Randomized Incremental Convex Hull is Highly Parallel

Guy E. Blelloch
Carnegie Mellon University
guyb@cs.cmu.edu

Yan Gu
UC Riverside
ygu@cs.ucr.edu

Julian Shun
MIT CSAIL
jshun@mit.edu

Yihan Sun
UC Riverside
yihans@cs.ucr.edu

ABSTRACT

The randomized incremental convex hull algorithm is one of the most practical and important geometric algorithms in the literature. Due to its simplicity, and the fact that many points or facets can be added independently, it is also widely used in parallel convex hull implementations. However, to date there have been no non-trivial theoretical bounds on the parallelism available in these implementations. In this paper, we provide a strong theoretical analysis showing that the standard incremental algorithm is inherently parallel. In particular, we show that for n points in any constant dimension, the algorithm has $O(\log n)$ dependence depth with high probability. This leads to a simple work-optimal parallel algorithm with polylogarithmic span with high probability.

Our key technical contribution is a new definition and analysis of the configuration dependence graph extending the traditional configuration space, which allows for asynchrony in adding configurations. To capture the "true" dependence between configurations, we define the *support set* of configuration c to be the set of already added configurations that it depends on. We show that for problems where the size of the support set can be bounded by a constant, the depth of the configuration dependence graph is shallow ($O(\log n)$ with high probability for input size n). In addition to convex hull, our approach also extends to several related problems, including half-space intersection and finding the intersection of a set of unit circles. We believe that the configuration dependence graph and its analysis is a general idea that could potentially be applied to more problems.

ACM Reference Format:
Guy E. Blelloch, Yan Gu, Julian Shun, and Yihan Sun. 2020. Randomized Incremental Convex Hull is Highly Parallel. In *Proceedings of the 32nd ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '20), July 15–17, 2020, Virtual Event, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3350755.3400255

1 INTRODUCTION

Finding the convex hull of a set of points in d-dimensions is one of the most fundamental problems in computational geometry. The incremental convex hull algorithm (adding points one by one) is surely the simplest efficient algorithm for the problem, at least for $d > 2$. The basic idea of the (sequential) incremental convex hull algorithm is to add the points one by one while maintaining the convex hull. A newly-added point either falls into the current convex hull and thus no further action is needed, or it removes existing faces (henceforth facets) that it is *visible* from, while adding new facets. For example, in Figure 1, adding c to the existing hull u-v-w-x-y-z-t would replace edges v-w, w-x, x-y, and y-z with v-c and c-z. Clarkson and Shor, in their seminal work over 30 years ago [28], showed that the incremental convex hull algorithm on n points in d-dimensions has optimal $O(n^{\lfloor d/2 \rfloor} + n \log n)$ expected runtime when points are added in random order. Their results are based on a more general setting, which they also used to solve several other geometry problems, and the work led to over a hundred papers and survey articles on the topic of random incremental algorithms. Their proof has been significantly simplified over the years, and is now described in several textbooks [21, 32, 35, 50]. Analysis techniques, such as backwards analysis [54], were developed in this context and are now studied in many intermediate algorithms courses.

In addition to beautiful theory, incremental convex hulls algorithms are also widely used in practice [10, 24, 48, 58] due to their simplicity and efficiency. In the parallel setting, there have been several asymptotically efficient parallel algorithms for convex hull [5, 7, 8, 42, 49, 52], although none of them are based on the incremental approach. Interestingly, although incremental algorithms seem inherently sequential, in practice they are widely used in parallel implementations. The observation is that if two points are visible from disjoint sets of facets, they can be added simultaneously. This idea is used in many parallel implementations of convex hull [27, 34, 38, 40, 42, 47, 56, 59], although with no strong theoretical bounds. Despite the fact that the randomized incremental convex hull algorithm is widely used in practice, it was not previously known whether the incremental approach for convex hull is asymptotically efficient in parallel.

In this paper we show that incremental convex hull is indeed highly parallel, with logarithmic dependence depth. Our result follows recent work on analyzing the the inherent parallelism in sequential incremental algorithms by considering their dependence depth. This work includes showing that incremental algorithms for maximal independent set and maximal matching [14, 36]; graph coloring [43]; correlation clustering [51]; random permutation, list contraction, and tree contraction [55]; and comparison sorting, linear programming, smallest enclosing disk, and closest pairs [17], all have shallow dependence depth. Recent work has also shown that the randomized incremental Delaunay triangulation algorithm in 2D has shallow dependence depth [17, 18]. The key idea is to allow for asynchrony: instead of having an added point build all relevant triangles at once, their analysis allows the triangles added by a point to be added on different rounds. In our results, we borrow the idea of using asynchrony in the convex hull problem. Interestingly, however, the analysis techniques in these previous papers do not apply to incremental convex hull, even just for the 2D case,

SPAA '20, July 15–17, 2020, Virtual Event, USA
© 2020 Association for Computing Machinery.
ACM ISBN 978-1-4503-6935-0/20/07…$15.00
https://doi.org/10.1145/3350755.3400255

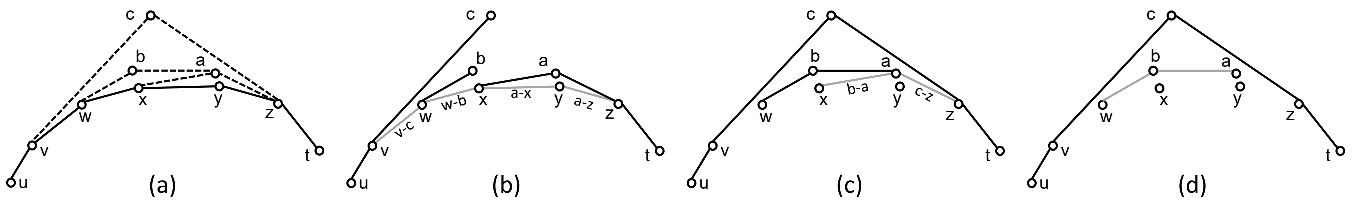


Figure 1: Example of incremental 2D convex hull. Starting with the hull u-v-w-x-y-z-t, we next need to add a, b, and c into the hull in lexicographical order. In our algorithm, the facets v-c, w-b, x-a, and a-z can all be added in parallel in the first step (from (a) to (b)), and b-a and c-z in the second step (from (b) to (c)). The grey edges are those that have already been replaced or buried in the previous step (and thus do not exist). The label on each grey edge indicates the new edge that replaces it, and those without labels are buried. We give a more detailed description of the example in Section 5.3.

although convex hull seems to be easier than many of the above problems.

The challenge for incremental convex hull is that each randomly inserted point may remove (and thus depend on) more than a constant number of existing points and facets (edges for the 2D case) on the hull. As an example of 2D convex hull, in Figure 1 we assume that the hull over the points u-v-w-x-y-z-t has already been generated (marked in dark solid lines) in previous rounds. The points a, b, and c are to be added in lexicographical order. Adding c replaces all edges between v and z, which clearly conflicts with adding a or b, so c has to wait. The number of edges that a newly-added point can remove can be arbitrarily large (more than any constant). In the sequential setting, showing a constant number of dependencies per inserted node *on average* is sufficient to derive the expected work bound since it is the sum of the dependencies for all inserted points. However, it does not work for the analysis of the parallel span (longest dependence path), since the degrees (the number of dependences) of the inserted points multiplicatively contribute to the number of possible paths.

New results and approach of this paper. As with Clarkson and Shor's work [28], and follow-on work by Mulmuley [50] and others, we derive the results based on configuration spaces. Our key technique consists of two aspects, the ***asynchrony*** that enables more fine-grained parallelism, and the concept of a ***support set*** that is used to distinguish the "true" dependences. Using both, we show that the dependence graph of convex hull is shallow, which further leads to a very simple work-efficient and polylogarithmic-span algorithm for d-dimensional convex hull.

The first technique that we employ is to allow asynchrony when adding the facets (edges for the 2D case) incident on one point, as is studied in previous work [17]. In the 2D example in Figure 1(a), we assume that the hull over the points u-v-w-x-y-z-t has already been generated (marked in dark solid lines). The points a, b, and c are to be added in lexicographical order. Adding a point c in previous incremental algorithms would add the edges v-c and c-z, which has to wait for a and b to be added. In our approach, v-c and c-z can be added separately in different rounds. In particular, v-c can be added first since there is no "true" dependence from v-c to edges incident on a or b (see below). This therefore allows for other edges depending on v-c to be processed earlier before c-z is added, which builds a "pipeline" of adding points and improves parallelism. This procedure adds exactly the same facets as the sequential algorithm, and thus compared to the sequential variant, work is not wasted, but rather just reshuffled.

The second technique that we use, which is also the main contribution of this paper, is the new concept of a ***support set*** in addition to the standard requirements of configuration spaces [28, 50]. The goal of the support set is to distinguish the "true" dependences from unnecessary dependences. We observe that when adding a new point p that forms a facet $t = (p, r)$ with an existing ridge r, t only depends on existing facets incident on r. Namely, the new facet t only depends on *two* other facets, t_1 and t_2, which both contain ridge r, and t will *replace* one of them (the one that p conflicts with). In this case, we say the pair of facets t_1 and t_2 *supports* the new facet t (the support set of t is $\{t_1, t_2\}$). Figure 1 shows a running example of our algorithm in two dimensions across three rounds. Starting with the hull u-v-w-x-y-z-t, we next need to add a, b, and c into the hull in lexicographical order. Adding v-c only depends on edges incident on v, which are v-w and v-u, and so we say v-c is supported by v-w and v-u. Similarly, w-b is not supported by x-y or a-z. Hence, v-c, w-b, x-a, and a-z can all be added in parallel since they do not support each other (Figure 1(a) to 1(b)). Then, some existing edges will be replaced by edges with the newly-added points due to visibility, e.g., v-c replaces v-w because c is visible from v-w, but not from v-u (see more details in Section 5.3). From Figure 1(b) to 1(c), b-a replaces x-a and c-z replaces a-z. In addition to replacement, a pair of facets can also be *buried* if it does not support any facet but it is visible by another point. From Figure 1(c) to 1(d), w-b and b-a are buried since they do not support a facet and are both visible from c. Indeed, our approach achieves better parallelism because we distinguish the "true" dependences (support) from the "false" ones (buried). Therefore, we can show that each facet only depends on (is supported by) at most two other facets. More details about the example in Figure 1 are given in Section 5.3.

More generally, we show that for a dependence graph built based on support sets, as long as each object has a constant-size support set, the longest dependence path is logarithmic with high probability. This indicates that the randomized incremental algorithm for convex hull in d-dimensions (and other related problems), when considering the iterations of the algorithm and the dependence structure among the facets, the depth of dependences for n points is only $O(\log n)$ with high probability for a constant dimension d. We formally introduce the relevant concepts about support sets in Section 3 and 4, and our convex hull algorithm in Section 5.

We note that the proposed approach based on support sets in this paper are also useful in several other applications, which we briefly discuss in Section 7.

In summary, the main result of the paper is to prove the following theorem, which states that the depth of dependences of the randomized incremental convex hull construction is shallow.

Theorem 1.1 (Main Theorem). *The configuration dependence graph (defined in Section 4) of incremental convex hull with n points in constant dimensions has depth $O(\log n)$ whp.*

This result directly leads to a very simple d-dimensional parallel incremental convex hull algorithm (Section 5.2), which is work-efficient with polylogarithmic span (see Theorem 5.4 and 5.5). Our algorithm does exactly the same set of plane-side (visibility) tests, and inserts exactly the same facets along the way as that of the sequential algorithm. The only difference is in allowing these tests and updates to run in a more relaxed order, which provides better parallelism. There exists other algorithms for parallel convex hull in the literature that are work-efficient and have better span (e.g., [5, 8]), although they are not based on the incremental approach. We also believe that our algorithm is simpler compared to existing work, and can lead to an efficient parallel implementation in practice.

Preliminaries. We use the work-span model to analyze algorithms, where the ***work*** W is defined to be the number of operations used and the ***span*** S is defined to be the length of the critical path in the computation [45]. Our algorithms can run on the PRAM as well as the nested parallel (fork-join) model. On P processors, the running time is $O(W/P + S)$ [22]. For the CRCW PRAM model, we assume concurrent reads and concurrent writes are supported in unit work. For concurrent writes to the same memory location, we assume an arbitrary write succeeds (arbitrary-CRCW PRAM). Most of the discussion will assume the CRCW PRAM model, but in Appendix A, we also show analysis of our algorithm in the binary-forking model [13], where a computation starts with an initial thread and each thread can call a Fork instruction to create a child task (thread), which is asynchronous and can run in parallel.

We use the term $O(f(n))$ ***with high probability (whp)*** in n to indicate the bound $O(kf(n))$ holds with probability at least $1 - 1/n^k$ for any $k \geq 1$. In asymptotic analysis we assume the dimension d is a constant.

2 RELATED WORK

There has been significant interest in designing parallel algorithms for finding convex hulls over the past decades. Starting in the mid-80s, several PRAM algorithms for 2D convex hull were developed that are optimal in work and have polylogarithmic span [7–9, 49]. Also, at around the same time various algorithms for 3D convex hulls were developed that were within polylogarithmic factors of optimal in work, and had polylogarithmic span [6, 7, 26, 31]. Reif and Sen [52] developed the first work-optimal PRAM algorithm for 3D convex hull, and it also had optimal logarithmic span. This was later generalized to arbitrary constant dimension by Amato, Goodrich, and Ramos [5]. Both of these results are based on ideas from Clarkson and Shor [28]. They are, however, not based on the incremental method but instead based on divide-and-conquer with sampling. The idea is to take a sample of points, build a convex hull on the sample using a more naive approach, and then bucket the remaining points by the facets that are visible from each, and process each bucket in parallel. The algorithms and their analyses, however, are relatively complicated. Gupta and Sen [42] later used the parallel divide-and-conquer approach to develop output-sensitive algorithms for convex hull.

Recent work by Alistarh et al. [3, 4] have used the parallelism guarantees of incremental algorithms to derive strong bounds for the corresponding algorithms using relaxed schedulers.

3 MODEL

We use the definition of configuration space from Mulmuley [32, 50] since we believe it is simpler and more general than Clarkson and Shor's original presentation [28].

A *configuration space* consists of a set of objects X, and a set of configurations $\Pi \subset 2^X \times 2^X$. Each configuration $(D, C) \in \Pi$ consists of a ***defining set*** $D \subseteq X$ and a ***conflict set*** $C \subseteq X \setminus D$. For $\pi \in \Pi$, we use $D(\pi)$ and $C(\pi)$ to indicate its defining and conflicting sets, respectively. For $\Phi \subseteq \Pi$, we use $D(\Phi)$ and $C(\Phi)$ to represent the union of their defining sets and the union of their conflicting sets, respectively.

The ***maximum degree*** of a configuration space is defined as $\max\{|D(\pi)| : \pi \in \Pi\}$, and, as is standard, we assume that the maximum degree is given by a constant g. We also limit the number of configurations with the same defining set by a constant c, which we refer to as the ***multiplicity***.

A configuration π is said to be active with respect to $Y \subseteq X$ if Y includes all of its defining set but none of its conflict set. A set of configurations is active if all configurations in the set are active. The *active configurations* of Y, $T(Y)$, are defined as follows:

$$T(Y) = \{\pi \in \Pi \mid (D(\pi) \subseteq Y) \wedge (C(\pi) \cap Y = \emptyset)\}$$

As an example of a configuration space, X could be a set of points in two dimensions, along with a configuration for each possible triangle (triple of points) in X. A configuration has the three corners of the triangle as its defining set and the points in the triangle's circumcircle as the conflicting set. The active configurations for any subset of points $Y \in X$ is then the Delaunay triangulation of S—i.e., all triangles on Y with no other point from Y in their circumcircle. It has multiplicity one. Table 1 summarizes the notation used in this paper and shows the mapping from configuration spaces to their concrete use for convex hulls (described in Section 5).

Based on the definition of configuration spaces, we have Clarkson and Shor's theorem bounding the expected total number of conflicts in an incremental algorithm that adds one object at a time.

Theorem 3.1 (Total Conflict Size [28]). *Consider a configuration space (X, Π) with maximum degree g. Let $x_1, \ldots, x_n$ be the elements of X (or a subset of them) in a uniformly random order, and $\mathcal{T} = \bigcup_{i=1}^{n} T(\{x_1, \ldots, x_i\})$, then:*

$$E\left[\sum_{\pi \in \mathcal{T}} |C(\pi)|\right] \leq ng^2 \sum_{i=1}^{n} \frac{E[|T(\{x_1, \ldots, x_i\})|]}{i^2}.$$

In Theorem 3.1, $\mathcal{T}$ represents all the configurations that are created, and since for most incremental algorithms the work is proportional to the number of conflicts across the configurations created, the theorem can often be used to bound the total expected work for such algorithms. For Delaunay triangulation, for example, $|T(\{x_1, \ldots, x_i\})| \in O(i)$ so the sum gives $O(n \log n)$, which is indeed the expected work for the incremental algorithm.

Notation	General Definition	For d-dimensional Convex Hull
d	–	The dimension, which we assume is a constant
X	The set of **objects**	The set of input points
Y	A subset of X	A subset of input points
S	An ordered sequence of a set of objects	A permutation of input points
$\Pi \subseteq 2^X \times 2^X$	The set of configurations	All possible facets
$\pi = (C, D)$, $\pi \in \Pi$	A **configuration** π consists of a defining set D and a conflict set C	A facet, i.e., an oriented simplex (hyperplane)
$D(\pi)$, $\pi \in \Pi$	The **defining set** for a configuration π	d points defining the facet π (assuming non-degeneracy)
$C(\pi)$, $\pi \in \Pi$	The **conflict set** for a configuration π	All points visible from π
Φ	A subset of Π. Usually used for a **support set** for (π, x), $\pi \in \Pi$, $x \in X$	Two facets t_1 and t_2 sharing a ridge r, where r is also a ridge of $\pi \backslash x$; see Fact 5.2
$D(\Phi)$, $\Phi \subseteq \Pi$	The union of the defining sets for a set of configurations Φ	All points on all facets in Φ
$C(\Phi)$, $\Phi \subseteq \Pi$	The union of the conflict sets for a set of configurations Φ	All points visible from any facet in Φ
$T(Y)$, $Y \subseteq X$	The **active configuration** set of a set of objects Y	A set of facets where none of their defining points are visible from themselves (e.g., the convex hull of Y)
(X, Π)	A **configuration space**	(points, facets)
g	The **maximum degree** of a configuration space	$g = d$
c	The **multiplicity**	$c = 2$, i.e., facing up and down
n_b	The **base size**	$n_b = d + 1$
k	$\forall$ configuration $\pi \in \Pi$, $\exists$ a support set with size $\leq k$	$k = 2$
$G(S)$	The configuration dependence graph	A graph with dependences among facets based on support sets
$\mathcal{D}(G(S))$	The depth of graph $G(S)$	The depth of graph $G(S)$
t, t', t_1, t_2	–	Used for facets, i.e., oriented d-simplexes
r, r'	–	Used for ridges, i.e., intersection of two facets
p, v_i	–	Used for points

Table 1: The list of notation in this paper.

We now extend the previous definition with some additional definitions needed to bound the depth of dependences.

Definition 3.2 (Support Set). For a configuration space (X, Π), consider a configuration $\pi \in \Pi$ and one of its defining objects $x \in D(\pi)$. We say that $\Phi \subset \Pi$ is a ***support set*** for (π, x) if (1) $D(\pi) \subseteq D(\Phi) \cup \{x\}$, and (2) $C(\pi) \cup \{x\} \subseteq C(\Phi)$.

This definition implies that if a support set Φ for (π, x) is active with respect to any set of objects Y (i.e., $\Phi \subseteq T(Y)$) then by adding x, the configuration π will become active (i.e., $\pi \in T(Y \cup \{x\})$). This is because by (1), adding x will define π, and by (2), nothing conflicts with π that does not conflict with Φ. Furthermore, it implies that adding x will destroy at least one object in Φ since x is in the conflict set of Φ. Importantly, this is all true without having to know about any other active configurations or objects.

Definition 3.3 (k-support). We say a configuration space has ***k-support*** if for all sufficiently large $Y \subseteq X$, and for every configuration $\pi \in T(Y)$ and all of its defining objects $x \in D(\pi)$, there is a support set for (π, x) in $T(Y \setminus \{x\})$ of size at most k. By sufficiently large we mean $|Y| \geq n_b$ for some constant n_b, and we refer to n_b as the ***base size***.

Having small support sets is important since it means that adding a configuration involving a new object depends on only at most k previous configurations. As we will see, this leads to shallow dependence depth. We will assume that for every Y, π, and x, there is exactly one support set of size at most k. If there is more than one, then we can choose one arbitrarily.

4 THE CONFIGURATION DEPENDENCE GRAPH

In this section, we show that the configuration space defined in Section 3 has shallow depth. We are interested in the process of incrementally adding objects in some sequential order and analyzing the dependence graph defined by this ordering. Each step will add some configurations and delete some (those that conflict with the new object). Here, we only care about the configurations added as we are interested in an upper bound on the depth. Let $S = \langle x_1, \ldots, x_n \rangle$ be an ordering of X. For any $Y \subseteq X$, we will use $\min_S(Y)$ and $\max_S(Y)$ to indicate the minimum and maximum element in Y, respectively, based on the ordering S, and we drop the subscript when clear from the context.

Definition 4.1 (Configuration Dependence Graph). For a configuration space (X, Π) with k-support and base size n_b, and a sequence of distinct objects $S = \langle x_1, \ldots, x_n \rangle \in X^*$, let

$$V_i = T(\{x_1, \ldots, x_i\}) \setminus T(\{x_1, \ldots, x_{i-1}\}),$$

i.e., the configurations added on step i. The ***configuration dependence graph*** $G(S)$ for S assigns a vertex to each configuration in $V = \bigcup_{i=1}^{n} V_i$, and edges to each configuration $\pi \in V_i$ for $i > n_b$ from the (up to k) configurations in $T(\{x_1, \ldots, x_{i-1}\})$ that support (π, x_i).

The important property here is that we can add a configuration once its (up to k) configurations from its predecessors in the configuration dependence graph have been added, regardless of what other configurations have been added. We use $\mathcal{D}(G)$ to indicate the depth of the configuration dependence graph G. We are interested in the distribution of dependence graphs considering that all orderings of $x_1, \ldots, x_n$ are equally likely.

Theorem 4.2 (Shallow Dependence). *Consider a configuration space (X, Π) with maximum degree g, multiplicity c, and k-support. For any $Y \subseteq X$, for a random ordering S of Y, and for all $\sigma \geq gke^2$:*

$$\Pr[\mathcal{D}(G(S)) \geq \sigma H_n] < cn^{-(\sigma-g)}$$

where $n = |Y|$, and $H_n = \sum_{i=1}^{n} 1/i$.

Proof. We analyze the depth by considering a single path in $G(S)$ and then take a union bound over the number of possible paths of a given length.

To analyze the single path, we use backwards analysis [54] by considering removing objects one at a time, each selected at random among the remaining objects. We decrease i from n down to n_b. Let Y_i be the set of objects that remain before step i, and $\pi_i \in T(Y_i)$ be a particular active configuration on step i, which we will track as described below and is on the path that we are considering. We start with $i = n$, $Y_n = Y$, and with an arbitrary $\pi_n \in T(Y)$—we account for all such π_n later in the union bound. On step i, we pick a random object x_i from Y_i to remove. If $x_i \in D(\pi_i)$, then it must be that π_i is removed on step i. In this case, we arbitrarily choose one of the (up to k) configurations that (π_i, x_i) has in its support set in $T(Y_{i-1})$, i.e., one of the configurations that it depends on, and make it π_{i-1}. We account for the up to k configurations in the union bounds given below. Picking π_{i-1} extends the dependence path by one. If $x_i \notin D(\pi_i)$, then π_i is not removed and so we keep it by setting $\pi_{i-1} = \pi_i$, and the dependence path is not extended. The probability of the event $x_i \in D(\pi_i)$ is at most g/i, since $|D(\pi_i)| \leq g$ and x_i is chosen at random from i objects. Therefore, on each backwards step i, the dependence path is extended by one with probability at most g/i, and otherwise stays the same.

We analyze a tail bound on the length of the single path. Let X_i be a random variable indicating $x_i \in \pi_i$ (i.e., π_i was removed on step i), and L be a random variable corresponding to the length of the path. We therefore have that $\mathbb{E}[L] \leq \sum_{i=1}^{n} \mathbb{E}[X_i] \leq \sum_{i=1}^{n} \min(1, g/i) \leq g \cdot H_n$. Although the X_i may not be independent[1], the upper bounds (g/i) on the probability of each event is independent of the previous events. This is because we always independently remove a random object from the remaining objects, and g is an upper bound on the degree of any configuration π. Let $\bar{L}$ be a random variable that is a sum of n independent indicator random variables with probabilities $\min(1, g/i)$, $1 \leq i \leq n$ of being 1. We have that $\Pr[L \geq A] \leq \Pr[\bar{L} \geq A]$ since the conditional probabilities of the events that make up L are at most the exact probabilities making up $\bar{L}$. Using a Chernoff bound,[2] for a sum Z of independent indicator random variables we have that:

$$\Pr[Z \geq A] < \left(\frac{eE[Z]}{A}\right)^A$$

which gives,

$$\Pr[L \geq A] \leq \Pr[\bar{L} \geq A] < \left(\frac{egH_n}{A}\right)^A.$$

This gives us the probability that our one path has length at least A.

We now consider the number of such paths and apply a union bound. Recall that c is the multiplicity, which is the number of configurations that can be defined by the same defining set. There are at most cn^g possible configurations in $T(Y)$, since each defining set contains g objects and each defining set can define at most c configurations. For each path ending at any of these configurations, at each configuration we arbitrarily picked one of the at most k configurations that support it. Hence there are at most k^l paths of length l per configuration, for a total of $cn^g k^l$ possible paths of length l. We are interested in the probability that some path is at least σH_n, for $\sigma \geq gke^2$. Using a union bound, we can upper bound this probability by taking the product of the number of possible paths of length $l = \sigma H_n$, and the probability that such a path appears. Note that any longer path must have a length l path as a prefix, so we need not consider the longer paths. This probability is bounded by $\Pr[L \geq A]$ given above with $A = \sigma H_n$. This gives:

$$\begin{aligned}
\Pr[\mathcal{D}(G) > \sigma H_n] &\leq cn^g k^{\sigma H_n} \cdot \Pr[L \geq \sigma H_n] \\
&< cn^g k^{\sigma H_n} \left(\frac{egH_n}{\sigma H_n}\right)^{\sigma H_n} \\
&= cn^g \left(\frac{kge}{\sigma}\right)^{\sigma H_n} \leq cn^g \left(\frac{1}{e}\right)^{(\ln n)\sigma} \\
&= cn^{-(\sigma-g)}.
\end{aligned}$$

□

Algorithm 1: Generic Parallel Incremental Algorithm

Input: An set of objects $X = \{x_1, \ldots, x_n\}$ with an ordering S on it.
Output: $T(X)$, the active configuration of all objects in X.
Maintains: $\mathcal{T}$ = the current set of configurations.

```
1  function ParallelIncremental($X = \{x_1, \ldots, x_n\}$, $S$)
2      $\mathcal{T} \leftarrow T(\{x_1, \ldots, x_{n_b}\})$
3      $\Psi \leftarrow \{\Phi \subseteq \mathcal{T} \mid \Phi$ is support sets for any $(\pi, x)$, $|\Phi| \leq k\}$
4      parallel foreach $\Phi \in \Psi$ do AddConfiguration($\Phi$, $S$)
5      return $\mathcal{T}$
6  function AddConfiguration($\Phi$, $S$)
7      $x \leftarrow \min_S(C(\Phi))$
8      if $\Phi$ supports $(\pi, x)$ for some $\pi$ then
9          $C(\pi) \leftarrow \{x' \in C(\Phi) \mid$ conflicts$(x', \pi)\}$
10         $\mathcal{T} \leftarrow (\mathcal{T} \cup \{\pi\}) \setminus \{\pi' \in \mathcal{T} \mid x \in C(\pi')\}$
11         $\Psi \leftarrow \{\Phi' \subseteq \mathcal{T} \mid \Phi'$ is support set for any $(\pi', x)$,
               $\pi \in \Phi'$, $|\Phi'| \leq k\}$
12         parallel foreach $\Phi' \in \Psi$ do
13             AddConfiguration($\Phi'$, $S$)
```

This leads to Algorithm 1 for executing a parallel incremental algorithm based on its configuration space. As given, this algorithm is under-specified since it does not describe how to find the support sets and what they support, but we will see a concrete implementation for convex hull in the next section.

Theorem 4.3. *For a dependence graph G with depth $\mathcal{D}(G)$, the maximum recursion depth of Algorithm 1 is $\mathcal{D}(G)$.*

Proof. We only make recursive calls on Line 13 when we have enabled a configuration S, meaning that all of its predecessors in the configuration dependence graph have already been executed. Therefore, a recursive call corresponds to descending one level in

[1]They would be independent if all configurations had degree exactly g.
[2]Not standard form, but easily derivable.

the dependence graph. The depth of the dependence graph is $\mathcal{D}(G)$, and thus the theorem follows. □

The algorithm is presented in a nested-parallel style but can be executed on a PRAM by processing the configuration dependence graph level-by-level, each level in parallel.

Relationship to History Graphs. We note that the configuration dependence graph is similar to a history [50] or an influence graph [19, 20] and used as a search structure to find conflicting configurations for an object. For example, for trapezoidal decomposition, an influence graph can be used for locating a point in its corresponding trapezoid. Mulmuley proved that such a search path has $O(\log n)$ length *whp* [50, Lemma 3.1.5]. However, an important point is that this does not by itself imply that the configuration dependence graph has $O(\log n)$ depth *whp* since there can be paths in the configuration dependence graph that do not correspond to any search path for an object. Mulmuley's proof relies on every configuration depending on a single previous configuration, which is true for a search path.

Indeed, our results do not show that the configuration dependence graphs for trapezoidal decompositions have logarithmic depth since the configuration spaces for trapezoidal decompositions do not have constant support. Adding a line segment can combine $\Omega(n)$ trapezoids into one. The new trapezoid depends on all those trapezoids even though a search for a particular point depends on only a single one. There will almost certainly be paths in the configuration dependence graph (or influence graph) that do not correspond to a search for any point. Our support condition states that the longest path of any type in the configuration dependence graph is logarithmic, *whp*, as long as the configuration space has constant support.

5 CONVEX HULLS

We are concerned with finding the convex hull of a set of points P in $\mathbb{R}^d$. In this section, we assume that the points are in general position (no degeneracies)—i.e., at most d points lie in a $d-1$ hyperplane. We relax this requirement in Section 6.

5.1 Constant Support for Convex Hull

The convex hull is a convex d-dimensional polytope (the smallest one enclosing the points). We refer to the $d-1$ dimensional faces of the polytope as ***facets***, and the $d-2$ dimensional interfaces between the faces as ***ridges***. For points in general position, each facet is an oriented d-simplex (defined by d points) and each ridge is a $(d-1)$-simplex (defined by $d-1$ points). The facets have an up (facing out) or down (facing in) orientation, each defining a half-space in d dimensions, with a $(d-1)$-dimensional hyperplane defined by its points. A point is ***visible*** from a facet if it is in the open half-space defined by the facet. A ridge is incident on exactly two facets of the same orientation. A facet is incident on d ridges.

To model d-dimensional convex hulls in general position as a configuration space, the objects X are the set of input points and each configuration corresponds to a possible facet. In particular, every subset of d points defines two facets, one oriented up and one oriented down. The configuration corresponding to a facet will conflict with points visible from that facet. The two configurations will therefore have complementary conflict sets (except for the defining points, which are in neither conflict set). The configuration space has maximum degree d and multiplicity 2.

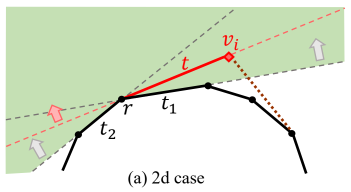
Algorithm 2: Sequential Incremental Convex Hull

```
Input: A sequence $V = \{v_1, \ldots, v_n\}$ of points in $\mathbb{R}^d$.
Output: The convex hull of $V$.
Maintains:
    $H$ = the current set of facets.
    $C$ = a map from facets to conflicting (visible) points.
    $C^{-1}$ = the inverse map of $C$.
function ConvexHull($V = \{v_1, \ldots, v_n\}$)
    $H \leftarrow$ the convex hull of $\{v_1, \ldots, v_{d+1}\}$
    foreach facet $t \in H$ do $C(t) \leftarrow \{v \in V \mid \text{visible}(v, t)\}$
    for $i \leftarrow d+2$ to $n$ do
        Let $R \leftarrow C^{-1}(v_i)$
        foreach ridge $r$ on the boundary of $R$ do
            $(t_1, t_2) \leftarrow$ the two facets incident on $r$, with $t_1$
                    visible from $v_i$ and $t_2$ invisible
            $t \leftarrow$ a new facet consisting of $r$ and $v_i$
            $C(t) \leftarrow \{v \in C(t_1) \cup C(t_2) \mid \text{visible}(v, t)\}$
            $H = H \cup \{t\}$
        $H = H \setminus R$
    return $H$
```

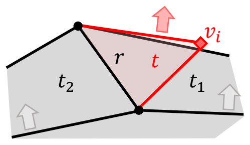
When adding a point p to a convex hull (or the corresponding configuration space), we remove all facets visible from p. These facets are contiguous (reachable by shared ridges) and the region that is removed leaves a simply connected boundary of ridges. The new convex hull with p is then the old hull with the visible facets removed, and a new facet from each ridge to p. This leads to the standard incremental algorithm for convex hulls given by Algorithm 2, which requires $O(n^{\lfloor d/2 \rfloor} + n \log n)$ visibility tests in expectation, and $O(n^{\lfloor d/2 \rfloor} + n \log n)$ work when using appropriate data structures [32, 50, 53].

Theorem 5.1 (Convex Hull Support). *The configuration space of convex hulls in d dimensions has 2-support with base size $d+1$, and with support sets that always consist of two facets sharing a ridge.*

Proof. Consider a set of points V_i with $|V_i| \geq d+1$, and its active configuration $T(V_i)$ (i.e., its convex hull). We argue that every configuration (simplex) $t \in T(V_i)$, and every point $v \in t$ has 2-support. Consider the ridge $r = t \setminus \{v_i\}$, which must be present in $T(V_i \setminus \{v_i\})$ since adding v_i extends each boundary (ridge) of the deleted region with v_i. We claim that the two adjacent facets, t_1 and t_2, of r in $T(V_i \setminus \{v_i\})$ form a support set for (t, v_i). In particular, no point in V_i can conflict with t (i.e., $C(t) \cap V_i = \emptyset$), because if a point is visible from t it must be visible from either t_1 or t_2. See Figure 2 for an illustration. In the 2D case (Figure 2(a)), based on the geometric relationship, since v_i is visible from either t_1 or t_2, the visible set of t consisting of v_i and r is a subset of $C(t_1) \cup C(t_2)$, indicating the 2-support. In higher dimensions, the ridge r has $d-2$ dimensions. When projecting the simplex to the 2 dimensions that r is not defined on (Figure 2(b)), the proof for 2-support is the same as in the 2D case. □



(a) 2d case



(b) 3d case

Figure 2: Illustration of 2-support for convex hulls. (a) The 2D case from Figure 1. The facet t consisting of r and v_i is supported by t_1 and t_2, since it is easy to check that $C(t) \cup \{v_i\} \subseteq C(\{t_1, t_2\})$ as $C(t)$ is above the red long dashed line, and $C(\{t_1, t_1\})$ is the green shaded region. Also, clearly $D(t) \subseteq D(\{t_1, t_2\}) \cup \{v_i\}$. Similarly, if we consider removing any point from a hull, both removed facets (edges) will have 2-support from facets in the remaining hull (i.e., the two facets sharing the opposite endpoint). (b) A 3D case where ridges are segments and facets are triangles. When rotating the viewpoint so that ridge r is perpendicular to this sheet of paper, the geometric relationship of the conflict sets $C(t)$, $C(t_1)$, and $C(t_2)$ is exactly the same as in the 2D case shown in (a). Similarly, the projection of a d-dimensional case onto the 2 dimensions that the ridge is not defined on is equivalent to the 2D case, and so the configuration space always has 2-support.

Based on Theorem 5.1, we give the following fact for the support set for the configuration space of convex hulls.

FACT 5.2. *$\{t_1, t_2\} \subseteq \Pi$ is the support set for (t, x) if and only if (1) t_1 and t_2 share a ridge r, and r and x define t; and (2) x is visible from either t_1 or t_2, and is not visible from the other, with t oriented away from the facet (one of t_1 and t_2) that x is visible from.*

Based on Theorem 5.1 and plugging in $g = d$ into Theorem 4.2, we obtain our main result, Theorem 1.1.

5.2 A Parallel Randomized Incremental Convex Hull Algorithm

We now consider a simple parallel variant of the sequential algorithm. It creates the exact same set of facets along the way and runs the exact same set of visibility tests, but in a relaxed order defined by the configuration dependence graph. We first define the ***conflict pivot*** of a facet t as $b_t = \min_S(C(t))$, i.e., the earliest inserted point in its conflict set (visible from the facet). In fact, the facet t should be removed if the point b_t is being inserted.

To create facets in the order defined by the graph, we need to recognize the facet supported by a support set as soon as the support set is created. Since support sets consist of adjacent facets sharing a ridge, we can identify each potential support set when creating a ridge, which happens when adding a facet. Since two facets define a ridge, the ridge is not ready until the second facet is added.

This simple idea leads to Algorithm 3. As with the sequential algorithm, it first assigns conflict sets for a convex hull on $d+1$ points. Then, for each pair of adjacent facets and the ridge between them, it makes calls to the recursive function ProcessRidge(t_1, r, t_2). The function determines if the two facets t_1 and t_2 sharing a ridge r support a new facet incident on r, and if so, creates the facet and recurses on all newly created ridges. This function is only called when both facets of the ridge r have been created. In the parallel setting, the two facet sharing a ridge r may be created at different times, and thus the facet that is created later is responsible for

Algorithm 3: Parallel Incremental Convex Hull

```
Input: A sequence $V = \{v_1, \ldots, v_n\}$ of points in $\mathbb{R}^d$.
Output: The facets on the convex hull of $V$.
Maintains:
    $H$ = the current set of facets.
    $C$ = a map from facets to conflicting (visible) points.
    $M$ = a map from ridges to the incident facets.
1  function ConvexHull($V = \{v_1, \ldots, v_n\}$)
2      $H \leftarrow$ the convex hull of $\{v_1, \ldots, v_{d+1}\}$
3      parallel foreach $t \in H$ do
4          $C(t) \leftarrow \{v \in V \mid \text{visible}(v, t)\}$
5      parallel foreach $\{t_1, t_2\} \subseteq H$ sharing ridge $r$ do
6          ProcessRidge($t_1, r, t_2$)
7      return $H$
8  function ProcessRidge($t_1, r, t_2$)
9      if $C(t_2) = C(t_1) = \emptyset$ then return
10     else if $\min(C(t_2)) = \min(C(t_1))$ then $H \leftarrow H \setminus \{t_1, t_2\}$
11     else if $\min(C(t_2)) < \min(C(t_1))$ then
12         ProcessRidge($t_2, r, t_1$)
13     else
14         $p \leftarrow \min(C(t_1))$
15         $t \leftarrow$ join $r$ with $p$
16         $C(t) \leftarrow \{v' \in C(t_1) \cup C(t_2) \mid \text{visible}(v', t)\}$
17         $H \leftarrow (H \setminus \{t_1\}) \cup \{t\}$
18         parallel foreach $r' \in$ boundary of $t$ do
19             if $r = r'$ then ProcessRidge($t, r, t_2$)
20             else if ($\neg M$.InsertAndSet($r', t$)) then
21                 $t' = M$.GetValue($r', t$)
22                 ProcessRidge($t, r', t'$)
```

calling ProcessRidge on r. Note that exactly two facets define one ridge. The interaction between the two facets sharing a ridge r, although created different times, can be handled by a hash table keyed by the ridges — the first facet that arrives creates the entry in the hash table and leaves its information, and the second one reads the information about the first facet and invokes ProcessRidge on r. If they arrive simultaneously the tie can be broken either way. More details are provided below.

This algorithm starts with an initial convex hull of $n_b = d + 1$ points. It then calls ProcessRidge on each possible ridge, which, along with the two facets incident on it, forms a possible support set for some (x, π). Consider the four cases of the conditional statement starting on Line 9. In the first case (Line 9), the two facets have no conflicts, and so there is no facet to support. Therefore, there are no points outside of these two facets, and no further action is needed.

In the second case, both facets have the same conflict pivot p' (Line 10). From Fact 5.2, $\{t_1, t_2\}$ is not a support set for a new facet with a point p' since p' is visible from both facets. Thus the point p', which will expand the current convex hull by forming new facets with some other ridges "surrounding" r, will remove the ridge r. We say that p' ***buries*** the ridge r in this case. This case also needs to delete both facets from the convex hull since they will be "covered" by other facets. However, we do not need to further deal with p' since p' will be processed by its support set.

In the last two cases (which are symmetric), the earliest (in insertion order) point is visible from one facet but not the other, and thus from Fact 5.2, $\{t_1, t_2\}$ supports a facet. WLOG, we assume that the conflict pivot of t_1 is earlier than that of t_2. Otherwise, we flip the order and call the function again (Line 12).

When the earliest point p is visible from just t_1 (Line 14), the new facet t will consist of the ridge r between t_1 and t_2 and the conflict pivot p. Based on Fact 5.2 (also as described in the proof of Theorem 5.1), (t, p) is supported by $\{t_1, t_2\}$. Since p is the earliest point supported by $\{t_1, t_2\}$, it can be processed immediately. The algorithm then removes t_1 from the current hull, and adds a new facet $t = (p, r)$. We say that t ***replaces*** t_1 in this case. After that, all new ridges on the new facet, if ready, need to be processed on the next round. By "ready", we mean that the facets on both sides of this ridge have been added to the hull.

Since facets are added asynchronously, for a ridge r, the facet of the two sides arriving later is responsible for processing the ridge. In particular, the ridge between t and t_2 is always ready since t_2 already exists (Line 19). For the other ridges, we store for each ridge, the first facet incident on it added by the algorithm. The second one that is added can therefore find the information of the first facet and recurse on the pair. This can be implemented using a global mapping M. In the pseudocode, the function M.InsertAndSet(r', t) checks if the ridge r' has already been mapped to some value in M, and if so, returns False. If r' has not yet been mapped to any facet, it sets the value of r' to t and returns True. When running in parallel, InsertAndSet has to be atomic. Note that there can be at most two values (facets) associated with one key (ridge) being inserted into the map. Accordingly, function M.GetValue(r', t) returns the value associated with r' which is not t. When InsertAndSet fails, this function call is then responsible for processing r' with two facets: the new facet t created by this call, and a facet t' associated with r' by previous calls.

On a CRCW PRAM, the mapping M can be implemented by a dictionary [39], such that all ridges to be processed in the next round can be inserted into M in $O(\log^* n)$ span *whp*. This requires the algorithm to run in rounds and use synchronization between rounds. For loosely-synchronized models such the binary-forking model, the mapping M and the functions InsertAndSet and GetValue can be implemented using a parallel hash table with compare-and-swap. We present the algorithm later in this subsection. We note that M can also be implemented with an even weaker atomic primitive TestAndSet. We present the algorithm in Appendix A. Using either CompareAndSwap or TestAndSet, InsertAndSet and GetValue can be implemented in $O(\log n)$ span *whp*.

The algorithm calls ProcessRidge exactly once for every triple t_1, r, t_2, where r is a ridge and t_1 and t_2 are the two facets defining r. This is because each ridge is defined by exactly two facets, and the second facet to be added will always make the call. After that, the ridge will either be finalized (Line 9) or disappear because at least one of the facet is buried (Line 10) or replaced (from Line 17). Furthermore, ProcessRidge never blocks—if it is the first to arrive on a ridge, it returns and lets the other facet handle the ridge.

Theorem 5.3 (Recursion Depth). *Algorithm 3 has recursion depth $O(\log n)$* whp.

Proof. Apply Theorem 4.3 with $\mathcal{D}(G) = O(\log n)$ *whp*. □

We note that assuming a constant dimension, the base case cost ($n_b = d + 1$ is also a constant) and the visibility check also only take constant work. With the recursion depth bounded by $O(\log n)$ *whp*, we can show that the algorithm is work-efficient with polylogarithmic span in various parallel models.

Theorem 5.4 (Cost of Convex Hull Algorithm). *Algorithm 3 runs in $O(n^{\lfloor d/2 \rfloor} + n \log n)$ expected work and $O(\log n \log^* n)$ span* whp *on a CRCW PRAM.*

Proof. The visibility tests performed in the parallel algorithm all correspond to a visibility test that would have been performed by the sequential algorithm. Therefore, the expected number of visibility tests is $O(n^{\lfloor d/2 \rfloor} + n \log n)$ and can be implemented in the same expected work (note that some sequential visibility tests are skipped in the parallel case due to buried ridges). We can maintain the data structures H, M, and C using parallel hash tables which support insertions, deletions, and finds in linear work and $O(\log^* n)$ span *whp* [39]. All other work can be charged to visibility tests that would have occurred in the sequential algorithm. Therefore, the total expected work is $O(n^{\lfloor d/2 \rfloor} + n \log n)$.

For the span, the number of levels of recursion is $O(\log n)$ *whp* by Theorem 4.3. We run the algorithm in rounds and fully synchronize between rounds. We now argue that each round (each call to a ProcessRidge ignoring recursion) can be implemented in $O(\log^* n)$ span *whp*. As mentioned above, updating the hash tables takes $O(\log^* n)$ span *whp*. Finding the minimum of a set (Lines 10, 12, and 14) takes $O(1)$ span *whp* [60]. Load balancing, and filtering the points that are visible (Line 16) can be done with approximate compaction in $O(\log^* n)$ span [41]. Therefore, including recursive calls, the overall span is $O(\log n \log^* n)$ *whp*. The initial convex hull (Line 2) can be found sequentially in constant work. □

This algorithm can be easily applied to other parallel models, such as models based on fork-join parallelism, and thus requiring no synchronization between rounds. In this case, to support InsertAndSet, some atomic primitive, such as TestAndSet or CompareAndSwap, is needed. In the following discussion, we consider the binary-forking model [13]. In this setting, tasks can fork one child task and continue its own computation in parallel. Tasks can be forked recursively and executed asynchronously. Furthermore, the atomic TestAndSet instruction is allowed as a primitive for threads to reach consensus. This model is a fundamental model for parallelism and has been widely used in analyzing parallel algorithms [1, 2, 11, 12, 15, 16, 29, 30, 33], and are also supported by programming systems such as Cilk [37], the Java fork-join framework [46], X10 [25], Habanero [23], TBB [44], and TPL [57]. In our algorithm, when the more powerful primitive CompareAndSwap is supported, we have a simpler implementation for InsertAndSet. For simplicity, we briefly show the algorithm using CompareAndSwap later in this subsection, and leave the full algorithm using the weaker TestAndSet in Appendix A. In both settings, InsertAndSet and GetValue take $O(\log n)$ work and span *whp*. In addition, in each function call to ProcessRidge, the span for combining the conflict sets and finding the minimum of the set is $O(\log n)$. In total, the algorithm has optimal expected work and $O(\log^2 n)$ span *whp*, which leads to the following theorem.

Theorem 5.5 (Cost of Convex Hull Algorithm). *Algorithm 3 runs in $O(n^{\lfloor d/2 \rfloor} + n \log n)$ expected work and $O(\log^2 n)$ span* whp *in the binary-forking model.*

Space Complexity. To maintain the hash tables in Algorithm 3, the space needed can be proportional to the work. We note that this is also the worst-case space usage for storing all output facets.

***Implementing* InsertAndSet *using* CompareAndSwap.** Here we present a simple implementation of InsertAndSet using the atomic primitive CompareAndSwap. Recall that InsertAndSet(r, t) checks in M if ridge r has already been added to it. If so, it returns False; otherwise, it inserts r to the hash table with the value t and returns True. GetValue(r, t) looks up the value associated with r in the map M, and returns t', the facet on the other side of the ridge. We note that in Algorithm 3, there can be at most two values associated with the same key (ridge) in M. We have to guarantee that, when calling GetValue(r, t), the value $t' \neq t$ associated with r has already been inserted into M.

The algorithm is shown in Algorithm 4. In this algorithm, the mapping is maintained by a hash table R with linear probing. When adding a key-value pair (r, t) to the hash table, we first find the index of it using the hash function f_R. The algorithm then tries to CompareAndSwap in the pointer of the key-value pair (r, t) on Line 3. There are two cases that may cause the CompareAndSwap to fail: a collision on the index due to the hash function, or a conflict since two facets have the same key. We then check if it is due to a duplicate key (Line 4). If so, the current facet t is the second value associated with r, and so we simply return False. If CompareAndSwap fails due to a collision, we do linear probing until an empty slot is found, and write the pair (r, t) to this slot. This means that t is the first facet incident on r being added, and so the algorithm return True.

For GetValue, we simply look up the value of r in R as in a standard hash table. Since the task calling GetValue must have failed in InsertAndSet(r', t) on Line 20 in Algorithm 3, the value of the other facet for r' must have already been added to the map then. Therefore, GetValue is guaranteed to find the facet $t' \neq t$ associated with r' in M.

The work and span for GetValue and InsertAndSet is just the cost for linear probing, which is $O(\log n)$ *whp.*

5.3 Example of the Parallel Algorithm

We use Figure 1 as an example to show how our parallel algorithm works in 2D. Starting from a hull u-v-w-x-y-z-t, suppose we add points a, b and c to the hull in lexicographical order. The algorithm will call ProcessRidge for the current support sets, which are all "corners" consisting of two edges incident on a point. Following the algorithm, when processing a corner, each of the two edges first finds its conflict pivot. In particular, w-v sees c as its conflict pivot, but v-u does not. This falls into case 4 of the algorithm (starting from Line 14), where v-c is added to the hull, and v-w is replaced by v-c. Similarly, w-x is replaced by w-b, x-y is replaced by x-a, and y-z is replaced by a-z. All these replacements can be processed in parallel. These edges being replaced are colored in grey in Figure 1(b), and are labeled by the edges replacing them. Proceeding to the next round, each newly-added corner will be processed.

Algorithm 4: InsertAndSet and GetValue on a multimap M.

```
InsertAndSet(r, t)
  Input: A ridge r and a facet t.
  Maintains:
      A map R as a hash table from keys as ridges and values as
      pointers to ridge-facet pairs. R uses linear probing for
      conflicts. Assume the hash function for R is $f_R$, which
      hashes a ridge r to an index in the range $[0, \dots, |R| - 1]$.
  Output: If $r \in M$, return false. Otherwise return true and map
          key r to value t in M.
1  function M.InsertAndSet(r, t)
2      $i \leftarrow f_R(r)$                  // get the starting index
3      while $\neg$CompareAndSwap(R[i], Null, (r, t)) do
4          if the key stored at R[i] is r then
5              return False
6          $i \leftarrow (i + 1)$ Mod M.size
7      return True
GetValue(r, t)
  Input: A ridge r and a facet t.
  Output: A value t' associated with r in M which is not t.
8  function M.GetValue(r, t)
9      $i \leftarrow f_R(r)$                  // get the starting index
10     while R[i].key $\neq$ r do $i \leftarrow (i + 1)$ Mod M.size
11     return the value in R[i]
```

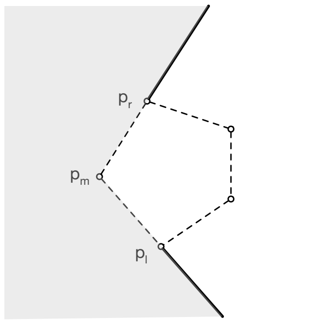


Figure 3: Example of the conflict set for the corner configuration $\{p_m, p_l, p_r\}$ with corner point p_m. The figure represents the plane of the facet (a pentagon) containing the corner. The shaded region and the dark lines conflict with the configuration, but the dashed lines and white space do not.

On this round (Figure 1(b) to 1(c)), the corners at c and b cannot be processed since the other side of the corner is not ready. The corner x-a-z can be processed since both of its edges have been added, and the conflict pivot is b (b-a is added). Similarly, the corner a-z-t can be processed since both of its edges have been added, and the conflict pivot is c (c-z is added). As a result, x-a and a-z are replaced by b-a and c-z, respectively, and in parallel. On the next round (Figure 1(c) to 1(d)), the corners w-b-a and v-c-z are ready. For w-b-a, both of the edges w-b and b-a see c as their conflict pivot, and so it falls into case 2 of the algorithm (Line 10), which directly buries w-b and b-a from the convex hull and returns. For v-c-z, the conflict set of the two edges are both empty, which is case 1 of the algorithm (Line 9). This means that these edges are finalized and the algorithm returns.

6 CONVEX HULLS WITH DEGENERACY

Here we describe how to use configuration spaces with constant support for 3D convex hulls with degenerate points, i.e., four or more lie on a plane or three or more lie on a line. The idea that we use is based on the description by Berg et. al. [32]. As before, the set of objects X is a set of points. When four or more points are degenerate, the facets need not be triangles and instead can be arbitrary convex polygons. We therefore cannot use the points on a facet to define configurations since in general they do not have constant maximum degree. Instead, we define the configurations in terms of the corners of possible facets on the convex hull. In particular, Π consists of six configurations for each non-collinear triple of points in X. The six configurations correspond to each one of the three points being the "corner", and for each such corner, one for each side of the plane defined by the three points. As with the non-degenerate case, a configuration conflicts with all points above its plane for the appropriate side. If we label the corner point as p_m and the other two points as p_l and p_r, the configuration also conflicts with all points on the plane defined by the three points that are visible from the outside (strict) of either of the two lines p_m-p_l or p_m-p_r, as well as points on the lines starting from p_r or p_l in the direction away from p_m. Figure 3 illustrates an example. We call this the *corner configuration space.* We note that for points that are collinear along a facet edge, only the outermost two along the line define the hull and are part of any corner. Similarly, points on a facet but not on its boundary do not define the convex hull.

Lemma 6.1. *For a corner configuration space* (X, Π) *and* $Y \subset X$, $T(Y)$ *includes one configuration for each corner of the 3D convex hull of* Y.

Proof. We will show that a configuration is in $T(Y)$ if and only if it is a corner of the convex hull of Y. Consider a corner $\pi = p_r$-p_m-p_l with $p_r, p_m, p_l \in Y$. No point above the plane defined by π is in Y, and also no point will be on the facet plane and outside either of the two lines p_m-p_l or p_m-p_r (e.g., the shaded region in Figure 3), or on those lines starting from p_r or p_l in the direction away from p_m (e.g., the solid black half-line starting from p_r or p_l in Figure 3). Therefore, if π is a corner, then $\pi \in T(Y)$ (i.e., it has no conflicts with Y).

For the "only if" direction, a configuration $\pi = p_r$-p_m-p_l that is not a corner must conflict either with a point above its plane (if not all on the same facet), a point outside p_m-p_l or p_m-p_r (if not adjacent on the facet), or a point on those lines but outside p_r-p_l (if not consisting of the outermost collinear points on the edge of the facet). Hence, since all $\pi' \in T(Y)$ have no conflicts with Y, they must all be corners. □

We note that the number of configurations for a set of points in the corner configuration space is at most three times the number in the non-degenerate triangle configuration space. This is because the worst case is when each facet is a triangle, and there are then three corners per triangle. If any face is degenerate, then this decreases the number of corners [32]. This means that the asymptotic work of incremental convex hull is unchanged. However, we need to bound the support to bound the depth of the dependence graph.

Lemma 6.2. *The corner configuration space* (X, Π) *has* 4*-support.*

Proof. Consider a configuration π and a point $x \in D(\pi)$. First, consider x as the corner point and call the other two points y and z. Adding x will remove a ridge incident on each of y and z. This is possibly, but not necessarily, a ridge joining them. Consider the ridge y-y' from y (as mentioned it is possible that $y' = z$, although this is not necessary). This ridge y-y' will be involved in two corners with y as the corner point, each corresponding to one of the two facets incident on y-y'. Similarly, we can find two such corners for z. We will then show that (π, x) is supported by those at most four corners.

One case is that x, y, and z define a different plane than those defined by any of the four corners (the non-degenerate case). In this case, the ridge incident on y and z is the same ridge, and the conflict set of π is included in the two half-spaces defined by the ridge and hence included in the conflict sets for the four corners. In fact using just two corners, one from each plane, would suffice.

The other case is when we are adding x to a facet that already contains at least 3 points (the degenerate case). This corresponds to adding a point to a 2D convex hull in the plane of the facet. Considering other points in this plane, the new corner is supported by the two corners in the plane with y and z as their corner points (as would be the case in a 2D convex hull). Considering points not on the plane, the new corner is supported by the other two corners—i.e., the ones with y and z as their corner points but not in the plane containing x, y, and z.

Now we consider x when it is not the corner point of π. In this case, (π, x) has 2-support. In particular, without loss of generality, let y be the corner point of π. Then, y will have one ridge incident on it removed by x. This ridge is defined by two facets, each of which contains a corner configuration with corner point y. (π, x) will be supported by just these two corners.

In all of the above cases, the corner configuration is supported by at most four corners. □

With Lemmas 6.1 and 6.2, we know that Theorem 5.3 still holds for inputs with degenerate points. With some minor changes to our CRCW PRAM algorithm (accounting for up to 4 conflicts instead of 2), Theorem 5.4 also holds.

7 OTHER CONFIGURATION SPACES WITH k-SUPPORT

We now discuss some other uses of configuration space with k-support. First, we mention that there is another formulation of convex hull. Here we describe the version for points in general position. The objects are points, but the configurations correspond to ridges of the convex hull with their two neighboring facets. Each such configuration can be defined by $d + 1$ objects: the $d - 1$ points on the ridge and the two points sharing facets with the ridge. Every set of $d + 1$ points define up to $\binom{d+1}{d-1}$ configurations, since subsets of size $d - 1$ will specify the ridge. It therefore has constant multiplicity for constant dimension. The conflict set of a configuration is all the points visible from either of its facets. The configuration space has 2-support. Both of the points not on the shared ridge have support sets of size one. In particular, the support for (π, x) will consist of a configuration with the same opposite facet, along with the previous facet sharing the ridge. The points on the shared ridge have support sets of size two. In particular, the support consists of the two configurations, each consisting of an opposite ridge to x (there are two such ridges), the facet on the other side of that ridge, and the facet previously sharing the opposite ridge.

This formulation has the property that adding a facet deletes all of its support set. This makes it easier to apply Theorem 3.1 to determine the total work of the algorithm (all conflicts on a configuration are examined once and then the configuration is removed). The corresponding algorithm, however, is more complicated.

By duality, we can use convex hulls to find the intersection of a set of intersecting half-spaces in d dimensions. However, it is also helpful to consider a direct formulation. In this case, the objects are the half-spaces, and the configurations are intersections of d half-spaces, which define a point. A configuration conflicts with a half-space if it is not contained in the half-space. As with convex hulls, such configurations have 2-support. In particular, for a configuration (point) π and object (half-space) $x \in D(\pi)$, the half-space x when added will cut a 1-dimensional edge between two existing points. Those two points support (π, x) since any half-space that conflicts with x must conflict with at least one of those points. Boundaries can be handled by using configurations with $d - 1$ half-spaces and a direction along the shared edge signifying infinity along the shared edge in that direction.

Finally, we consider the problem of finding the intersection of a set of unit circles [28]. In this case, the objects are the circles and the configurations are arcs defined by intersecting either two or three circles. For two circles, there are two arcs that bound the intersection of the circles, and for three circles there are three. The multiplicity is therefore bounded by three. An arc conflicts with any circle that overlaps with it but does not fully contain it. This configuration space has 2-support. Consider an arc π defined by three circles. If x is the circle on which the arc is a boundary, then (π, x) has a support of size two consisting of the two arcs that are cut at the ends of π by adding x. If x is one of the other circles, then (π, x) has a singleton support set on the one side for the arc being cut. Similarly, if the arc is defined by two circles, it has a support set of size one for the arc on the other circle being cut.

8 CONCLUSION

We showed that the randomized incremental convex hull algorithm is inherently parallel with $O(\log n)$ dependence depth. Based on this, we have presented a simple work-efficient polylogarithmic-span algorithm for convex hull in constant dimensions, which is perhaps even simpler than the standard sequential variant. The key idea is analyzing the dependence in the configuration space based on facets, and showing that adding a facet only depends on a constant number of existing facets (support set). We showed this within a more general setting based on configuration spaces, and the definition of support sets and k-support. This work follows a line of work that aims at exposing inherent parallelism in seemingly sequential algorithms [3, 4, 14, 17, 36, 43, 51, 55].

Although our analysis of k-support applies to convex hull, Delaunay triangulation, half-space intersection, and circle intersection, it does not cover some other problems that use configuration spaces. In particular, it does not cover the standard randomized incremental algorithm for trapezoidal decomposition [28, 32, 50]. It is an open question whether some other formulation of trapezoidal decomposition fits within the framework, or whether the framework can be extended to incorporate trapezoidal decomposition.

Acknowledgements. This research was supported by DOE Early Career Award #DE-SC0018947, NSF CAREER Award #CCF-1845763, NSF grants CCF-1910030 and CCF-1919223, Google Faculty Research Award, DARPA SDH Award #HR0011-18-3-0007, and the Applications Driving Algorithms (ADA) Center, a JUMP Center co-sponsored by SRC and DARPA.

REFERENCES

[1] Umut Acar, Guy E. Blelloch, and Robert Blumofe. The data locality of work stealing. *Theory of Computing Systems (TOCS)*, 35(3):321–347, 2002.
[2] Kunal Agrawal, Jeremy T. Fineman, Kefu Lu, Brendan Sheridan, Jim Sukha, and Robert Utterback. Provably good scheduling for parallel programs that use data structures through implicit batching. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2014.
[3] Dan Alistarh, Trevor Brown, Justin Kopinsky, and Giorgi Nadiradze. Relaxed schedulers can efficiently parallelize iterative algorithms. In *ACM Symposium on Principles of Distributed Computing (PODC)*, pages 377–386, 2018.
[4] Dan Alistarh, Giorgi Nadiradze, and Nikita Koval. Efficiency guarantees for parallel incremental algorithms under relaxed schedulers. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 145–154, 2019.
[5] Nancy M. Amato, Michael T. Goodrich, and Edgar A. Ramos. Parallel algorithms for higher-dimensional convex hulls. In *IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 683–694, 1994.
[6] Nancy M. Amato and Franco P. Preparata. The parallel 3D convex hull problem revisited. *International Journal of Computational Geometry & Applications*, 2(02):163–173, 1992.
[7] Mikhail J. Atallah, Richard Cole, and Michael T. Goodrich. Cascading divide-and-conquer: A technique for designing parallel algorithms. *SIAM J. on Computing*, 18(3):499–532, June 1989.
[8] Mikhail J. Atallah and Michael T. Goodrich. Efficient parallel solutions to some geometric problems. *Journal of Parallel and Distributed Computing*, 3(4):492 – 507, 1986.
[9] Mikhail J. Atallah and Michael T. Goodrich. Parallel algorithms for some functions of two convex polygons. *Algorithmica*, 3(4):535–548, 1988.
[10] Brad Barber. Qhull. http://www.qhull.org/html/index.htm, 2015.
[11] Naama Ben-David, Guy E. Blelloch, Jeremy T. Fineman, Phillip B. Gibbons, Yan Gu, Charles McGuffey, and Julian Shun. Parallel algorithms for asymmetric read-write costs. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 145–156, 2016.
[12] Guy E. Blelloch, Daniel Ferizovic, and Yihan Sun. Just join for parallel ordered sets. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 253–264. ACM, 2016.
[13] Guy E. Blelloch, Jeremy T. Fineman, Yan Gu, and Yihan Sun. Optimal (randomized) parallel algorithms in the binary-forking model. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2020.
[14] Guy E. Blelloch, Jeremy T. Fineman, and Julian Shun. Greedy sequential maximal independent set and matching are parallel on average. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 308–317, 2012.
[15] Guy E. Blelloch, Phillip B. Gibbons, and Harsha Vardhan Simhadri. Low depth cache-oblivious algorithms. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2010.
[16] Guy E. Blelloch and Yan Gu. Improved parallel cache-oblivious algorithms on dynamic programming. *SIAM Symposium on Algorithmic Principles of Computer Systems (APOCS)*, 2020.
[17] Guy E. Blelloch, Yan Gu, Julian Shun, and Yihan Sun. Parallelism in randomized incremental algorithms. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 467–478, 2016.
[18] Guy E. Blelloch, Yan Gu, Julian Shun, and Yihan Sun. Parallel write-efficient algorithms and data structures for computational geometry. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2018.
[19] Jean-Daniel Boissonnat, Olivier Devillers, René Schott, Monique Teillaud, and Mariette Yvinec. Applications of random sampling to on-line algorithms in computational geometry. *Discrete & Computational Geometry*, 8(1):51–71, Jul 1992.
[20] Jean-Daniel Boissonnat and Monique Teillaud. On the randomized construction of the Delaunay tree. *Theoretical Computer Science*, 112(2):339–354, 1993.
[21] Jean-Daniel Boissonnat and M. Yvinec. *Algorithmic Geometry.* Cambridge University Press, 1998.
[22] Richard P. Brent. The parallel evaluation of general arithmetic expressions. *J. ACM*, 21(2):201–206, April 1974.

[23] Zoran Budimlic, Vincent Cave, Raghavan Raman, Jun Shirako, Sagnak Tasirlar, Jisheng Zhao, and Vivek Sarkar. The design and implementation of the Habanero-Java parallel programming language. In *Symposium on Object-oriented Programming, Systems, Languages and Applications (OOPSLA)*, pages 185–186, 2011.
[24] Matt Campbell. MIConvexHull. https://designengrlab.github.io/MIConvexHull/, 2017.
[25] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph Von Praun, and Vivek Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In *Symposium on Object-oriented Programming, Systems, Languages and Applications (OOPSLA)*, volume 40, pages 519–538, 2005.
[26] A. Chow. *Parallel Algorithms for Geometric Problems.* PhD thesis, Department of Computer Science, University of Illinois, Urbana-Champaign, December 1981.
[27] Marcelo Cintra, Diego R. Llanos, and Belén Palop. Speculative parallelization of a randomized incremental convex hull algorithm. In *International Conference on Computational Science and Its Applications*, pages 188–197, 2004.
[28] Kenneth L. Clarkson and Peter W. Shor. Applications of random sampling in computational geometry, II. *Discrete & Computational Geometry*, 4(5):387–421, 1989.
[29] Richard Cole and Vijaya Ramachandran. Resource oblivious sorting on multicores. *ACM Transactions on Parallel Computing (TOPC)*, 3(4), 2017.
[30] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms (3. ed.).* MIT Press, 2009.
[31] Norm Dadoun and David G. Kirkpatrick. Parallel construction of subdivision hierarchies. *J. Computer and System Sciences*, 39(2):153–165, 1989.
[32] Mark de Berg, Otfried Cheong, Marc van Kreveld, and Mark Overmars. *Computational Geometry: Algorithms and Applications.* Springer-Verlag, 2008.
[33] Laxman Dhulipala, Guy E. Blelloch, and Julian Shun. Theoretically efficient parallel graph algorithms can be fast and scalable. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2018.
[34] Pedro Diaz, Diego R. Llanos, and Belen Palop. Parallelizing 2D-convex hulls on clusters: Sorting matters. *Jornadas De Paralelismo*, 2004.
[35] Herbert Edelsbrunner. *Geometry and Topology for Mesh Generation.* Cambridge University Press, 2006.
[36] Manuela Fischer and Andreas Noever. Tight analysis of parallel randomized greedy mis. In *ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 2152–2160, 2018.
[37] Matteo Frigo, Charles E Leiserson, and Keith H Randall. The implementation of the Cilk-5 multithreaded language. *ACM Conference on Programming Language Design and Implementation (PLDI)*, 33(5):212–223, 1998.
[38] Mingcen Gao, Thanh-Tung Cao, Ashwin Nanjappa, Tiow-Seng Tan, and Zhiyong Huang. gHull: A GPU algorithm for 3D convex hull. *ACM Transactions on Mathematical Software*, 40(1):3:1–3:19, October 2013.
[39] J. Gil, Y. Matias, and U. Vishkin. Towards a theory of nearly constant time parallel algorithms. In *IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 698–710, 1991.
[40] Arturo Gonzalez-Escribano, Diego R. Llanos, David Orden, and Belen Palop. Parallelization alternatives and their performance for the convex hull problem. *Applied Mathematical Modelling*, 30(7):563 – 577, 2006.
[41] Michael T. Goodrich, Yossi Matias, and Uzi Vishkin. Optimal parallel approximation for prefix sums and integer sorting. In *ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 1994.
[42] Neelima Gupta and Sandeep Sen. Faster output-sensitive parallel algorithms for 3D convex hulls and vector maxima. *Journal of Parallel and Distributed Computing*, 63(4):488–500, April 2003.
[43] William Hasenplaugh, Tim Kaler, Tao B Schardl, and Charles E Leiserson. Ordering heuristics for parallel graph coloring. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 166–177, 2014.
[44] https://www.threadingbuildingblocks.org.
[45] Joseph JáJá. *An Introduction to Parallel Algorithms.* Addison Wesley, 1992.
[46] http://docs.oracle.com/javase/tutorial/essential/concurrency/forkjoin.html.
[47] Diego R. Llanos, David Orden, and Belen Palop. MESETA: A new scheduling strategy for speculative parallelization of randomized incremental algorithms. *International Conference on Parallel Processing Workshops*, pages 121–128, 2005.
[48] Mikola Lysenko. incremental-convex-hull. https://github.com/mikolalysenko/incremental-convex-hull, 2014.
[49] Russ Miller and Quentin F. Stout. Efficient parallel convex hull algorithms. *IEEE Trans. on Comput.*, 37(12):1605–1618, December 1988.
[50] Ketan Mulmuley. *Computational geometry - an introduction through randomized algorithms.* Prentice Hall, 1994.
[51] Xinghao Pan, Dimitris Papailiopoulos, Samet Oymak, Benjamin Recht, Kannan Ramchandran, and Michael I. Jordan. Parallel correlation clustering on big graphs. In *Advances in Neural Information Processing Systems (NIPS)*, pages 82–90, 2015.
[52] John H. Reif and Sandeep Sen. Optimal randomized parallel algorithms for computational geometry. *Algorithmica*, 7(1-6):91–117, 1992.
[53] Raimund Seidel. Small-dimensional linear programming and convex hulls made easy. *Discrete & Computational Geometry*, 6(3):423–434, 1991.
[54] Raimund Seidel. Backwards analysis of randomized geometric algorithms. In *New Trends in Discrete and Computational Geometry*, pages 37–67. 1993.
[55] Julian Shun, Yan Gu, Guy E. Blelloch, Jeremy T. Fineman, and Phillip B. Gibbons. Sequential random permutation, list contraction and tree contraction are highly parallel. In *ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 431–448, 2015.
[56] Ayal Stein, Eran Geva, and Jihad El-Sana. Applications of geometry processing: CudaHull: Fast parallel 3D convex hull on the GPU. *Comput. Graph.*, 36(4):265–271, June 2012.
[57] https://msdn.microsoft.com/en-us/library/dd460717 %28v=vs.110%29.aspx.
[58] The CGAL Project. *CGAL User and Reference Manual.* CGAL Editorial Board, 4.14 edition, 2019.
[59] Stanley Tzeng and John D. Owens. Finding convex hulls using Quickhull on the GPU. *CoRR*, abs/1201.2936, 2012.
[60] Uzi Vishkin. Thinking in parallel: Some basic data-parallel algorithms and techniques, 2010. Course notes, University of Maryland.

A IMPLEMENTING Insert-and-set USING Test-and-set

In Algorithm 3, we presented our parallel convex hull algorithm. On a CRCW PRAM, our algorithm can run synchronously in rounds, where on each round we pack all new ridges to be processed in the next round using a hash table [39], which costs linear work and $O(\log^* n)$ span *whp*. However, on modern machines, global synchronization can be expensive. We note that Algorithm 3 is recursive and does not require synchronization. The only additional places that need more discussion are Lines 20 and 21, where two facets incident on one ridge r have to communicate and decide which facet will process r. Our approach is based on using a global multimap M using InsertAndSet and GetValue functions. In particular, for the two facets associated with r, whichever calls InsertAndSet second will be responsible for processing r.

In Section 5.2, we have shown how to use CompareAndSwap to implement InsertAndSet. As discussed in [13], in contrast to TestAndSet, CompareAndSwap is a stronger primitive that is not assumed in the binary-forking model by default. In this section, we show how to implement the InsertAndSet(r, t) and GetValue(r) functions on a multimap M using only TestAndSet, where a key r is associated with at most two values. In this case, the InsertAndSet function checks in a hash table if ridge r has been already added to it—if so, it returns False, but also writes t as the second value of r; otherwise, it inserts r into the hash table with the value t, and returns True. GetValue(r, t) looks up the values associated with r in M, and returns the facet $t' \neq t$.

Algorithm 5 shows the implementation of the two functions on M. M maintains a linear probing hash table R, which stores key-value pairs and allows for duplicate keys. For each cell in R, there are three fields: two boolean flags, `taken` and `check`, and `data` which will store the corresponding key-value pair. Assume the hash function for R is f_R, which hashes a ridge r to an index in the range $[0, \ldots, |R| - 1]$. When inserting a key-value pair (r, t) into R, the algorithm first computes the hash value $i = f_R(r)$ of r (Line 2) and tries to reserve the slot with a TestAndSet on $R[i]$.`taken` (Line 3). If a conflict (due to concurrency) or a collision (due to hashing) occurs, R uses linear probing to attempt to assign the next slot to (r, t) (the while-loop on Lines 3–4). After reserving a slot $R[i]$, the algorithm writes (r, t) into the `data` field. Note that even if the key k already exists, R finds a slot for it using linear probing. Therefore, each key-value pair is ensured to be added to R and an insertion on R never fails. Then the algorithm makes a second pass

Algorithm 5: InsertAndSet and GetValue on a multimap M.

```
InsertAndSet(r, t)
  Input: A ridge r and a facet t
  Maintains:
      A hash table R with linear probing from keys as ridges
      and values as facets. Each cell in R has three fields: two
      boolean flags, taken and check, and data which will store
      the corresponding key-value pair. Assume the hash
      function for R is $f_R$, which hashes a ridge r to an index in
      the range $[0, \ldots, |R| - 1]$.
  Output: If $r \in M$, add t as the second value of r in M and
          return false. Otherwise, add t as the first value of r in
          M and return true.
1  function M.InsertAndSet(r, t)
2      $i \leftarrow f_R(r)$                // get the starting index
3      while $\neg$TestAndSet(R[i].taken) do
4          $i \leftarrow (i + 1)$ Mod M.size
5      R[i].data $\leftarrow (r, t)$          // write entry to data
6      $i \leftarrow f_R(r)$                // start from initial index
7      while R[i].taken do
8          if R[i].data.key = r then
9              if $\neg$TestAndSet(R[i].check) then
10                 return False
11         $i \leftarrow (i + 1)$ Mod M.size
12     return True
GetValue(r, t)
  Input: A ridge r and a facet t.
  Output: The values $t' \neq t$ associated with r in M.
13 function M.GetValue(r, t)
14     $i \leftarrow f_R(r)$                // get the starting index
15     while R[i].taken do
16         if R[i].data.key = r then
17             $t \leftarrow$ R[i].data.value
18             if $t' \neq t$ then return $t'$
19         $i \leftarrow (i + 1)$ Mod M.size
```

over R starting at the initial index from the hash value, and tries to find r in R. It stops when it sees an empty slot. Whenever the algorithm sees a slot with key r, it performs a TestAndSet on the check field of the slot. If the TestAndSet on Line 9 fails, we let this InsertAndSet return false (Line 10), and the corresponding facet t will process r in Algorithm 3. Otherwise, if a facet t does not fail the TestAndSet, it returns true (Line 12). Note that when making the second pass, a InsertAndSet(r, t) algorithm may see a slot with a key other than r due to linear probing. The algorithm thus needs to check if the key is equal to r (Line 8) during the while-loop.

The GetValue(r, t) algorithm starts at the index of the hash value of r, and scans until it finds a facet $t' \neq t$ associated with r.

We next prove the correctness of the algorithm. We first prove that InsertAndSet works as expected. It is straightforward that the insertion into the hash table (Lines 2–5) works as expected. We then need to show that the returned boolean value of InsertAndSet works as expected. Note that for a ridge r, there will be exactly two facets that call InsertAndSet on r throughout the algorithm. A correct InsertAndSet algorithm allows for exactly one of them to return False. This unsuccessful InsertAndSet will take over to recurse on ridge r on Line 22 of Algorithm 3.

Theorem A.1. *For two invocations to* InsertAndSet(r, t_1) *and* InsertAndSet(r, t_2)*, exactly one of them returns* False.

Proof. Suppose the two indices for ridge r in the hash table R are i (reserved by t_1) and $j > i$ (reserved by t_2). The other case is symmetric. We first show that there must be one of t_1 and t_2 that returns False in InsertAndSet.

Case 1. We call TestAndSettwice on $R[i]$.check on Line 9. Then, the second one must fail and return False.

Case 2. We call TestAndSetonly once on $R[i]$.check on Line 9. This can only happen if when t_2 is making the second pass, t_1 has marked $R[i]$.taken, but has not written its key r to $R[i]$.data. Then, for both t_1 and t_2, when they make the second pass, they must both reach $R[j]$ and call TestAndSet on $R[j]$.check on Line 9. One of them has to fail and return False.

Secondly, we show that it is impossible that t_1 and t_2 both return false in InsertAndSet. The TestAndSet on $R[i]$.check can only fail once, and similarly for the TestAndSet on $R[j]$.check. If both t_1 and t_2 return false, then this means that Line 9 fails on both $R[i]$ and $R[j]$. This is impossible because whichever fails on $R[i]$ will not proceed to $R[j]$, and so $R[j]$ cannot be reached twice. □

We next show that GetValue(r, t) works as expected. In particular, both facets t' and t'' associated with r in R must have been inserted into R, and thus the one that is not equal to t will be found and returned.

Theorem A.2. *When* GetValue(r, t) *is called, the two facets* t_1 *and* t_2 *incident on* r *have both been added into* R.

Proof. A GetValue(r, t) is called only when an invocation of InsertAndSet(r, t) returns False on Line 20 of Algorithm 3. This InsertAndSet fails because it fails on Line 9 of Algorithm 5. This means that the key-value pair (r, t) itself has been added to R on Line 5 of Algorithm 5.

Suppose that the other facet incident on r is t'. We will show that (r, t') has also been added to R. Since (r, t) fails on the TestAndSet on Line 9 of Algorithm 5, it means that the check flag has been already set to True by (r, t') before (r, t) processes it. Since the algorithm InsertAndSet(r, t') reached Line 9 to set the check flag, it also must have finished adding the data on Line 5. □

In summary, the multimap M with its two functions InsertAndSet and GetValue works as expected by using TestAndSet. The work and depth are just the cost of linear probing, which is $O(\log n)$ *whp*.