 4.5 is not walkable, performs poorly.

In Figure 4.13, we compare the algorithms on uniformly random inputs, which are expected to have a Shannon entropy of $\Theta(\log n)$, so we scale the running times by $n \log n$, such that any $O(n \log n)$ algorithm should appear constant. And indeed, all the algorithms (notably except the version of TimSort implemented with the walk-back algorithm) appear constant, as expected. The fact that walk-back TimSort appears to perform worse than $O(n \log n)$ on such inputs is interesting.

4.9 Supplemental Experiments

4.9.1 Impact of Different Stack Sizes

In the previous section, we gave each in-place variant their minimal stack size required, k, which for PowerSort and c-Adaptive ShiversSort is 3, and for TimSort is 4. We now compare each in-place algorithm as its stack size increases from k to 5, to 10, against the standard version (which can be thought of as having stacks of size up to $\Theta(\log n)$).

As expected, all the in-place versions of our stack-based natural mergesorts perform better as their stack size increases, performing the best when the stack size is not bounded. The walkable mergesorts in particular perform very well, performing within a very small constant

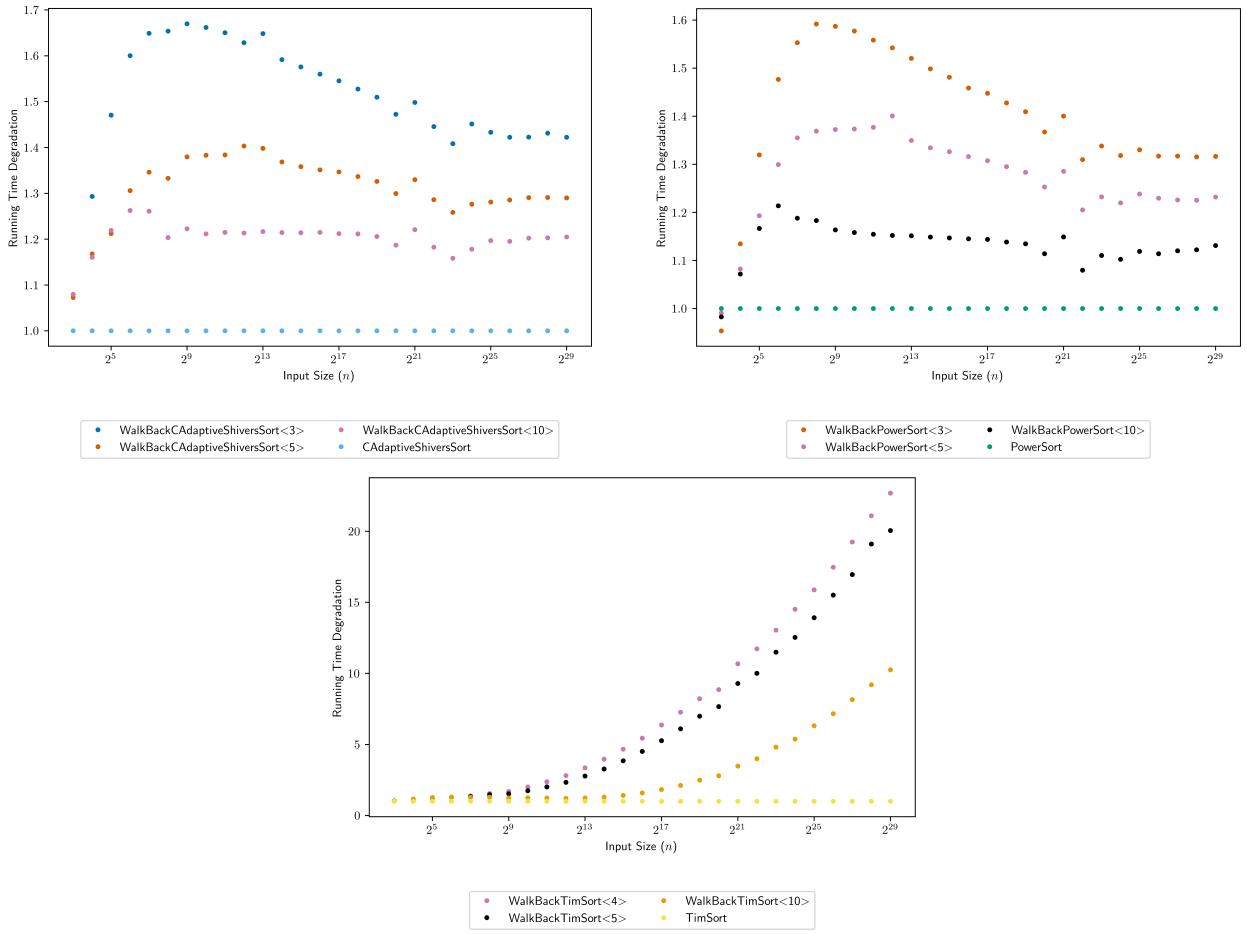

Figure 4.14: A comparison of the different in-place versions of each algorithm against their standard version on uniformly random inputs. Each algorithm is measured with respect to the standard version, which is normalized to 1. While the in-place versions of c-Adaptive ShiversSort and PowerSort are within a small constant factor of the standard version, the in-place versions of TimSort clearly perform asymptotically worse.

factor of the standard version. Even for the smallest stack size of just 3, the in-place versions of PowerSort and c-Adaptive ShiversSort perform within a factor of 1.7 and 1.6, respectively, of the standard version. This further confirms that not only is our simple walk-back algorithm effective, but that it is likely benefiting from being very cache-friendly. In contrast, despite the cache-friendliness of our walk-back algorithm, the in-place version of TimSort clearly does not keep up with the standard version. Even when allowed to store the top 10 runs, it is only able to keep up with the standard version for so long before diverging. See Figure 4.14. This directly aligns with our results from Section 4.5.

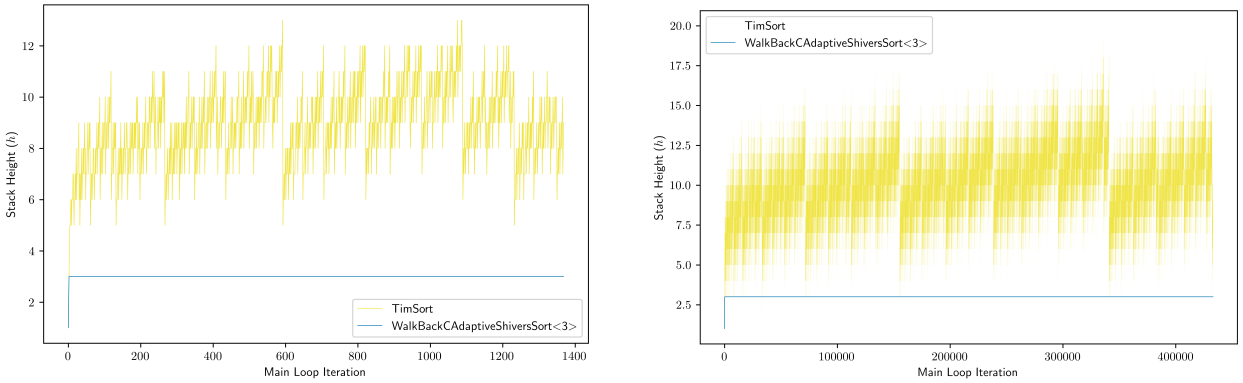

Figure 4.15: Here we track TimSort's stack height (h) during each iteration of the main loop, right after a new run is pushed onto the stack S for arrays of size $n = 2^{20}$. On the left, we use our TimSort counterexample input as described in Section 4.5 while on the right we use uniformly random inputs.

4.9.2 Stack Height Variability

In Figure 4.15, we plot TimSort's stack height (h) during each iteration of the main loop, right after a new run is pushed onto the stack S for arrays of size $n = 2^{20}$ for both our bad TimSort example (left) and uniformly random inputs (right), and compare it to the stack size of the in-place version of c-Adaptive ShiversSort. It is clear to see from both plots how TimSort's stack size often significantly fluctuates in hard-to-predict ways, while the in-place version of c-Adaptive ShiversSort has a very consistent stack size, bounded by 3. These simple figures demonstrate the predictability of our in-place versions, which may lead to their implementations being less error-prone.

Chapter 5

Raiders of the Lost Log: Synchronous Parallel In-Place Models and Algorithms

5.1 Background

Much of the focus in parallel algorithm design has been on finding parallel algorithms that minimize work, the total number of operations performed, and span, the time until the last processor terminates. However, data processing tasks have increased in size by orders of magnitude over the past decade. Machine rental and energy costs grow linearly with a machine's memory capacity [80]. Furthermore, memory bandwidth/latency, cache misses, and page faults all present larger bottlenecks to performance and scalability if shared memory usage is high. Thus, memory usage in Big Data tasks becomes a concern equally important to runtime. This issue is magnified when dealing with shared-nothing data processing frameworks such as MapReduce, Hadoop, and Spark since shared working set memory is not locally accessible but rather distributed over the network. Many classic parallel algo-

rithms, despite having excellent performance in theory, require $\Omega(n)$ auxiliary memory (see, e.g., [9, 23, 25, 26, 74, 141]). As a result, they scale poorly in practice.

In response, researchers have developed *Parallel In-Place* (PIP) algorithms [80, 81, 82, 102, 114, 143], which restrict auxiliary space usage. Notably, work by Gu, Obeya, and Shun [80] synthesizes past efforts and introduces two models for parallel in-place processing: Strong PIP and Relaxed PIP. A shared goal of these models is to strike a balance between parallelism and space usage by requiring PIP algorithms to scale down to space-efficient sequential algorithms. Specifically, in the *Strong PIP* model an algorithm (1) allocates no shared memory aside from the input, (2) uses $O(\log n)$ private memory per processor, (3) has polylogarithmic span, and (4) uses at most $O(\log n)$ space when run sequentially. In the *Relaxed PIP* model an algorithm (1) allocates $O(t)$ auxiliary shared memory, (2) uses $O(\log n)$ private memory per processor, and (3) has $O((n/t)\text{polylog}(n))$ span, where t denotes a cap on auxiliary memory. Importantly, these PIP models are specializations of the *software parallel* asynchronous binary-forking model of parallel computation, which is based on binary "fork" operations that exist in some parallel programming languages.

We are instead interested in *hardware parallel* models for in-place computations, however, where we have a specified number, p, of processors, where p lies somewhere in between 1 and the input size, n. Indeed, providing efficient algorithms that do not allocate additional memory for these intermediate processor counts is the main motivation for our new model, *Synchronous Strict PIP*, which is a specialization of the well-known hardware-parallel PRAM model of computation.

5.1.1 Our Results

In this chapter, we introduce a family of *Synchronous Strict PIP* models.

Definition 5.1 (Synchronous Strict PIP Model)**.** *Let $1 \leq p \leq n$ be the number of processors available. A PRAM algorithm is Synchronous Strict PIP if it (1) allocates no shared memory aside from the input, (2) uses $\Theta(1)$ private memory per processor, (3) has $O((n/p)\text{polylog}(n))$ span, and (4) uses at most $\Theta(1)$ additional space, i.e., is strictly in-place, when run sequentially.*

Observe that the Synchronous Strict PIP model admits synchronous parallel algorithms with strictly stronger memory guarantees than those under the asynchronous Strong PIP and Relaxed PIP models. Indeed, Synchronous Strict PIP retains the flexibility of the Relaxed PIP model by allowing a range of processor counts while providing space guarantees that are even stronger than the Strong PIP model. To the best of our knowledge, there have not been previous algorithms defined in terms of this Synchronous Strict PIP model. The work that comes closest is a parallel random permutation algorithm of Hutton and Melrod [89]. When emulated on a CRCW PRAM, their algorithm satisfies our conditions (1), (2), and (4), but not (3).

To show the effectiveness of our hardware-based models, we give several efficient Synchronous Strict PIP algorithms for fundamental parallel problems. Our results are summarized in Table 5.1. In Sections 5.2 and 5.3, we introduce the *IJ swap*, an algorithmic technique used throughout the work and give several parallel in-place algorithms analyzed in the work-span framework. In Section 5.4, we present our second main contribution, the *parallel-augmented sweep* paradigm. This is an algorithmic insight that allows us to efficiently schedule our in-place algorithms analyzed in the work-span framework to run on $p < n$ processors while maintaining the same in-place guarantees.

We also provide an orthogonal result that is of independent interest. We apply a novel balls in bins analysis to develop our Relaxed PIP SampleSort, the first PIP comparison sorting algorithm with optimal span and optimal work (cf. [13, 143]).

Algorithm	Work-Efficient	PIP
Prefix*	✓	Synch. Strict
Packing/Partition*	✓	Synch. Strict
QuickSort	✓	Strong
List Contraction*	✓	Synch. Strict
Tree Contraction*	✓	Synch. Strict
Rand. Permutation*	✓	Synch. Strict
Convex Hull	✓	Strong
Convex Hull Presorted*		Synch. Strict
SampleSort*	✓	Relaxed

Table 5.1: A summary of our new parallel in-place algorithms. We indicate our main results with *.

5.2 The Indiana Jones Swap

In this section, we introduce the *Indiana Jones* (IJ) swap, which is a technique used throughout this chapter. This technique was inspired by an iconic scene in the movie, *Raiders of the Lost Ark*, in which Indiana Jones attempts to steal a Golden Idol by swapping it with a bag of sand. See Figure 5.1.

Figure 5.1: A still from *Raiders of the Lost Ark*. Directed by Steven Spielberg, Lucasfilm Ltd. and Paramount Pictures, 1981. This image is from copyrighted material, the use of which has not been specifically authorized by the copyright owner. The author(s) have determined this to be fair use of the copyrighted material as referenced and provided for in § 107 of the U.S. Copyright Law.

Our technique similarly involves each processor strategically swapping data between shared memory and its private stash. If applied with care, we do not need to allocate any more

shared memory than what is required to store the input. For the remainder of this section, we demonstrate the IJ swap and sub-techniques we call *stashing* and *windowing* by implementing fundamental parallel algorithms in-place.

5.2.1 Parallel Prefix

The stashing technique simply involves processors making a copy of the element at their respective indices of the input array, copying it into their private stash memory. Such a step requires constant span and work proportional to the number of processors. We frequently use this operation to preserve values that would be destroyed or to free up space for some in-place computation.

Now we consider the parallel prefix problem. We begin by observing that the recent Strong PIP prefix algorithm of Gu, Obeya, and Shun [80] can be easily adapted to our framework. Their algorithm is a simple variant of the standard work-efficient tree-based parallel prefix algorithm [21], which we described in Section 1.4.1. In general, this algorithm takes $O(n)$ extra space. Gu, Obeya, and Shun [80] make the simple observation that we can overwrite values of the original array to maintain the intermediate sums. In the up-sweep, whenever we are summing elements a and b, we simply overwrite b with $a + b$. The root of the tree is held at the end of the array. In the down-sweep, the nodes of the "binary tree" can be recovered by index manipulation. At depth i, each node's right child is at its own index and its left child is $n/2^{i+1}$ indices to the left. They show that both passes still take $O(\log n)$ span and $O(n)$ work.

We first observe that their algorithm is "destructive", in the sense that it overwrites the array's original contents after performing the operation. We can remedy this by performing an initial stashing operation taking $O(1)$ span and $O(n)$ work to preserve the original values in each processor's private memory. Notably, this allows us to implement packing optimally, which we show in the following section.

Using standard techniques, we observe that their algorithm can be implemented in the EREW PRAM model with $O(\log n)$ span and $O(n)$ work. Furthermore, we note that computation is only done prior to each recursive call in both the up-sweep and down-sweep of the algorithm. This means that processors can simply terminate when they finish work at the bottom of the recursive tree and so do not have to maintain any information about prior recursive calls. Thus, if there are $\Theta(n)$ processors, each processor only needs $O(1)$ private memory to maintain information relevant to their current recursive call as well as the stashed original data.

Theorem 5.1. *There exists an algorithm that performs parallel prefix with $O(\log n)$ span and $O(n)$ work in the EREW PRAM model when each of n processors has $O(1)$ private memory.*

As Gu, Obeya, and Shun note, the naive serialization of this algorithm gives a sequential algorithm that consumes $O(\log n)$ space due to recursive overhead [80]. In Section 5.4, we apply the parallel-augmented sweep technique to generalize the above algorithm such that it is efficient and in-place for all processor counts $p \leq n$. This naturally implies a strictly in-place sequential algorithm.

5.2.2 Packing and Parallel Partition

Recently, Kuszmaul and Westover [101] used inversion encoding to implement a parallel partition algorithm in-place in the EREW PRAM model. However, their implementation takes $O(\log n \log\log n)$ span, assumes $O(\text{polylog } n)$ private space per processor, and requires a complicated workaround to handle duplicate elements. In this section, we present a simple algorithm using the IJ swap that has optimal span, requires only $O(1)$ space per processor, and handles duplicate elements automatically. As an aside, our partition algorithm immediately admits the first optimal, $O(\log^2 n)$-span Strong PIP QuickSort algorithm in the EREW PRAM.

Before discussing the parallel partition problem, let us first discuss a related problem called *packing*. Recall from Section 1.4.1, in the packing problem we are given an array of length n containing 0's and 1's in any order and we wish to pack the 1's such that they appear in the beginning of the array. Zhang and Rao [141] give a simple algorithm in which we perform a prefix sum on the array, which reveals a unique index for each 1 that packs them in the beginning of the array. A symmetric operation gives us a place to write each 0. To make this in-place, we simply apply stashing to preserve each value such that they are not destroyed by the prefix sum.

Then the algorithm consists of a constant number of stashing and swapping operations, each of which take $O(1)$ span and $O(n)$ work and use at most $O(1)$ words of private memory per processor. In addition, the algorithm performs a constant number of in-place parallel prefixes, which take $O(\log n)$ span and $O(n)$ work and use $O(1)$ private memory per processor. We conclude with the following.

Theorem 5.2. *There exists an algorithm that performs parallel packing with $O(\log n)$ span and $O(n)$ work in the EREW PRAM model when each of n processors has $O(1)$ private memory.*

In the parallel partition problem, we are given a list of n elements and a pivot q. We wish to return a list such that all elements less than q are to its left in the array and all elements greater than or equal to q are to its right. Assume that q is initialized as the last element in the array. Observe that we can reduce the partitioning problem to parallel packing in which an element is assigned a 1 if it is smaller than the pivot and a 0 otherwise. Thus, we immediately have the following.

Corollary 5.3. *There exists an algorithm the performs parallel partition in-place in $O(n)$ work and $O(\log n)$ span in the EREW PRAM model when each of n processors has $O(1)$ private memory.*

5.2.3 Deterministic Reservations and Windowing

In this section, we describe our second sub-technique, *windowing*. We do so by implementing in-place algorithms that perform list contraction and tree contraction in polylogarithmic span and linear work for the EREW PRAM.

We adapt algorithms from Shun, Gu, Blelloch, Fineman, and Gibbons [132], who developed their algorithms using a *deterministic reservation* framework, a paradigm in which parallel algorithms hew closely to their sequential counterparts [22, 24, 80, 130, 131, 132]. We review this algorithm design paradigm in Section 1.4.1.

In the context of list contraction and tree contraction, conflicts occur when two processors attempt to contract the same node. In both problems, Shun *et al.* handle this by initializing a reservation boolean array R. In each round of the algorithm, if the processor has an operation to commit, they first reserve their location in R. Then, if their reservation holds, they subsequently commit the operation by manipulating data in L. See Algorithm 10.

Algorithm 10 Non-In-Place List Contraction Algorithm of [132].

```
1:  Input: Array L of size 3n containing n nodes and 2n pointers to each node's predecessor
    and successor.
2:  Output: Contracted list L.
3:  R = {0, ..., 0}                                              ▷ boolean array
4:  procedure RESERVE(i)
5:      if i < L[i].prev and i < L[i].next then
6:          R[i] = 1                                             ▷ reserve own location
7:      return 1
8:  procedure COMMIT(i)
9:      if R[i] = 1 then
10:         if L[i].prev ≠ null then
11:             L[L[i].prev].next ← L[i].next
12:         if L[i].next ≠ null then
13:             L[L[i].next].prev ← L[i].prev
14:         return 0
15:     else
16:         return 1
```

In Shun *et al.*'s list contraction algorithm [132], the main operation that is not in-place involves the allocation of R. We make the trivial observation that, rather than allocating a new array R, we can simply use L to store reservations. Have processor p_i temporarily swap $L[i]$ with its reservation value (either a 1 or 0) after the RESERVE phase. Then, after the first if statement in the COMMIT phase, we can swap the original values of L back in and proceed as normal. We call this temporary revealing of metadata *windowing*.

After each round of reserving and committing in Algorithm 10, we perform a packing step to compact all the processors that have not yet committed their operation [132]. We can free up space to do this by temporarily stashing the first p elements in L, where p is the current number of active processors, and then applying our in-place packing algorithm from Section 5.2.2.

An analysis by Shun *et al.* [132] shows that this algorithm takes $O(\log n)$ rounds with high probability in n,[1] assuming the ordering of L is randomized. Standard techniques can be used to ensure that all reads and writes are exclusive. Thus, we have the following.

Theorem 5.4. *There exists an algorithm that solves list contraction in the EREW PRAM model when each of n initial processors has $O(1)$ private memory in $O(\log^2 n)$ span w.h.p. and $O(n)$ work in expectation, assuming the list L is in random order.*

Their tree contraction algorithm has an almost identical structure [132], so the same technique implies the following.

Corollary 5.5. *There exists an algorithm that solves tree contraction (on complete binary trees with randomly-ordered leaves) in the EREW PRAM model when each of n initial processors has $O(1)$ private memory in $O(\log^2 n)$ span w.h.p. and $O(n)$ work in expectation.*

[1] An algorithm succeeds with high probability in n (w.h.p.) if it succeeds with probability greater than $1 - 1/n^c$ for some constant $c \geq 1$.

5.3 Efficient Parallel Algorithms with Constant Space Per Processor

In this section, we demonstrate the IJ swap further by implementing non-trivial algorithms for in-place random permutations, 2D convex hulls, and sorting.

5.3.1 Random Permutation

We now adapt another deterministic reservations algorithm of Shun, Gu, Blelloch, Fineman, and Gibbons [132], this time for computing a random permutation of an array, A. Their algorithm is a parallelization of the sequential Knuth shuffle algorithm for the priority-write CRCW PRAM. In the Knuth shuffle, we iterate backwards through the array and repeatedly swap current element $A[i]$ with some randomly selected element from indices 0 to i. Shun *et al.*'s parallelized version works like so. Given n processors, their algorithm assigns each processor to an array index i. We have them each generate $H[i]$, a random value between 0 and i. Each processor's goal is to commit a swap between $A[i]$ and $A[H[i]]$. However, swaps that overlap the same index are in conflict and cannot be done concurrently. To solve this, the algorithm of Shun *et al.* uses reservations.

At the beginning of each round, we initialize an array of reservations, R, in which all values are set to -1. Each processor p_i "reserves" their swap by writing i to $R[i]$ and $R[H[i]]$. This is a priority write operation, so the maximum value of i written concurrently is preserved. Each processor p_i then reads $R[i]$ and $R[H[i]]$. If their two reservations were preserved, then processor p_i commits their swap in A. Shun *et al.* show that the depth of these conflicts is at most $O(\log n)$ w.h.p. [132]. As a result, after $O(\log n)$ rounds, every processor will have committed their swap w.h.p. Once again, [132] use packing after each round to limit the work of inactive processors.

Using the IJ swap, we can make their algorithm in-place. Just like our list contraction and tree contraction algorithms, we can perform windowing on the input array A to simulate R. However, we must be more careful since processors can read and write to multiple, sometimes overlapping, locations of R rather than just a single location. Each active processor p_i begins each round by stashing $A[i]$ and $A[H[i]]$ in private memory. Then each p_i writes a -1 to $A[i]$ and $A[H[i]]$ concurrently. Since each index that an active processor intends to reserve is set to a -1, we can act as if A is the reservation array R. Next, each p_i performs a priority write to reserve $A[i]$ and $A[H[i]]$ in parallel. If a processor observes that its reservation was overwritten, it immediately writes its stashed values back to their original spots. Finally, the processors with successful reservations write their stashed values back in swapped order.

As with list and tree contraction, we use our in-place packing method after each round. Stashing and windowing add $O(1)$ span to each round. Since only active processors perform such steps, we can charge the extra work done as a constant multiple of the work done in the original algorithm to read and write reservations. As a result, total work increases by no more than a constant factor. In combination with a runtime analysis by Shun *et al.* [132], we conclude the following.

Theorem 5.6. *There exists an algorithm that computes a random permutation in the priority-write CRCW PRAM model when each of n initial processors has $O(1)$ private memory in $O(\log^2 n)$ span w.h.p. and $O(n)$ work in expectation.*

5.3.2 Convex Hull

We next show how to construct the upper hull of a set of n points in the plane in-place, assuming the input is sorted by its x-coordinate. Our algorithm requires $O(\log n)$ span and $O(n \log n)$ work in the CREW PRAM model. The lower hull can be computed symmetrically.[2] Our key insight is to represent each intermediate hull as a permutation of its hull

[2]If points are sorted radially, the entire convex hull can be recovered at once (see, e.g., [119]). We consider the upper hull for simplicity.

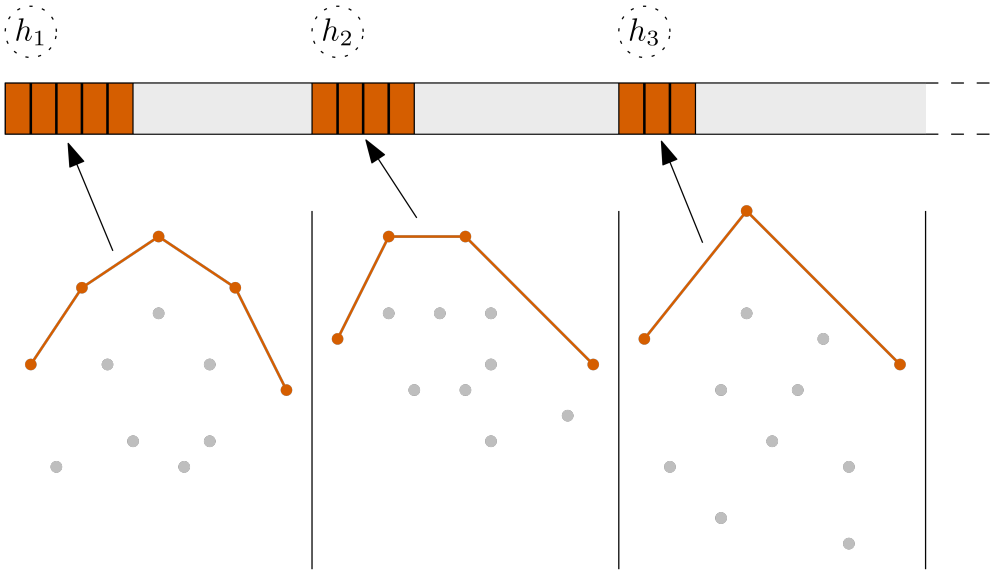

Figure 5.2: The combine step for our in-place convex hull algorithm. Intermediate hulls are represented in-place with hull points packed the left of their subarray.

points, rather than in an external data structure. Indeed, this technique is used often in sequential in-place algorithms for computational geometry [29, 30, 32, 33, 38, 39]. Surprisingly, we believe we are the first to do so in parallel computational geometry. We also present a method to maintain recursive metadata despite having only constant space per processor. For simplicity, we assume that points are in general position and that no two points share the same x-coordinate. These assumptions may be lifted via standard techniques [110].

Our algorithm is based on the parallel 2D convex hull algorithm of Goodrich and Atallah [74], which we review in Section 1.4.3. The original algorithm uses extra space to maintain each intermediate hull such that they can be binary-searched, split, and joined in parallel. For example, they could be represented as binary search trees ordered by x-coordinate. We cannot afford this extra $O(n)$ space. Instead, we maintain upper hulls of each subproblem in-place by rearranging points in the input array.

For example, say that subproblem S_i has an upper hull of h_i points. Let us assume that, when S_i returns to its parent subproblem, the first h_i entries in S_i's section of the array represent its upper-hull points ordered by their x-coordinate (the remaining $|S_i| - h_i$ points

may be in arbitrary order). Furthermore, let us assume that the first processor assigned to S_i has h_i stored in its private stash. See Figure 5.2. Assuming this is true for all subproblems, we will show that we can construct the upper hull of their parent subproblem in-place in $O(\log m)$ span and $O(m \log m)$ work, where $m = \sum_i |S_i|$.

The first step in combining all subproblems is to assign each processor a pair of upper hulls for which to compute an upper tangent. This pairing can be computed independently by each processor by examining their index with respect to the current problem size. Then each processor must perform a double binary search on a pair of hulls. The first processors of each subproblem S_i can use windowing to temporarily reveal h_i in the first $\sqrt{m}$ array indices of the parent subproblem. Processors that access S_i's upper hull can stash h_i and use it to determine the middle element of S_i's hull. Then the first processor of each subproblem can swap h_i back into their private memory and we can perform concurrent double binary searches as usual on each hull's packed array representation.

After each processor determines an upper tangent, they can stash their index's point and write the slope of the computed tangent to their index. Processors from each subproblem S_i can in parallel perform two max-finds to determine the largest-slope tangent lines coming from the left and from the right of S_i via our in-place prefix algorithm. With this information, each intermediate hull is reduced to a (possibly empty) convex chain, which we must combine into a single upper hull of h points which represents the combined subproblems. This can be done easily via in-place packing. Each processor assigns a 0 to their point if it is not part of the new upper hull and a 1 if it is. By properties of packing, our packed array preserves the ordering of the hull points. The first processor stashes the number of hull points, h, and we return.

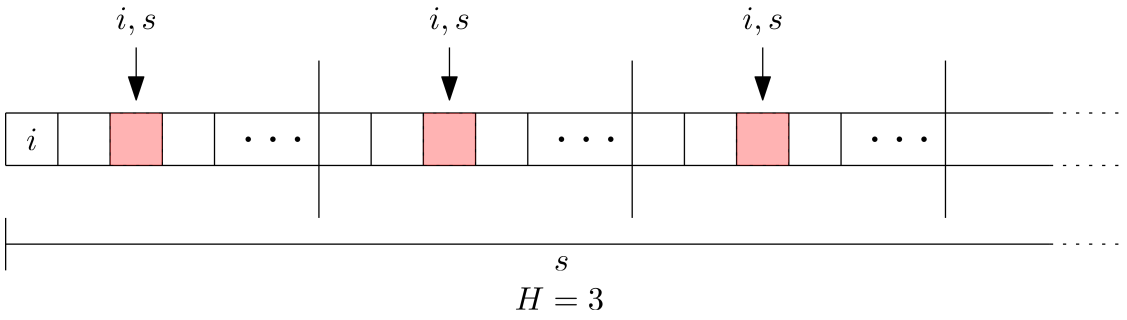

Figure 5.3: An example of recursive metadata load sharing. We are at a parent problem at recursive depth $H = 3$ about to recurse into several child subproblems. Metadata (starting index and problem size) for the parent subproblem is stored at index 3 of each child subproblem. By definition of the recursion and by the fact that all subproblems are size $> \alpha \log \log n$, each index of our input array, and so each processor, is associated with exactly one metadata storage.

Recursive metadata load-sharing

The last issue is how to maintain metadata about recursion such that processors can efficiently return to parent problems. Naively, we can have each processor maintain information about each subproblem they are a part of, but this blows up space usage to $\alpha \log \log n$ per processor for some constant α, where $\alpha \log \log n$ is the number of levels of recursion. Instead, we can spread the load across several processors.

First, we note that all subproblems of size $\leq \alpha \log \log n$ can be computed via in-place Graham scan in $O(\log \log n)$ span and $O(n)$ work since the points are already sorted [33]. Thus, doing so does not increase the asymptotic complexity of the algorithm. Let H be our current level of recursion, starting with $H = 1$. Say that we are at some parent subproblem of size m and depth H and about to recurse into its children. Just before recursing, have the H-th processor of each child subproblem store two values: the position of the first processor of the parent problem and m. See Figure 5.3.

After computing the upper hulls of each child subproblem and just before returning to the parent problem, have the H-th processor in each child subproblem window the position and size information of the parent problem. Each processor can maintain their current depth

$(H+1)$ with constant storage, so it takes $O(1)$ span and $O(m)$ total work across all the child subproblems to broadcast the parent problem's position and size information.

From the observation above, we can assume each subproblem above the base case is size $> \alpha \log\log n$. Then by construction of the original algorithm and our load-sharing procedure, each index of the array is associated with at most a single metadata storage. As a result, each processor stores at most a constant amount of recursive metadata.

The cost of the rest of the combine step in the parent subproblem is dominated by the binary searches, which take $O(\log m)$ span and $O(m \log m)$ work. These time bounds match the original algorithm's bounds asymptotically. We use their runtime analysis to conclude that the span of the algorithm is $O(\log n)$ and its work is $O(n \log n)$ [5, 74].

Theorem 5.7. *There exists an algorithm to compute the 2D convex hull of n presorted points in the plane in $O(\log n)$ span and $O(n \log n)$ work for the CREW PRAM when each of n processors has $O(1)$ private memory.*

5.3.3 SampleSort

In this section, we apply an analysis of a new variant of parallel balls in bins to prove the following.

Theorem 5.8. *We can implement an algorithm that sorts n elements in $O(\log n)$ span with high probability and $O(n \log n)$ expected work for the arbitrary CRCW PRAM when each of n processors has $O(1)$ private memory.*

SampleSort is a generalization of randomized quicksort in which we sample more than one pivot [13, 28, 64]. We review the algorithm in detail in Section 1.4.1. We briefly observe that sorting the $n^{1/3} - 1$ sampled pivots can be done by performing stashing to free the first $O(n^{2/3})$ elements of the array and performing a quadratic sorting method that compares

every pairwise element in $O(\log n)$ span and $o(n)$ work. We can then pack the pivots in sorted order to the beginning of the array and perform parallel binary searches to assign each remaining element a bucket index in-place.

The main issue arises in the distribution step. The CREW PRAM integer sorting method presented in [91, Section 9.6.4] does not appear to be a good candidate for an in-place algorithm, as it involves building a balanced binary tree over the input elements. The second algorithm we present for the CRCW PRAM, which was proposed by Blelloch *et al.* [23], seems more promising. Recall that in this algorithm, we allocate arrays of size $2cn^{2/3}$ to serve as the buckets between each pivot. Each processor then attempts to write to a randomly chosen array location of their element's corresponding bucket until they are able to place it in a free location. Since each bucket is guaranteed to have size $cn^{2/3}$ w.h.p. (Lemma 1.4, for $c = 5$), each bucket will half-empty and so each attempt will succeed with probability $1/2$. An analysis based on coin flips shows that this takes $O(\log n)$ span and $O(n)$ work w.h.p. (see Lemmas 1.5 and 1.6). However, this requires $O(n)$ additional words of shared memory. As a result, we must modify this strategy.

Distributing Elements In-Place

Our in-place algorithm to distribute elements into buckets begins by having each processor stash the element originally at their index. We then partition the input array evenly into groups of size roughly $n^{2/3}$, each of which represents a bucket. However, as we noted above, each bucket may contain up to $cn^{2/3}$ elements. Thus the array itself may not have enough space to contain the bucket's elements. We can solve this by using each processor's $O(1)$-sized stash to augment the capacity of each bucket.

Our method proceeds in rounds. See Figure 5.4 for a visual of the following description. Each processor takes their initially stashed element e and attempts to write it to a random location j in the section of the array corresponding to the bucket it belongs to. Write conflicts are

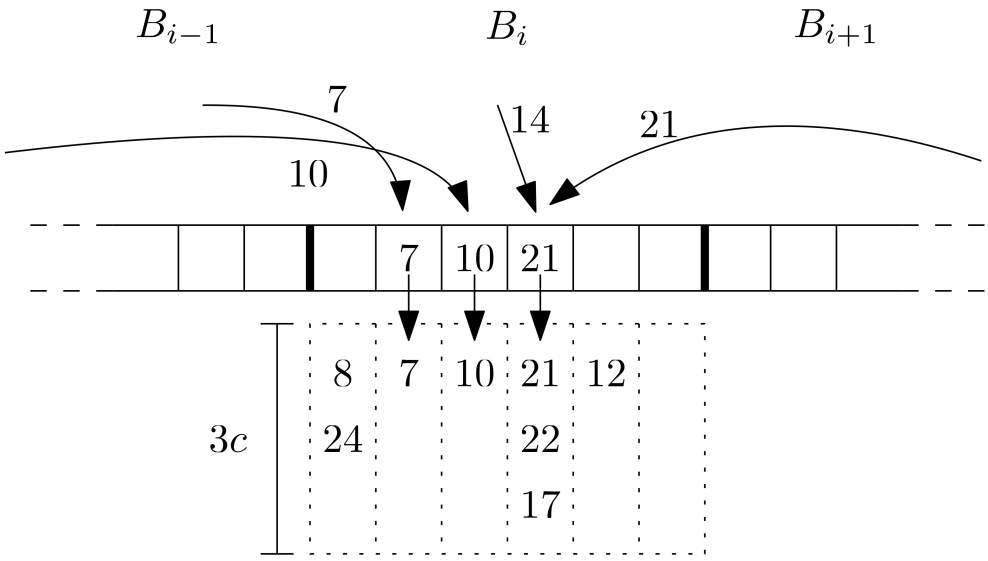

Figure 5.4: Our in-place distribution algorithm for sample sort. At each step, every element that belongs in B_i tries to write itself to a random location in B_i. Write conflicts (e.g., with 14 and 21) are resolved arbitrarily. At the end of the round, each entry copies the element that lands there into its stash. It does so until it reaches its stash size of $3c$.

dealt with arbitrarily. After the write step, each processor p_i reads the element at their own index i. They copy $A[i]$ into their stash so long as they have less than $3c$ elements already stashed. If they are at capacity, they write a null value to $A[i]$ to indicate that they did not stash anything. Processors then read j to determine if they have successfully written their element e.

By augmenting the capacity of each bucket index with $3c$ elements of private storage, each bucket can store $3cn^{2/3}$ elements, which is enough space to store all elements w.h.p. in n.

After the distribution process completes, each processor counts the number of elements they have stashed. Similar to parallel partition, we can perform a prefix sum on these counts. The information can be used to assign each element a unique position in the array such that all elements in the same bucket are contiguous. Each element writes to those locations in the input array and then we recurse into each bucket. These final steps take $O(\log n)$ span and $O(n)$ work.

It remains to show that our distribution algorithm terminates in $O(\log n)$ rounds and performs $O(n)$ work w.h.p.

Congested Parallel Balls into Bins

Our algorithm can be mapped to a parallel variant of balls into bins, where the balls are the elements that need to be distributed and the bins are the array indices. Several works have already studied variants of parallel balls into bins, often in the context of load balancing tasks on machines. See, e.g., [1, 2, 17, 18, 19, 105, 106, 134]. However, it appears that our setting is unique in that all remaining balls are thrown at once yet bins accept at most one ball each round, even if their remaining capacity is more than one.

We call this variant the *congested parallel balls into bins* problem. Formally, we have m balls and b bins. Each bin has a threshold $t \geq m/b$ and a *congestion factor* g. Each round, we throw all remaining balls uniformly into the n bins. Bins try to accept as many balls that land in it, subject to the following constraints. Bins can accept no more than g balls each round, and bins can accept no more than t balls in total. Balls that are not accepted by a bin must be thrown again in the next round. For a given instantiation of m, b, t, and g, we are interested in knowing how many rounds it takes until all balls are accepted by a bin w.h.p. in the number of bins b.

Let us first focus on a single bucket. The matching instance of the congested parallel balls into bins problem has $b = n^{2/3}$ be the number of bins available, $m \leq cb$, $t = 3c$, and $g = 1$. We will show that after $O(\log b)$ rounds, all m balls will be allocated w.h.p. in b.

Our approach defines two (implicit) phases of the distribution process. The first phase concerns the first $3c$ rounds of the process. Since each bin has capacity $3c$, every bin in the first $3c$ rounds must accept a ball if it receives one. Due to this property, we show that a large majority of the balls are placed in this phase. The remaining rounds form the second phase,

in which all straggler balls are placed. We show that the second phase takes an additional $O(\log b)$ rounds w.h.p. in b.

Lemma 5.9. *After $3c$ iterations, there will be at most $b/2$ balls remaining w.h.p. in b, the number of bins.*

Proof. We first observe that, since $g = 1$ and $t = 3c$, no bin can exceed its threshold for the first $3c$ rounds. As a result, any bin that receives one or more balls in the first $3c$ rounds must accept a ball.

Let $m' \leq m$ be the number of balls thrown at the beginning of a given round. Each ball lands in a bin with probability $1/b$. The chance a bin is not hit by any ball is $(1 - 1/b)^{m'}$. Assume that $m' > b/2$ (otherwise, we are done). Then $(1 - 1/b)^{m'} \leq (1 - 1/b)^{b/2} \leq 1/\sqrt{e}$. As a result, we expect at most $b/\sqrt{e}$ bins to not be hit by a single ball each round.

Let X be a random variable representing the number of bins not hit this iteration. Applying a Chernoff bound, we see that

$$\Pr[X \geq (1 + \delta) \times b/\sqrt{e}] \leq e^{-b\delta^2/4\sqrt{e}}.$$

This holds with exponential probability[3] in b for any $\delta > 0$. For concreteness, we set $\delta = 2\sqrt{e}/3 - 1$. This implies that no more than $2/3$ of the bins will fail to be hit with exponential probability in b.

As a consequence, when $m' > b/2$, at least $b/3$ bins will be hit with one or more balls with high probability. Since no bin hits their threshold until round $3c$ at the earliest, this means that bins accept at least $b/3$ balls in a given round w.h.p. By a union bound over the first

[3]An event succeeds with exponential probability in n if it succeeds with probability $1 - c_1^{-c_2 n}$, for some constants $c_1 > 1$, c_2.

$3c$ rounds, this holds with probability $1 - 3ce^{-O(b)}$ over all rounds, which is still exponential in b. $m - 3c \times b/3 \leq cb - 3c \times b/3 < b/2$, so we have the desired result. □

Lemma 5.10. *When there are $b/2$ balls remaining, it takes $O(\log b)$ rounds until all balls are accepted by a bin w.h.p. in b.*

Proof. We will first show that each ball has a constant probability of being accepted by a bin each round. Because our threshold is set to $3c$ and $m \leq bc$, no more than $b/3$ bins can ever be at their threshold. For simplicity, we assume that exactly $b/3$ bins are at their threshold and cannot accept any new balls.

Since the balls are thrown uniformly, each lands in a bin with space available with probability at least $2/3$. Let us assume that a ball must land by itself in an available bin to be accepted.[4] If a pair of balls is thrown uniformly at b bins, they collide with probability $1/b$. If we fix one of the balls, the probability that it collides with another ball is at most $(b/2 - 1)/b < 1/2$. Thus, each ball lands by itself in an available bin with probability at least $1/2 \times 2/3 = 1/3$.

We can model this event with biased coin flips, where the probability of success is $1/3$. A simple modification of Lemmas 1.5 and 1.6 shows that the algorithm takes $O(\log b)$ rounds w.h.p. in b and no more than $O(b)$ balls are thrown with probability $1 - 2^{O(b)}$.

□

Theorem 5.11. *Our algorithm to distribute elements in-place succeeds after $O(\log n)$ rounds w.h.p. in n.*

[4]This is a pessimistic assumption. In reality, the ball might get lucky and get picked arbitrarily if it lands with one or more other balls.

Proof. This follows from Lemma 5.9 and Lemma 5.10. Since $b = n^{2/3}$, small adjustments in constant factors of the number of rounds can be made to ensure that both lemmas succeed w.h.p. in n, even after taking a union bound over all $O(n^{1/3})$ buckets. □

By Theorem 5.11 and our initial discussion, our algorithm takes $O(\log n)$ span and $O(n \log n)$ work w.h.p. per recursive step of size n. By the existing runtime analysis of SampleSort, our algorithm satisfies Theorem 5.8.

5.4 The Parallel-Augmented Sweep Paradigm: Synchronous Strict PIP Scheduling

In this section, we introduce the *parallel-augmented sweep* paradigm, a new technique that combines our in-place parallel algorithms defined above with strictly in-place sequential sweep algorithms to produce efficient, in-place parallel algorithms for all processor counts $p \leq n$. Our technique is a generalization of the *decomposability* property described by Gu, Obeya, and Shun [80].

A problem is decomposable if, given p processors, we can sweep the input in blocks of size p and compute each block independently to form a correct solution. Our technique goes further and handles problems in which these solved subproblems are not independent and must be combined to produce a globally correct solution.

In addition, the algorithms of Gu *et al.* [80] that leveraged this property were Relaxed PIP, as they all required allocating $O(p)$ extra words of shared memory. In contrast, the algorithms we present in this section allocate no extra shared memory, consume $O(1)$ private memory per processor, and elegantly degrade to standard, strictly in-place sequential algorithms when p is set to 1.

5.4.1 Parallel Prefix

As a warm-up, we adapt a well-known algorithm for prefix sums when the number of processors $p < n$ to produce a Synchronous Strict PIP prefix algorithm. The algorithm works like so. Split n into p groups of size n/p, assign each processor to a group, and compute each group's sum iteratively. We then compute a prefix sum on these intermediate sums in parallel using the algorithm from Section 5.2.1. Each processor can then return to its group and infer the prefix values of the remaining $n/p - 1$ elements using the results of the parallel prefix operation.

Theorem 5.12. *There exists an algorithm that performs parallel prefix in $O(n)$ work and $O(n/p + \log p)$ span given $O(p)$ processors for the EREW PRAM that each have $O(1)$ private space.*

When p is 1, this algorithm becomes the trivial sweep algorithm for prefix sums.

Corollary 5.13. *There exists a Synchronous Strict PIP parallel prefix algorithm.*

5.4.2 Packing

Recall that in the packing problem we are given an array of length n containing 1's and 0's. The goal is to move all the 1's to the beginning of the array.

We divide the input into blocks of size p and perform parallel packing on the first block using our algorithm from Section 5.2.2. Let i be the index of the first 0 in this packed block. We use a prefix copy operation to share i with all processors.

In the next step, we perform packing on the second block. Let j be the index of the first 0 after packing the second block. We then merge the two blocks such that the first $2p$ elements of the array are packed. To merge the two blocks, we simply move all the 1's from the second block and place them into the first block, starting at i. Each block is size p, so

we have enough processors to copy all 1's in the second block at once. Any 0's that were overwritten are placed in the original location of the shifted 1's.

This merge step can be done in constant span and $O(p)$ work with exclusive reads and writes since each processor can compute the at most three indices they must read from and write to independently from i, j, the starting index of the current block, and their processor number. After this merge step, we update i to be the location of the first 0 among the merged blocks and then repeat for the next block.

There are n/p blocks, and processing and merging each block takes $O(\log p)$ span and $O(p)$ work, dominated by the cost of performing parallel packing on that block. Thus, the algorithm takes $O(\frac{n}{p} \log p)$ span and $O(n)$ work in total.

Theorem 5.14. *There exists an algorithm that performs parallel packing in $O(n)$ work and $O(\frac{n}{p} \log p)$ span given $O(p)$ processors for the EREW PRAM that each have $O(1)$ private space.*

When p is 1, this algorithm becomes the following standard sequential packing algorithm. We maintain two pointers, i and j, which both begin at the first 0 in the array. At each iteration, advance j forward one index. If j's index now holds a 1, swap its value with i's and advance i forward by one. Otherwise, do nothing.

By the same observation we make in Section 5.2.2, we also have the following.

Corollary 5.15. *There exists an algorithm that performs parallel partition in $O(n)$ work and $O(\frac{n}{p} \log p)$ span given $O(p)$ processors for the EREW PRAM that each have $O(1)$ private space.*

Corollary 5.16. *There exist Synchronous Strict PIP parallel packing and parallel partition algorithms.*

Thus, our algorithm provides a strong memory-usage guarantee for all p and shaves a log-log factor in span over the previous best in-place parallel partition algorithm by Kuszmaul and Westover [101].

5.4.3 Deterministic Reservations

Gu, Obeya, and Shun [80] made the simple observation that random permutation, list contraction, and tree contraction are decomposable. As described above, algorithms with this property are a special case of our parallel-augmented sweep paradigm as the subproblems do not need to be merged after being processed. As a result, we immediately have the following.

Theorem 5.17. *There exist algorithms that perform list and tree contraction in $O(n)$ expected work and $O(\min\{\frac{n}{p}\log p \log n, n\log p\})$ span w.h.p. given $O(p)$ processors for the EREW PRAM that each have $O(1)$ private space. Furthermore, there exists an algorithm that computes a random permutation of n values in $O(n)$ expected work and $O(\min\{\frac{n}{p}\log p \log n, n\log p\})$ span w.h.p. given $O(p)$ processors for the priority-write CRCW PRAM that each have $O(1)$ private space.*

Proof. This follows from a sweep of the input array, applying our in-place algorithms for the respective problems to successive blocks of size p. Recall that the conflict depth of these algorithms is $O(\log n)$ w.h.p. if processing all n entries at once [132]. Then the conflict depth of each p-sized subproblem can be no more than $O(\log n)$ w.h.p. since each processes a subset of all conflicting nodes.

We note that conflict depth is also surely at most $O(p)$, giving us an alternate span bound of $O(\frac{n}{p}\log p \times p) = O(n\log p)$ that holds deterministically. In particular, this implies that the algorithm has $O(n)$ span when $p = O(1)$. □

When p is 1, these problems reduce to their respective standard in-place sweep algorithms (see, e.g., [132]).

Corollary 5.18. *There exist Synchronous Strict PIP algorithms for parallel list contraction, tree contraction, and random permutation*

5.4.4 2D Convex Hull

In this section, we adapt our packing algorithm to show the following.

Theorem 5.19. *There exists an algorithm to compute the convex hull of n presorted points in the plane in $O(\frac{n}{p}\log n)$ span and $O(n\log p + \frac{n}{p}\log n)$ work given $O(p)$ processors for the CREW PRAM that each have $O(1)$ private space.*

Assume the points have been sorted by x-coordinate. We will compute the upper hull (again, we note that the entire hull can be computed if the points are sorted radially). We once again sweep the array from left to right in blocks of size p. We begin by computing the hulls of the first two blocks separately using our in-place convex hull algorithm of Section 5.3.2. Recall that the algorithm permutes the input such that all the hull points are packed to the left of their section of the array in sorted order. To merge these hulls, we then have a single processor compute the tangent points between them using the method of Overmars and van Leeuwen [115]. Say that the tangent points for the first and second hull are stored in indices i and j respectively.

Now we must remove the hull points between the tangent points and then concatenate the remaining points. By convexity and by definition of our algorithm, the surviving points on each hull are contiguous in the array. In addition, since the leftmost point must be on the convex hull, the surviving portion of the first hull is still packed against the left end of the array.

It remains to move the surviving portion of the second hull so that the merged hull is stored contiguously. Since the number of points on the surviving portion of the second hull is no more than p, this final step reduces to the merge procedure of our packing algorithm.

We can compute the convex hull of each block in-place in $O(\log p)$ span and $O(p \log p)$ work. Computing tangents is done sequentially in $O(\log n)$ time. Moving the points in the right hull takes $O(1)$ span and $O(p)$ work. Thus, over all n/p blocks, our algorithm takes $O(\frac{n}{p} \log n)$ span and $O(n \log p + \frac{n}{p} \log n)$ work and so satisfies Theorem 5.19.

When p is 1, this algorithm is an in-place variant of Preparata's online convex hull algorithm [119].

Corollary 5.20. *There exists a Synchronous Strict PIP algorithm to construct a 2D convex hull with presorted points.*

Corollary 5.21. *There exists a Strong PIP algorithm to construct a 2D convex hull of n unsorted points.*

Proof. This follows by combining the above algorithm with a Strong PIP sorting algorithm [80, 89, 101]. □

Bibliography

[1] M. Adler, P. Berenbrink, and K. Schröder. Analyzing an infinite parallel job allocation process. In *European Symposium on Algorithms*, pages 417–428. Springer, 1998.

[2] M. Adler, S. Chakrabarti, M. Mitzenmacher, and L. Rasmussen. Parallel randomized load balancing. In *Proceedings of the twenty-seventh annual ACM symposium on Theory of computing*, pages 238–247, 1995.

[3] P. Afshani, J. Barbay, and T. M. Chan. Instance-optimal geometric algorithms. *Journal of the ACM*, 64(1):3:1–3:38, Mar. 2017.

[4] P. K. Agarwal and M. Sharir. Pseudo-line arrangements: Duality, algorithms, and applications. *SIAM Journal on Computing*, 34(3):526–552, 2005.

[5] A. Aggarwal, B. Chazelle, L. Guibas, C. Ó'Dúnlaing, and C. Yap. Parallel computational geometry. *Algorithmica*, 3:293–327, 1988.

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

[7] M. Ajtai, J. Komlós, and E. Szemerédi. Sorting in $c \log n$ parallel steps. *Combinatorica*, 3(1):1–19, 1983.

[8] J. Allcock, J. Bao, A. Belovs, T. Lee, and M. Santha. On the quantum time complexity of divide and conquer, 2023.

[9] N. M. Amato, M. T. Goodrich, and E. A. Ramos. Parallel algorithms for higher-dimensional convex hulls. In *Proceedings 35th Annual Symposium on Foundations of Computer Science*, pages 683–694. IEEE, 1994.

[10] N. Auger, V. Jugé, C. Nicaud, and C. Pivoteau. On the worst-case complexity of TimSort, 2019.

[11] N. Auger, C. Nicaud, and C. Pivoteau. Merge strategies: from merge sort to TimSort. Research Report hal-01212839, Dec. 2015.

[12] F. Aurenhammer and G. Paulini. On shape Delaunay tessellations. *Inf. Process. Lett.*, 114(10):535–541, 2014.

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

[14] Y. Azar, A. Z. Broder, A. R. Karlin, N. Linial, and S. Phillips. Biased random walks. *Combinatorica*, 16(1):1–18, 1996.

[15] J. Barbay and G. Navarro. On compressing permutations and adaptive sorting. *Theoretical Computer Science*, 513:109–123, 2013.

[16] J. L. Bentley and T. A. Ottmann. Algorithms for reporting and counting geometric intersections. *IEEE Transactions on Computers*, 100(9):643–647, 1979.

[17] P. Berenbrink, A. Czumaj, M. Englert, T. Friedetzky, and L. Nagel. Multiple-choice balanced allocation in (almost) parallel. In *International Workshop on Approximation Algorithms for Combinatorial Optimization*, pages 411–422. Springer, 2012.

[18] P. Berenbrink, T. Friedetzky, P. Kling, F. Mallmann-Trenn, L. Nagel, and C. Wastell. Self-stabilizing balls & bins in batches: The power of leaky bins. In *Proceedings of the 2016 ACM Symposium on Principles of Distributed Computing*, pages 83–92, 2016.

[19] P. Berenbrink, F. Meyer auf der Heide, and K. Schröder. Allocating weighted jobs in parallel. In *Proceedings of the ninth annual ACM symposium on Parallel algorithms and architectures*, pages 302–310, 1997.

[20] A. Biswas, V. Raman, and S. Saurabh. Approximation in (poly-) logarithmic space. *Algorithmica*, 83(7):2303–2331, 2021.

[21] G. E. Blelloch. Prefix sums and their applications. Technical report, School of Computer Science, Carnegie Mellon University Pittsburgh, PA, USA, 1990.

[22] G. E. Blelloch, J. T. Fineman, P. B. Gibbons, and J. Shun. Internally deterministic parallel algorithms can be fast. In *Proceedings of the 17th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*, pages 181–192, 2012.

[23] G. E. Blelloch, J. T. Fineman, Y. Gu, and Y. Sun. Optimal parallel algorithms in the binary-forking model. In *Proceedings of the 32nd ACM Symposium on Parallelism in Algorithms and Architectures*, pages 89–102, 2020.

[24] G. E. Blelloch, J. T. Fineman, and J. Shun. Greedy sequential maximal independent set and matching are parallel on average. In *Proceedings of the Twenty-Fourth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '12, page 308–317, New York, NY, USA, 2012. Association for Computing Machinery.

[25] G. E. Blelloch, P. B. Gibbons, and H. V. Simhadri. Low depth cache-oblivious algorithms. In *Proceedings of the twenty-second annual ACM symposium on Parallelism in algorithms and architectures*, pages 189–199, 2010.

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

[27] G. E. Blelloch, Y. Gu, J. Shun, and Y. Sun. Randomized incremental convex hull is highly parallel. In *Proceedings of the 32nd ACM Symposium on Parallelism in Algorithms and Architectures*, pages 103–115, 2020.

[28] G. E. Blelloch, C. E. Leiserson, B. M. Maggs, C. G. Plaxton, S. J. Smith, and M. Zagha. An experimental analysis of parallel sorting algorithms. *Theory of Computing Systems*, 31(2):135–167, 1998.

[29] H. Blunck and J. Vahrenhold. In-place algorithms for computing (layers of) maxima. *Algorithmica*, 57(1):1–21, 2010.

[30] P. Bose, A. Maheshwari, P. Morin, J. Morrison, M. Smid, and J. Vahrenhold. Space-efficient geometric divide-and-conquer algorithms. *Computational Geometry*, 37(3):209–227, 2007.

[31] R. P. Brent. The parallel evaluation of general arithmetic expressions. *Journal of the ACM (JACM)*, 21(2):201–206, 1974.

[32] H. Brönnimann, T. M. Chan, and E. Y. Chen. Towards in-place geometric algorithms and data structures. In *Proceedings of the twentieth annual symposium on Computational geometry*, pages 239–246, 2004.

[33] H. Brönnimann, J. Iacono, J. Katajainen, P. Morin, J. Morrison, and G. Toussaint. In-place planar convex hull algorithms. In *Latin American Symposium on Theoretical Informatics*, pages 494–507. Springer, 2002.

[34] S. Buss and A. Knop. Strategies for stable merge sorting. In *ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 1272–1290. SIAM, 2019.

[35] S. Chakraborty, A. Mukherjee, V. Raman, and S. R. Satti. A framework for in-place graph algorithms. In *26th Annual European Symposium on Algorithms (ESA 2018)*, pages 13–1. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2018.

[36] T. M. Chan. Optimal output-sensitive convex hull algorithms in two and three dimensions. *Discrete & computational geometry*, 16(4):361–368, 1996.

[37] T. M. Chan. On levels in arrangements of curves. *Discrete & Computational Geometry*, 29:375–393, 2003.

[38] T. M. Chan and E. Y. Chen. In-place 2-d nearest neighbor search. In *Proceedings of the nineteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 904–911, 2008.

[39] E. Y. Chen and T. M. Chan. A space-efficient algorithm for segment intersection. In *CCCG*, pages 68–71, 2003.

[40] J. Chen. Optimizing stable in-place merging. *Theoretical Computer Science*, 302(1):191–210, 2003.

[41] J. Chen. A simple algorithm for in-place merging. *Information Processing Letters*, 98(1):34–40, 2006.

[42] K. L. Clarkson. Applications of random sampling in computational geometry, ii. In *Proceedings of the fourth annual symposium on Computational geometry*, pages 1–11, 1988.

[43] R. Cole. Searching and storing similar lists. *Journal of Algorithms*, 7(2):202–220, 1986.

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

[45] M. E. Dalkilic, E. Acar, and G. Tokatli. A simple shuffle-based stable in-place merge algorithm. *Procedia Computer Science*, 3:1049–1054, 2011.

[46] M. de Berg, M. van Kreveld, M. Overmars, and O. Schwarzkopf. *Computational Geometry: Algorithms and Applications*. Springer, 3rd edition, 2008.

[47] S. de Gouw, J. Rot, F. S. de Boer, R. Bubel, and R. Hähnle. Openjdk's java.utils.collection.sort() is broken: The good, the bad and the worst case. In *Computer Aided Verification*, Cham, 2015. Springer International Publishing.

[48] D. Dereniowski, A. Lukasiewicz, and P. Uznanski. Noisy searching: Simple, fast and correct, 2023.

[49] D. Dereniowski, S. Tiegel, P. Uznanski, and D. Wolleb-Graf. A framework for searching in graphs in the presence of errors. In J. T. Fineman and M. Mitzenmacher, editors, *2nd Symp. on Simplicity in Algorithms (SOSA)*, volume 69, pages 4:1–4:17, 2019.

[50] L. Devroye and B. Reed. On the variance of the height of random binary search trees. *SIAM Journal on Computing*, 24(6):1157–1162, 1995.

[51] M. Dillencourt, M. T. Goodrich, and M. Mitzenmacher. Leveraging parameterized Chernoff bounds for simplified algorithm analyses. *Inf. Process. Lett.*, 187:106516, 2025.

[52] D. Dobkin and R. J. Lipton. Multidimensional searching problems. *SIAM Journal on Computing*, 5(2):181–186, 1976.

[53] R. L. S. Drysdale, III. A practical algorithm for computing the Delaunay triangulation for convex distance functions. In D. S. Johnson, editor, *ACM-SIAM Symp. on Discrete Algorithms (SODA)*, 1990.

[54] S. Edelkamp and A. Weiß. Worst-case efficient sorting with quickmergesort. In *2019 Proceedings of the Meeting on Algorithm Engineering and Experiments (ALENEX)*, pages 1–14, 2019.

[55] S. Edelkamp, A. Weiß, and S. Wild. QuickXsort: A fast sorting scheme in theory and practice. *Algorithmica*, 82(3):509–588, 2020.

[56] H. Edelsbrunner and E. P. Mücke. Simulation of simplicity: a technique to cope with degenerate cases in geometric algorithms. *ACM Trans. Graphics*, 9(1):66–104, 1990.

[57] J. Ellis and M. Markov. In situ, stable merging by way of the perfect shuffle. *The Computer Journal*, 43(1):40–53, 2000.

[58] E. Emamjomeh-Zadeh, D. Kempe, and V. Singhal. Deterministic and probabilistic binary search in graphs. In D. Wichs and Y. Mansour, editors, *48th ACM Symp. on Theory of Computing (STOC)*, pages 519–532, 2016.

[59] D. Eppstein. *Forbidden Configurations in Discrete Geometry*. Cambridge University Press, 2018.

[60] J. Erickson and R. Seidel. Better lower bounds on detecting affine and spherical degeneracies. *Discrete Comput. Geom.*, 13(1):41–57, 1995.

[61] U. Feige, P. Raghavan, D. Peleg, and E. Upfal. Computing with noisy information. *SIAM J. Comput.*, 23(5):1001–1018, 1994.

[62] M. Fernández. *Models of computation: an introduction to computability theory*. Springer Science & Business Media, 2009.

[63] S. Fortune and C. J. Van Wyk. Static analysis yields efficient exact integer arithmetic for computational geometry. *ACM Trans. Graphics*, 15(3):223–248, 1996.

[64] W. D. Frazer and A. C. McKellar. Samplesort: A sampling approach to minimal storage tree sorting. *Journal of the ACM (JACM)*, 17(3):496–507, 1970.

[65] A. Fronczak and P. Fronczak. Biased random walks in complex networks: The role of local navigation rules. *Phys. Rev. E*, 80(1):016107, 2009.

[66] V. Geffert, J. Katajainen, and T. Pasanen. Asymptotically efficient in-place merging. *Theoretical Computer Science*, 237(1):159–181, 2000.

[67] B. Geissmann, S. Leucci, C. Liu, and P. Penna. Optimal sorting with persistent comparison errors. In M. A. Bender, O. Svensson, and G. Herman, editors, *27th European Symp. on Algorithms (ESA)*, volume 144 of *LIPIcs*, pages 49:1–49:14, 2019.

[68] B. Geissmann, S. Leucci, C. Liu, and P. Penna. Optimal dislocation with persistent errors in subquadratic time. *Theory Comput. Syst.*, 64(3):508–521, 2020.

[69] W. C. Gelling, M. E. Nebel, B. Smith, and S. Wild. Multiway powersort. In *Symposium on Algorithm Engineering and Experiments (ALENEX)*. SIAM, 2023.

[70] E. Ghasemi, V. Jugé, G. Khalighinejad, and H. Yazdanyar. Galloping in fast-growth natural merge sorts. *Algorithmica*, 87(2):242–291, 2025.

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

[72] O. Gila. In-place stack-based mergesorts. `https://github.com/ofekih/in-place-stacked-merge`, 2025.

[73] O. Gila, M. T. Goodrich, and V. Sridhar. How to sort in a refrigerator: Simple entropy-sensitive strictly in-place sorting algorithms. In *Latin American Symposium on Theoretical Informatics*. Springer, 2026.

[74] M. T. Goodrich. *Efficient Parallel Techniques for Computational Geometry*. PhD thesis, Purdue University, 1987.

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

[76] M. T. Goodrich and R. Kitagawa. Making quickhull more like quicksort: A simple randomized output-sensitive convex hull algorithm. *arXiv preprint arXiv:2409.19784*, 2024.

[77] M. T. Goodrich and V. Sridhar. Raiders of the lost log: Synchronous parallel in-place models and algorithms. In *2026 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, 2026.

[78] R. L. Graham. An efficient algorithm for determining the convex hull of a finite planar set. *Information processing letters*, 1(4):132–133, 1972.

[79] B. Groz, F. Mallmann-Trenn, C. Mathieu, and V. Verdugo. Skyline computation with noisy comparisons. In L. Gasieniec, R. Klasing, and T. Radzik, editors, *Combinatorial Algorithms*, pages 289–303, Cham, 2020. Springer.

[80] Y. Gu, O. Obeya, and J. Shun. Parallel in-place algorithms: Theory and practice. In *Symposium on algorithmic principles of computer systems (APOCS)*, pages 114–128. SIAM, 2021.

[81] X. Guan and M. A. Langston. Time-space optimal parallel merging and sorting. *IEEE Transactions on Computers*, 40(05):596–602, 1991.

[82] X. Guan and M. A. Langston. Parallel methods for solving fundamental file rearrangement problems. *Journal of Parallel and Distributed Computing*, 14(4):436–439, 1992.

[83] J. Gudmundsson and M. P. Seybold. A tail estimate with exponential decay for the randomized incremental construction of search structures. In J. S. Naor and N. Buchbinder, editors, *ACM-SIAM Symp. on Discrete Algorithms (SODA)*, pages 610–626, 2022.

[84] L. J. Guibas and R. Sedgewick. A dichromatic framework for balanced trees. In *19th IEEE Symp. on Foundations of Computer Science (FOCS)*, pages 8–21, 1978.

[85] N. Gupta and S. Sen. Optimal, output-sensitive algorithms for constructing planar hulls in parallel. *Computational Geometry*, 8(3):151–166, 1997.

[86] K. Hinrichs, J. Nievergelt, and P. Schorn. Plane-sweep solves the closest pair problem elegantly. *Inf. Process. Lett.*, 26(5):255–261, 1988.

[87] B.-C. Huang and M. A. Langston. Practical in-place merging. *Commun. ACM*, 31(3):348–352, Mar. 1988.

[88] B.-C. Huang and M. A. Langston. Fast stable merging and sorting in constant extra space*. *The Computer Journal*, 35(6):643–650, 12 1992.

[89] C. Hutton and A. Melrod. Encoding schemes for parallel in-place algorithms. *arXiv preprint arXiv:2503.06999*, 2025.

[90] K. Iwama, R. Raymond, and S. Yamashita. General bounds for quantum biased oracles. *IPSJ Digital Courier*, 1:415–425, 2005.

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

[92] R. A. Jarvis. On the identification of the convex hull of a finite set of points in the plane. *Information processing letters*, 2(1):18–21, 1973.

[93] V. Jugé. Adaptive shivers sort: An alternative sorting algorithm. *ACM Transaction on Algorithms*, 20(4), Aug. 2024.

[94] F. Kammer and A. Sajenko. Linear-time in-place dfs and bfs on the word ram. In *International Conference on Algorithms and Complexity*, pages 286–298. Springer, 2019.

[95] J. Katajainen, T. Pasanen, and J. Teuhola. Practical in-place mergesort. *Nordic J. of Computing*, 3(1):27–40, Mar. 1996.

[96] J. Keller, C. Keßler, and J. Träff. *Practical PRAM programming*. WileyInterscience, J. Wiley & Sons, Inc., 2001.

[97] K. Khadiev, N. Savelyev, M. Ziatdinov, and D. Melnikov. Noisy tree data structures and quantum applications. *Mathematics*, 11(22), 2023.

[98] D. Kirkpatrick. Optimal search in planar subdivisions. *SIAM J. Comput.*, 12(1):28–35, 1983.

[99] D. G. Kirkpatrick and R. Seidel. The ultimate planar convex hull algorithm? *SIAM journal on computing*, 15(1):287–299, 1986.

[100] R. Klein, R. Penninger, C. Sohler, and D. P. Woodruff. Tolerant algorithms. In C. Demetrescu and M. M. Halldórsson, editors, *19th European Symp. on Algorithms (ESA)*, volume 6942 of *LNCS*, pages 736–747. Springer, 2011.

[101] W. Kuszmaul and A. Westover. Cache-efficient parallel-partition algorithms using exclusive-read-and-write memory. In *Proceedings of the 32nd ACM Symposium on Parallelism in Algorithms and Architectures*, pages 551–553, 2020.

[102] M. A. Langston. Time-space optimal parallel computation. In *Parallel Algorithm Derivation and Program Transformation*, pages 207–223. Springer, 1993.

[103] D. T. Lee and F. P. Preparata. Location of a point in a planar subdivision and its applications. *SIAM J. Comput.*, 6(3):594–606, 1977.

[104] T. Leighton, Y. Ma, and C. G. Plaxton. Breaking the $\Theta(n \log^2 n)$ barrier for sorting with faults. *J. Comput. System Sci.*, 54(2):265–304, 1997.

[105] C. Lenzen, M. Parter, and E. Yogev. Parallel balanced allocations: The heavily loaded case. In *The 31st ACM Symposium on Parallelism in Algorithms and Architectures*, pages 313–322, 2019.

[106] C. Lenzen and R. Wattenhofer. Tight bounds for parallel randomized load balancing. In *Proceedings of the forty-third annual ACM symposium on Theory of computing*, pages 11–20, 2011.

[107] M. Löffler. *Data Imprecision in Computational Geometry*. PhD thesis, Utrecht University, 2009.

[108] M. Löffler and J. Snoeyink. Delaunay triangulation of imprecise points in linear time after preprocessing. *Comput. Geom.*, 43(3):234–242, 2010.

[109] Y. Matias and U. 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.

[110] K. Mehlhorn, R. Osbild, and M. Sagraloff. Reliable and efficient computational geometry via controlled perturbation. In M. Bugliesi, B. Preneel, V. Sassone, and I. Wegener, editors, *Int. Colloq. Automata, Languages and Programming (ICALP)*, volume 4051 of *LNCS*, pages 299–310. Springer, 2006.

[111] K. Mehlhorn and S. Schirra. Exact computation with leda_real – theory and geometric applications. In G. Alefeld, J. Rohn, S. M. Rump, and T. Yamamoto, editors, *Symbolic Algebraic Methods and Verification Methods*, pages 163–172. Springer, 2001.

[112] K. Mulmuley. *Computational Geometry: An Introduction Through Randomized Algorithms*. Prentice Hall, 1994.

[113] J. I. Munro and S. Wild. Nearly-optimal mergesorts: Fast, practical sorting methods that optimally adapt to existing runs. In *26th European Symposium on Algorithms (ESA)*, LIPIcs, pages 63:1–63:16, 2018.

[114] O. Obeya, E. Kahssay, E. Fan, and J. Shun. Theoretically-efficient and practical parallel in-place radix sorting. In *The 31st ACM symposium on parallelism in algorithms and architectures*, pages 213–224, 2019.

[115] M. H. Overmars and J. van Leeuwen. Maintenance of configurations in the plane. *J. Comput. System Sci.*, 23(2):166–204, 1981.

[116] A. Pelc. Searching with known error probability. *Theoret. Comput. Sci.*, 63(2):185–202, 1989.

[117] A. Pelc. Searching games with errors—fifty years of coping with liars. *Theoret. Comput. Sci.*, 270(1-2):71–109, 2002.

[118] T. Peters. Timsort description. retrieved April 2025.

[119] F. P. Preparata. An optimal real-time algorithm for planar convex hulls. *Communications of the ACM*, 22(7):402–405, 1979.

[120] F. P. Preparata and S. J. Hong. Convex hulls of finite sets of points in two and three dimensions. *Communications of the ACM*, 20(2):87–93, 1977.

[121] M. O. Rabin. Probabilistic algorithms. In J. F. Traub, editor, *Symp. New Directions in Algorithms and Complexity*, pages 21–39. Academic Press, 1976.

[122] T. Rauber and G. Rünger. *Parallel programming*. Springer, 2013.

[123] J. H. Reif and S. Sen. Optimal parallel randomized algorithms for three-dimensional convex hulls and related problems. *SIAM Journal on Computing*, 21(3):466–485, 1992.

[124] J. H. Reif and S. Sen. Optimal randomized parallel algorithms for computational geometry. *Algorithmica*, 7(1):91–117, 1992.

[125] A. Rényi. On a problem in information theory. *Magyar Tud. Akad. Mat. Kutató Int. Közl.*, 6:515–516, 1961.

[126] N. Sarnak and R. E. Tarjan. Planar point location using persistent search trees. *Commun. ACM*, 29(7):669–679, 1986.

[127] J. E. Savage. *Models of computation*, volume 136. Addison-Wesley Reading, MA, 1998.

[128] J. K. R. Schou and B. Wang. PersiSort: a new perspective on adaptive sorting based on persistence. In *Canadian Conference on Computational Geometry*, 2024.

[129] S. Sen. A unified approach to tail estimates for randomized incremental construction. In *36th International Symposium on Theoretical Aspects of Computer Science (STACS 2019)*, pages 58–1. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2019.

[130] J. Shun and G. E. Blelloch. Phase-concurrent hash tables for determinism. In *Proceedings of the 26th ACM Symposium on Parallelism in Algorithms and Architectures*, pages 96–107, 2014.

[131] J. Shun, G. E. Blelloch, J. T. Fineman, and P. B. Gibbons. Reducing contention through priority updates. In *Proceedings of the twenty-fifth annual ACM symposium on Parallelism in algorithms and architectures*, pages 152–163, 2013.

[132] J. Shun, Y. Gu, G. E. Blelloch, J. T. Fineman, and P. B. Gibbons. Sequential random permutation, list contraction and tree contraction are highly parallel. In *Proceedings of the twenty-sixth annual ACM-SIAM symposium on Discrete algorithms*, pages 431–448. SIAM, 2014.

[133] S. Skyum. A sweepline algorithm for generalized Delaunay triangulations. *DAIMI Report Series*, 20(373), 1991.

[134] V. Stemann. Parallel balanced allocations. In *Proceedings of the eighth annual ACM symposium on Parallel algorithms and architectures*, pages 261–269, 1996.

[135] T. Takaoka. Partial solution and entropy. In R. Královic and D. Niwinski, editors, *Mathematical Foundations of Computer Science 2009*, pages 700–711. Springer, 2009.

[136] E. Viola. The communication complexity of addition. *Combinatorica*, 35(6):703–747, 2015.

[137] Z. Wang, N. Ghaddar, B. Zhu, and L. Wang. Noisy sorting capacity. *IEEE Transactions on Information Theory*, 70(9):6121–6138, 2024.

[138] J. K. Wong. Some simple in-place merging algorithms. *BIT Numerical Mathematics*, 21:157–166, 1981.

[139] A. C. Yao. Probabilistic computations: Toward a unified measure of complexity. In *18th IEEE Symp. on Foundations of Computer Science (FOCS)*, pages 222–227, 1977.

[140] C.-K. Yap. Symbolic treatment of geometric degeneracies. *J. Symbolic Comput.*, 10(3-4):349–370, 1990.

[141] W. Zhang and N. S. Rao. Optimal parallel quicksort on erew pram. *BIT Numerical Mathematics*, 31(1):69–74, 1991.

[142] Y. Zhang and S. Swanson. A study of application performance with non-volatile main memory. In *31st IEEE Symp. on Mass Storage Systems and Technologies*, 2015.

[143] S. Zheng, B. Calidas, and Y. Zhang. An efficient general in-place parallel sorting scheme. *The Journal of Supercomputing*, 14(1):5–17, 1999.