Highly Parallel and Cache-Optimal Construction of 2D Convex Hulls

Reilly Browne
Department of Computer Science, Stony Brook University

Rezaul A. Chowdhury
Department of Computer Science, Stony Brook University

Shih-Yu Tsai
Department of Computer Science, Stony Brook University

Yimin Zhu
Department of Computer Science, Stony Brook University

Abstract

We present three new parallel 2D convex hull algorithms in the binary-forking model. One of them is a deterministic algorithm that finds a convex hull cache-optimally in the worst-case optimal $O(n \log n)$ work while matching the best known span of $O(\log n \log \log n)$ for this model. We also present a parameterized algorithm that achieves $O(k \log n)$ span at ~~the~~ a cost of $O\left(n^{1+\frac{1}{k}} \log n\right)$ work for any integer $k \in [1, \log n]$. These results allow us to also adapt a very recent randomized parallel sorting algorithm to construct a convex hull in $O(n \log n)$ work and $O(\log n)$ span, both *whp* in n. These algorithms exploit a connection between the convex hull of a set of n points in 2D Euclidean space and the upper envelope of a set of n sinusoidal waves.

2012 ACM Subject Classification Computing methodologies → Massively parallel algorithms; Theory of computation → Computational geometry

Keywords and phrases Convex hull, Binary Forking model, parallel algorithms, computational geometry

Digital Object Identifier 10.4230/LIPIcs.ISAAC.2022.23

Acknowledgements We would like to thank Pramod Ganapathi, for useful discussion

1 Introduction

Finding the convex hull of a set of n points in d dimensions is one of the most fundamental problems in computational geometry. It has wide applications, such as robot motion planning in robotics, image processing in pattern recognition, and tracking disease epidemics in ethology [41, 24, 4]. In the serial setting, there have been several efficient algorithms for constructing convex hulls [19, 35, 45, 25]. In the parallel setting, a bunch of efficient convex hull algorithms have been developed for a set of d-dimensional points [5, 8, 9, 36, 42, 47, 17]. There are also many parallel convex hull algorithms specifically for presorted or unsorted points in 2 dimensions [12, 30, 20, 42]. In this paper, we focus on designing parallel algorithms for the convex hull of a set of points in 2 dimensions in the binary-forking model [15].

We use the *work-span* model [23] to analyze the performance of parallel algorithms. The *work*, $W(n)$ of a parallel algorithm is defined as the total number of CPU operations it performs when it is executed on a single processor. Its *span*, $S(n)$ on the other hand, is the maximum number of operations performed by any single processor when the program runs on an unlimited number of processors.

The binary-forking model [15, 1, 11, 13, 14] realistically captures the performance of parallel algorithms designed for modern multicore shared-memory machines. ~~Its formal definition is in [15]~~. It has been def'd formally by Blelloch et al. [15]. In this model, the computation starts with a single thread, and then threads are dynamically and asynchronously created by some existing threads as time progresses. The creation of threads is based on the spawn/fork action: a thread can spawn/fork a concurrent asynchronous child thread while it continues its task. Note that the forks can happen recursively. The model also includes a join operation to synchronize the threads and an atomic test-and-set (TS) instruction. This model can be viewed as an extension of the binary fork-join model [23] which does not include the TS instruction. This model and its variants [23, 1, 2, 13, 18, 16] are widely used in many parallel programming languages or environments such as Intel TBB [50], the Microsoft Task Parallel Library [49], Cilk [28], and the Java fork-join framework [26].

\emph \emph it includes?

The binary-forking model is an ideal candidate for modeling parallel computations on modern architectures when compared with the closely related PRAM model [37]. The main difference between the binary-forking model and the PRAM model is synchronicity. The binary-forking model allows asynchronous thread creation; in the PRAM model, all processors work in synchronous steps. PRAM does not model modern architectures well because they utilize new techniques such as multiple caches, branch prediction, and many more, which lead to many asynchronous events such as varying clock speed, cache misses, etc. [15]. Any algorithm designed for the PRAM model can be transformed into an algorithm for the binary-forking model at the cost of an $O(\log n)$-factor blow-up in the span while keeping the work asymptotically the same as in the PRAM model.

We also employ ~~the use of~~ the cache-oblivious model, first described by Frigo et al. [27]. In this model, memory is assumed to have two or more layers, a cache of size M and a main memory of unlimited size. The memory is split into blocks of size B, and every time the processor tries to access a data point that is not in the cache, it incurs a cache miss and the block containing the data point is copied into the cache from the main memory. When copying into a cache that is already full, an old block is evicted to make space for the new block. However, in contrast to the cache-aware model, cache-oblivious algorithms do not use the knowledge of the values of M and B. In both models, cache complexity of an algorithm is measured in terms of the number of cache misses it incurs and is referred to as cache-optimal if it incurs the fewest possible cache misses asymptotically. \emph

In terms of cache analysis for convex hulls, ~~the~~ one of the earliest serial algorithms developed by Graham [35], when combined with a cache oblivious sorting algorithm, ~~could~~ achieves $O(n/B \log_M n)$ cache misses. Arge and Miltersen [7] showed that this bound is optimal for non-output sensitive convex hull algorithms in the cache-aware model, which carries over to the cache-oblivious model. When output sensitivity is accounted for, where h is the number of points comprising the convex hull, this bound decreases to $O(n/B \log_{M/B}(h/B)$, as is achieved by Goodrich et al. [34] for the external memory model. In terms of parallel cache-oblivious algorithms, Sharma and Sen [48] presented a randomized CRCW algorithm which achieves expected $O(n/B \log_M n)$ cache misses and expected $O(\log n \log\log n)$ span.

Murakami et al. [43] showed that the problem of finding the convex hull of a given set of points can be reduced to the problem of finding the upper envelope of a set of phase-shifted sine waves of the same frequency. The connection can be seen by taking the projection of a point (x, y) onto a straight line passing through the origin that forms an angle θ with the horizontal axis (See Figure 1). If we rotate this line with respect to the origin, the distance $l_{x,y,\theta}$ of the point of projection of (x, y) from the origin forms the sinusoidal curve $l_{x,y,\theta} = x \cos\theta + y \sin\theta$. This sinusoidal curve is called the "Hough curve" [43] of point (x, y) and the mapping of (x, y) to $l_{x,y,\theta}$ is called the "Hough transform". For a given set S of points transformed to have their geometric center at the origin, every point $(x, y) \in S$ that has the largest $l_{x,y,\theta}$ value among all points in S for any angle θ must be an extreme point (i.e., must \emph \emph lie on the convex hull). Thus finding the upper envelope (or skyline) of these sine waves is the same as finding the extreme points of the convex hull of S.

There has not been much exploration of convex hull construction using the Hough transform since the connection was first proposed more than 30 years ago by Murakami et al. [43]. They presented a serial approximation algorithm that runs in $O(KL)$ time for parameters K and L which are set to values much larger than n chosen empirically for a good approximation. Then ~~in 1996,~~ Wright et al. [51] provided two serial algorithms which run in $O\left(\frac{n}{\tan^{-1}(1/p)}\right)$ and $O\left(nh\right)$ time, respectively, where p is the greatest distance between any two points in the input and h is the number of points that define the boundary of the hull. All three algorithms presented in this paper use the Hough transform, but to the best of our knowledge, no existing parallel convex hull algorithms use it.

Our Contributions. In summary, we have the following results:

- A deterministic cache-oblivious algorithm based on multi-way merging which performs worst-case optimal $O(n \log n)$ work in $O(\log n \log \log n)$ span, and achieves optimal parallel cache complexity. This algorithm uses neither the atomic test-and-set operation nor concurrent writes. To the best of our knowledge, no existing deterministic 2D convex hull algorithm for the binary-forking model (including those implied by known exclusive-write PRAM algorithms [52, 42, 30]) has span lower than that achieved by our algorithm and none of them are cache-efficient. Its span is lower than the recently proposed randomized incremental convex hull algorithm for the binary-forking model as well [17].
- A deterministic parameterized convex hull algorithm that achieves $O(k \log n)$ span with $O(n^{1+\frac{1}{k}} \log n)$ work for a positive integer parameter k.
- A randomized convex hull algorithm that achieves $O(\log n)$ span and $O(n \log n)$ work, both *whp* in n, based on a randomized sorting algorithm [3] in the binary-forking model. ?

All our bounds hold for the binary fork-join model as well except for the parameterized algorithm which uses the atomic TS instruction for list-ranking.

The paper is organized as the following. We discuss the related work in Section 2 and show the two representations for the upper envelope of a set of sine waves in Section 3. Based on these two representations, we propose three algorithms for convex hull in Section 4, Section 5, and Section 6 respectively. In the end, Section 7 concludes our work.

2 Related Work

I think Chan achieved $O(n \log h)$.

Considering non-output-sensitive serial algorithms, the best running time that can be reached is $O(n \log n)$. Graham Scan [35] was the first algorithm that achieves this optimal running time. There are several other approaches proposed afterward that have the same running time as well [6, 45, 10, 39]. For the output-sensitive serial algorithms, the Gift Wrapping method achieves a running time of $O(nh)$ [38], which was later improved to $O(n \log h)$ [40, 19].

Efficient convex hull algorithms have been designed in various parallel models. With presorted input, the optimal bound of span and work in PRAM models are $(O(\log n), O(n))$ (the first element of this tuple represents the asymptotic span bound and the second one represents the asymptotic work bound) in the EREW (exclusive reads and writes) model [52] and $(O(\log \log n), O(n))$ in the CRCW (concurrent reads and writes) model [12]. For the unsorted input, the optimal span is $O(\log n)$ and work is $O(n \log h)$ in the EREW model [42] and randomized CRCW with n-exponential probability [31].

A randomized incremental convex hull algorithm is the first to be analyzed in binary-forking model [17]. It performs $O(n \log n)$ expected work. Its span is $O(\log n \log^* n)$ in

The ?

PRAM model and $O(\log^2 n)$ in the binary-forking model with high probability in n. There ha~~ve~~s been a recent surge of interest in designing parallel algorithms for solving various problems on variants of the binary-forking model [21, 46, 15, 22, 33, 3]

3 Preliminaries

In this paper, we use ~~2~~ two different representations of the upper envelope of a set of sine waves. First, we are restricted to sine waves of the form $x \cos \theta + y \sin \theta$ as only these waves can be obtained from the Hough transform of a set of points. Therefore, all representations of the waves themselves are stored simply as tuples (x, y).

3.1 Vector-range form

The first data structure is used in our algorithms is *Vector-Range Form*, as it is the most intuitive. Every wave in the envelope has its vector representation, the range of angles over which it is in the envelope, its rank in the envelope, and the curve which succeeds it in rank order. These are usually stored in an array in rank order but can also be stored as a linked list.

3.2 Primitives

space too long

We use ~~5~~ five primitive functions, *MaxVal*, *MaxSlope*, *MaxToRight*, *Intersections*, and *CommonRange*. Each of them performs $O(1)$ work. We provide pseudocode for them in Appendix B, but it suffices to note that since their input sizes are constant, it is impossible for them to have greater time complexity than $O(1)$. *MaxVal* and *MinVal* return the a? wave of higher or lower value between the two input waves at a given angle. *MaxSlope* and *MinSlope* do the same but for slope instead of value. *MaxToRight* and *MaxToLeft* do the same as MaxVal but in case of a tie, they return the one which continues to be on top to the right and left of the given angle, respectively. *Intersections* determines the points of intersection between two input waves (where they cross each other). *CommonRange* determines the intersection of two intervals in $[0, 2\pi)$ (the range of angles common to both intervals).

3.3 Positive curve form

Another data structure we use is *Positive Curve Form* (PCF). This structure maintains a list of angle-wave pairs which represent the upper envelope of a given set of sine waves and the x-axis (or $0 \cos \theta + 0 \sin \theta$). The result is an envelope for which only positive values are considered.

An envelope with only a single wave is represented differently depending on where in the range $[0, 2\pi)$ it is above the x-axis. If a curve intersects the x-axis at θ_1, θ_2 where $\theta_1 < \theta_2$ and its value is negative for angles $\theta_1 < \theta < \theta_2$, then ~~that~~ it is a "split waves". Split waves are represented as two separate waves, one for the $[0, \theta_1]$ component and the other for the $[\theta_2, 2\pi]$ component. See Figure 2 for an example. For non-split curves, we represent them as a single wave. See Figure 3 for an example

In LaTeX, use `` ''

For each curve, we associate it in a pair with the first angle for which it is active. We also record the gaps where the x-axis dominates, with those being represented as an angle and a hyphen "-". Lastly, we record the end of an envelope with a "." and the angle 2π. These ends are not considered as waves by our algorithms. See Figures 4 and 5 for examples of upper envelopes.

Don't use bitmaps, but vector graphics (pdf)!

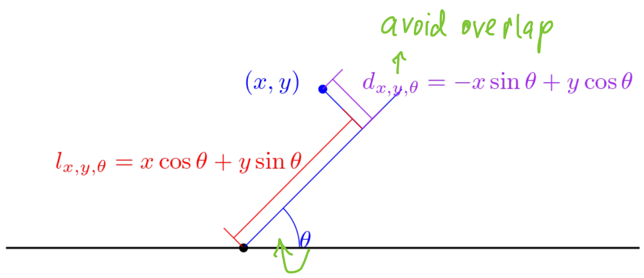
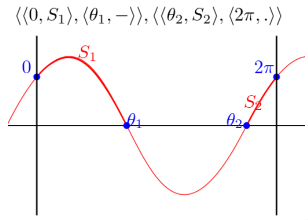


Figure 1 ~~Length~~ The Length of the projection of a point (x, y) on a straight line passing through the origin making θ angle with the horizontal axis is given by $l_{x,y,\theta} = x\cos\theta + y\sin\theta$ (the Hough curve). The perpendicular distance of the point from the line is given by $d_{x,y,\theta} = -x\sin\theta + y\cos\theta$ (another sinusoidal curve).



What about abbreviating to PCF?

Figure 2 Positive Curve Form for split waves

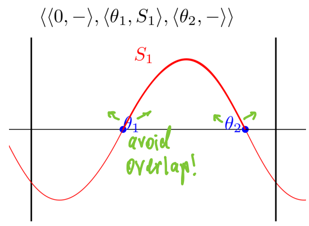
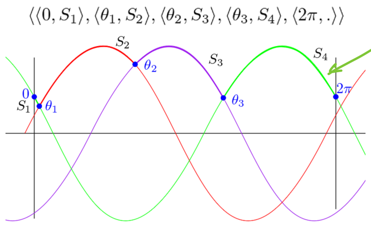


Figure 3 Positive Curve Form for non-split waves



Make thick curves even thicker!

Figure 4 Positive Curve Form for an upper envelope

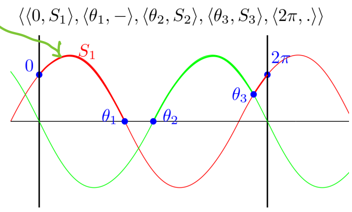


Figure 5 Positive Curve Form for an envelope with gaps

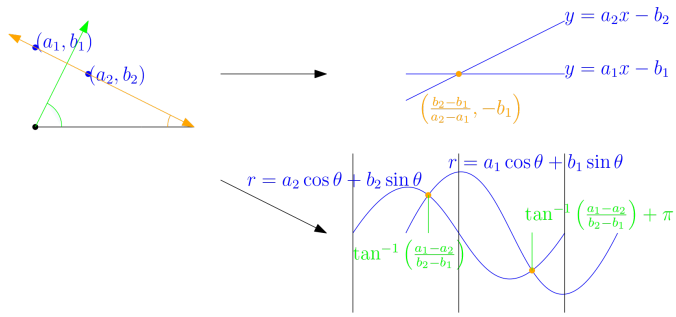


Figure 6 Showing the relationship between Hough transform and point-line duality

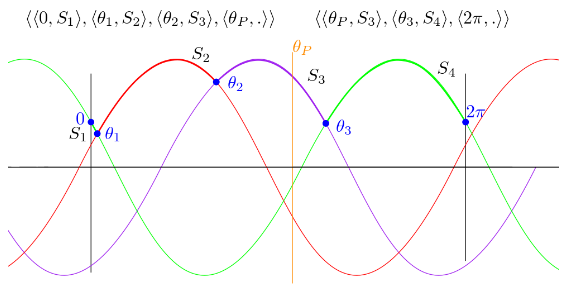


Figure 7 Showing how the pivots work in Algorithms 2

Unless stated otherwise, the algorithms in this paper use the Vector-Range Form to represent the upper envelope. A curve can be converted from Vector-Range Form to Positive-Curve Form in constant time using the Intersections primitive with the wave $0 \cos \theta + 0 \sin \theta$ (the x-axis).

3.4 Equivalence of transformations

A common technique used in convex hull construction is finding the lower and upper envelopes of the dual representations of all the points under the standard point-line duality transformation. As opposed to our method of finding the upper envelope of sinusoidal curves, this method finds the upper envelope of lines. The advantage of using point-line duality is that there is no need to use trigonometric functions (which may take $\omega(1)$ work to compute with the required precision) to compute the relevant intersections along the envelope. However, using point-line duality requires two runs, one to find the upper envelope (corresponding to the lower hull) and one to find the lower envelope (corresponding to the upper hull).

For conceptual reasons, we use the sinusoidal model. However, our algorithms can be easily adapted to use point-line duality instead. This is because the relevant intersections along both envelopes are essentially the same. The two angles at which the sinuisoidal waves intersect correspond to a line coming from the origin which is perpendicular to the line formed by the two points. The intersection point in the dual plane for point-line duality translates back into the primal plane as the line formed by the two points. See Figure ~~5~~ 6 for a demonstration of this.

Thus, instead of keeping track of the θ ranges for which a given wave is on top, we can instead keep track of the $-\cot \theta$ range for which a given wave would be on top to find the equivalent of the lower hull. This is because the negative cotangent of the angle is equal to the slope of the point-duality line, and the slope of that line is the x-value of the intersection point in the dual. This is essentially equivalent to taking the $r = a \cos \theta + b \sin \theta$ representation and dividing all by $-\sin \theta$ to get $\frac{r}{-\sin \theta} = a \cot \theta - b$, which is equivalent to the point-line duality transformation, $y = ax - b$.

4 A Cache- and Work-Optimal $\Theta(\log n \log \log n)$ Span Algorithm

We propose a divide-and-conquer algorithm which performs $O(n \log n)$ work and $O(\log n \log \log n)$ span. It is based heavily on Cole and Ramachandran's [21] parallel sorting algorithm, with some modifications to account for the fact that sorting alone does not remove waves which are not a part of the upper envelope. Our augmentations preserve many key properties of Cole and Ramachandran's algorithm. Notably, ours is also cache-oblivious and cache optimal (From Arge and Miltersen [7], optimal for non-output sensitive). The cache complexity matches Sharma and Sen's [48] randomized cache oblivious convex hull algorithm but does so deterministically rather than in expected cache misses.

Describe that alg. first!

This algorithm makes use of the Positive Curve Form for representing the upper envelope. Since sometimes a sine wave needs to be split into two (see Section 3), this representation starts with at most $2m$ representations of sine waves where m is the number of actual distinct waves, which means the same asymptotic bounds apply. For ease of use, we will just take $n = 2m$. At the start, we treat these sine waves as singleton lists representing their own upper envelope.

\dots

Data: A, a collection of sinusoidal envelopes, $L_1, L_2, ..., L_r$ where $n \leq 3r^6$
Result: The combined upper envelope of A
if $n \leq 24$ **then** apply any serial upper envelope algorithm and return;
if $n \leq 3r^3$ **then** $k \leftarrow 1, A_1 \leftarrow A$ **else**
  Form a sample S of every r^2th curve (by associated angle) in each L_i, for a total of n_i/r^2 elements from each, where n_i is the size of list L_i;
  Compute ranks of elements of S using associated angle;
  Form a sample P of every $2r$th member in S by rank for a total of $\leq n/2r^3$ elements;
  Using P as a set of pivots, partition A into $k = |P| + 1$ subsets $(A_1, ..A_k)$. Include the curves which their right and left pivot angles pass ~~through~~ (See Figure 6);
end through
parallel foreach *subset* A_i *of* A *(from* 1 *to* k*)* **do**
A_i ? Separate W_i into smaller subsets A_{ij} such that each contains elements from at most $\sqrt{r}$ different lists;
  parallel for each A_{ij} **do** MulitwayMerge(A_{ij}) ;
  Run MulitwayMerge(A_i) using the sorted A_{ij} as lists;
end
parallel foreach *pivot angle* p_i *of* P **do**
  Determine the intersection angle between the last element of A_i and the first element of A_{i+1} and set the A_{i+1} element's starting angle to it.
end

Algorithm 1 Divide-and-Conquer with Multiway Merge

▶ **Theorem 1.** *For n sine waves of the form $x \cos\theta + y \sin\theta$, Algorithm 2 (Multiway Merge) finds their upper envelope in $O(n \log n)$ work and $O(\log n \log\log n)$ span.*

Our algorithm is identical to Cole and Ramachandran's [21] except for two key differences. The first is the size of the subsets W_i (there they use A_i), since ours includes points along the pivot there can be at most $2r$ more elements in our subsets. However, since the size of their subsets is at most $3r^3 - r^2 - r$ (From Lemma 2.1 of [21]) and the bound required for the recursion to hold is subsets of size at most $3r^3$, ours will still work since $3r^3 - r^2 + r \leq 3r^3$.

The other difference is that we have to check the boundaries of each subset to determine the exact angle where the two adjacent waves intersect since that was not determined by the pivot. explain!

We do not need to check any more than the boundaries. All the waves to the right of the pivot must be dominated by some wave to the left of the pivot. The waves which are in the solutions for the subsets to the left of the pivot must dominate or equal those waves.

This check takes $O(\log r) \leq O(\log n)$ span and performs $O(n/r^3) \leq O(\sqrt{n})$ work, which is dominated by the work and span of the rest of the algorithm. Therefore, the work and span bounds of our algorithm are equivalent to theirs [21], $O(n \log n)$ and $O(\log n \log\log n)$, respectively.

1?

▶ **Theorem 2.** *For n points, Algorithm 5 (Multiway Merge) finds the upper envelope of the dual representation with at most $O((n/B) \log_M n)$ cache misses.*

Proof. This follows directly from the Cole and Ramachandran bounds. At each recursion layer, at most $O(n/B)$ cache misses are incurred. All procedures preceding the pivot check at the end are asymptotically identical to Cole and Ramachandran's.

For the end procedure, we access at most s/r memory locations, where s is the size of the sample S. Observation 2.3 of Cole and Ramachandran notes that $s \cdot r = O(n/B)$. Since $s/r < s \cdot r$, then even if every memory access in the end procedure is a cache miss, we will not exceed $O(n/B)$ cache misses at each recursion layer. ◀

\$S\$

4.1 Other paths to $O(\log n \log\log n)$ span

There are several PRAM algorithms which, when analyzed under the binary-forking model, achieve the same span as our modification of Cole and Ramachandran's sorting algorithm. However, their cache complexity is significantly larger.

One was developed by Atallah and Goodrich [9], achieving $O(\log n)$ span with $O(n \log n)$ work, predicated upon an initial sorting step. However, since deterministic sorting in the binary-forking is still $O(\log n \log\log n)$, this algorithm matches our span. The algorithm is based on an $\sqrt{n}$-way merge, using n pairwise common tangent finding procedures to determine which points remain on the hull during the merge. After this identification stage, a prefix sum calculation is used to readjust the array. Using an array format and Overmars and van Leeuwen's [44] tangent finding technique is very inefficient, with each incurring $O(\log(n/B))$ cache misses, or $O(n \log(n/B))$ total. Since n binary searches are done, a more cache efficient structuring of the hulls such as a static B-tree would only improve the misses to $O(n \log_B n)$ cache misses. Also, the added work of maintaining this structure would only increase the cache misses. The same argument can be made for other merge-based PRAM convex hull algorithms that use binary-search based common tangent finding, such as Chen's [20] and Goodrich's CRCW [32] which will also incur $\Omega(n)$ cache misses due to the searches.

Another $O(\log n \log\log n)$ span algorithm, which uses concurrent writes instead of exclusive writes, is Berkman, Schieber and Vishkin's [12] algorithm. It similarly uses a presorting step, but when analyzed under the binary-forking model, achieves $O(\log n \log\log n)$ span with $O(n)$ work after sorting. This is due to its spawning of $O(n)$ threads at each level in a $\log\log n$ height recursion, so spawning threads dominates span. In terms of cache complexity, it confronts similar problems with formatting. In the first step, a $O(\log n)$ span algorithm from Goodrich [32] is called on $n/\log n$ subproblems of size $\log n$. This algorithm also uses a binary search technique which, due to the structure of the data can at most be improved to support $O(\log_B(n))$ searches. The Goodrich algorithm makes $O(\frac{n}{\log n})$ such binary searches, meaning that the first step alone puts Berkman's algorithm at $O(\frac{n \log\log n}{\log B \log n})$ cache misses on top of the initial sort.

the alg of Berkman et al.

5 A Paramaterized Work-Span Tradeoff Algorithm

In this section we present a recursive tradeoff algorithm for finding 2D convex hulls which achieves $O(k \log n)$ span with $O(n^{1+\frac{1}{k}} \log n)$ work, where k is a positive integer. This allows us to achieve $O(\log n)$ span for any constant value of k, so a small work increase is incurred as a result. In addition to the primitives we have described, the algorithm makes use of two subroutines. The first, which we call *DominatingRange*, determines for a given wave and a set of other waves the range for which the given wave dominates the others. The second, which we refer to as just *BaseCase*, uses that subroutine to determine the active ranges of each wave in the upper envelope. We finally bring them all together to create a recursive algorithm that goes a parameterized depth before calling the Base Case and then using the subproblem solutions to decrease the number of waves individual wave needs to be compared ~~against.~~

to against which an

5.1 Determining the dominating range

DominatingRange (Algorithm 3 in Appendix A) takes a particular wave, *wave*, and a set of waves, W, and determines over what range $[0, 2\pi]$ *wave* has a R value that is greater than all those of W. It also determines which wave overtakes it at the end of the range, specifically the wave which intersects it at an endpoint of its range and has a greater slope at that intersection. This will be useful for determining neighbors of waves that are in the upper envelope.

We find this recursively. At the base case, $|W| = 1$ so we compare *wave* to one other wave in W. This can be done in constant time by finding the intersection points and determining which side of the split *wave* is above the other. Above this level, we find the intersection of the two ranges given by the recursive calls using the *CommonRange* primitive.

▶ **Theorem 3.** *For n sine waves of the form $x\cos\theta + y\sin\theta$, Algorithm 4 (Dominating Range) finds the range ~~over~~ in which one sine wave dominates all the others in $\Theta(\log n)$ span and $\Theta(n)$ work.*

Proof.

$$\text{Work, } W(n) = \begin{cases} \Theta(1), & \text{if } n \text{ is 1.} \\ 2W(\frac{n}{2}) + \Theta(1), & \text{otherwise.} \end{cases}$$

$$\text{Span, } S(n) = \begin{cases} \Theta(1), & \text{if } n \text{ is 1.} \\ S(\frac{n}{2}) + \Theta(1), & \text{otherwise.} \end{cases}$$

This follows from the fact that each primitive function performs $O(1)$ work. Using the Master~~'s~~ theorem, we get $\Theta(n)$ work and $\Theta(\log n)$ span.

◀

5.2 Base case of the algorithm

Our base case algorithm essentially applies *DominatingRange* to each individual wave for the entire set of waves. It does this in parallel for all waves in W, which amounts to n calls of *DominatingRange* running alongside each other. At this point, we have a linked list representation of the upper envelope. We can transfer these elements in parallel to an array by determining their order and then placing them into the array in parallel. We determine the rank using a list ranking algorithm, with the one modification being that the ranks of waves with empty ranges are set to n to start to avoid any possibility of accidentally being added into the array.

For both the base case algorithm and the main algorithm, we use Blelloch et al.'s [15] adaptation of Wyllie's list ranking algorithm, which performs $O(n \log n)$ work in $O(\log n)$ span. Since this subroutine uses Test-and-Set (TS), this algorithm as a whole does not apply to the binary-fork join model.

▶ **Theorem 4.** *For n sine waves of the form $x\cos\theta + y\sin\theta$, Algorithm 5 (the Base Case of Angular Elimination) finds the upper envelope in $\Theta(\log n)$ span and $\Theta(n^2)$ work.*

Proof. Since *DominatingRange* performs $\Theta(n)$ work and this is performed once for every wave in W, the total work must be $\Theta(n^2)$, which dominates the $O(n \log n)$ list ranking and $O(n)$ array assignments.

Since *DominatingRange*, list ranking, and spawning in threads all have $\Theta(\log n)$ span, it follows that the entire algorithm must have $\Theta(\log n)$ span in total. ◀

5.3 Recursive structure

The final algorithm combines *DominatingRange* and the base case to form a parameterized recursive algorithm which allows a trade-off between optimal span for $k = O(1)$ and optimal work for $k = \log n$.

The structure of its recursion is very similar to Goodrich and Ghouse's [29] "inductive" convex hull algorithm which performs $O(k)$ span using $n^{1+\frac{2}{k}}$ processors in CRCW PRAM model. We split the waves into groups of size $O(n^{\frac{1}{k}}$, apply our algorithm on these subgroups, and decrease the k value by 1 for these calls. If k is 1 or less, then we use the base case algorithm.

After finding the envelopes of the subproblems, we apply *DominatingRange* to determine the range for which each wave is part of the upper envelope. From the subproblem solutions, each wave now has some active range or has already been eliminated. For each subproblem envelope, the wave has some subsection that it may overlap with. We find this subsection with binary search and then apply *DominatingRange* to only the set of waves in that subsection. We do this from every wave to every subproblem envelope and collect the resulting ranges from each of these in an array. We then find the intersections of all these ranges to get the final range for which each wave is dominant. Just as with the base case, we apply list ranking to return the upper envelope in array form.

▶ **Theorem 5.** *For n sine waves of the form $x \cos\theta + y \sin\theta$, Algorithm 6 (Angular Elimination) finds the upper envelope in $\Theta(k \log n)$ span and $O(n^{1+\frac{1}{k}} \log n)$ work, where $k \in [1, \log n]$ is an integer.*

Proof. Let $W(n,k)$ be the work and $S(n,k)$ be the span of the algorithm for an input size n and parameter value k. Then

$$W(n,k) = \begin{cases} \Theta(n^2), & \text{if } k \leq 1. \\ n^{\frac{1}{k}} W(n^{\frac{k-1}{k}}, k-1) + \Theta(n^{1+\frac{1}{k}} \log n), & \text{otherwise.} \end{cases}$$

$$S(n,k) = \begin{cases} \Theta(\log n), & \text{if } k \leq 1. \\ S(n^{\frac{k-1}{k}}, k-1) + \Theta(\log n), & \text{otherwise.} \end{cases}$$

The recurrences follow since for $k > 1$, the work and span are equal to that of the division into subproblems plus the merging of those subproblems. The subproblem work and span comes from there being $n^{\frac{1}{k}}$ subproblems of size $n^{\frac{k-1}{k}}$.

For the merging, first, we consider the work. We iterate over the set of all waves. Each of these then compares themself against $n^{1/k}$ subproblem solutions. Finding the leftmost and rightmost curves with the overlapping range can be done in $\Theta(\log n)$ time using binary search. This part takes $\Theta(n^{1+\frac{1}{k}} \log n)$ work.

Applying *DominatingRange* to the range between can have worst-case $O(n^{k-1}k)$ work, but this would require that there are $O(n^{k-1}k)$ curves that overlap with the wave we are analyzing. If that is the case, then only 2 of those curves can overlap with the ranges of the other waves in the same subproblem as the wave we are analyzing. which means we can amortize this work. For the $n^{1/k}$ subproblem solutions, we perform $(n^{1/k} - 1) \cdot \Theta(n^{\frac{k-1}{k}})$ work, which multiplies out to be $\Theta(n^{1+\frac{1}{k}})$ work, which is dominated by the previous work.

For span, all operations take $\Theta(\log n)$ time including spawning threads, so the contribution of the merge is $\Theta(\log n)$.

The recurrences can be shown to be $S(n,k) = O(k \log n)$, and $W(n,k) = O(n^{1+\frac{1}{k}} \log n)$ through induction on k. Evaluating at $k = 1$, we have $S(n,1) = \Theta(\log n) = \Theta(1 \cdot \log n)$ and $W(n,1) = O(n^2) \in O(n^{1+1/1} \log n)$. Assuming that for some $k_0 \geq 1$ the bounds hold, we can show that the bounds hold for $k_0 + 1$.

$$\begin{aligned}
S(n, k_0+1) &= S(n^{\frac{k_0-1}{k}}, k_0) + \Theta(\log n) \\
&= \Theta(k_0 \log(n^{\frac{k_0-1}{k_0}}) + \Theta(\log n) \\
&\leq c_1(k_0-1)(\log(n)) + c_2 \log(n) \\
&\leq (k_0+1)\log n, \ \text{ for } c_1 = 1, c_2 = 2
\end{aligned}$$

And the same applies for work:

$$\begin{aligned}
W(n, k_0+1) &= n^{\frac{1}{k_0+1}} W(n^{\frac{k_0}{k_0+1}}, k_0) + \Theta(n^{1+\frac{1}{k_0+1}} \log n) \\
&= O\left(n^{\frac{1}{k_0+1}} \cdot n^{\left(\frac{k_0+1}{k_0}\right)\left(\frac{k_0}{k_0+1}\right)} \log\left(n^{\frac{k_0}{k_0+1}}\right)\right) \\
&+ \Theta(n^{1+\frac{1}{k_0+1}} \log n) \\
&\leq c_1 \frac{k_0}{k_0+1} n^{1+\frac{1}{k_0+1}} \log n + c_2 n^{1+\frac{1}{k_0+1}} \log n \\
&\leq n^{1+\frac{1}{k_0+1}} \log n, \ \text{ for } c_1 = 0.5, c_2 = 0.5
\end{aligned}$$

Therefore, Angular Elimination performs $O\left(n^{1+\frac{1}{k}} \log n\right)$ work with $O(k \log n)$ span. ◀

Since we used induction on k starting with 1 as our base case, k is restricted to positive integers. Additionally, for any $k > \log n$, we incur more span with no improvements in work, which at $k = \log n$ becomes $O(n \log n)$. Thus we ~~define~~ restrict the parameter k to be a positive integer in the range $[1, \log n]$.

w.h.p.

6 A Randomized Algorithm with Optimal Work and Span WHP in n

We also present an adaptation of Ahmad et al.'s [3] randomized sorting algorithm which matches its bounds, $O(\log n)$ span and $O(n \log n)$ work, both *whp* in n. It follows mostly the same format as the sorting algorithm. It is comprised of an Almost-Sort algorithm that returns a sorted list of some of the elements and a Full-Sort algorithm that calls on the Almost-Sort and then reincorporates the elements not included in its output. The Almost-Sort algorithm uses a bucket-based approach, first by sorting a sample of $\sqrt{n}\log^3 n$ elements to form pivots for the buckets and then attempting to place elements into the buckets. If two or more elements try to write to the same slot in the bucket, this is considered a collision and only one of them is written while the others are set aside for processing later. This bucketing is done recursively until they are of size $n^{1/\log\log n}$ so that Cole and Ramanchandran's [21] sorting algorithm can be run on them in $O(\log n)$ span. After running the Almost-Sort, the Full-Sort algorithm reincorporates all the elements that were set aside and due to a high- probability guarantee on the number of these leftover elements, reincorporation dominates neither span n or work.

Our approach for convex hulls differs in a few key ways:

1) We use convex hull algorithms at every level. We replace their n^ϵ-way sort with angular elimination (described in Section 5). To match the workbound, we use angular elimination with $k = 2$, giving us $O(n^{4/3} \log n)$ instead of $O(n^{3/2})$ work. We also replace the Cole and

So the randomized alg. also does not adhere to the binary join-fork model ?!

Ramachandran sorting algorithm with our adaptation, MultiwayMerge (described in Section 4).

2) We bucket by angle, but when recursing over the said bucket, we find the envelope across the entire domain instead of just between bucket boundaries. For example, if we put points into the bucket for $[\pi/3, 3\pi/4]$, then evaluating AlmostHull (our adaptation of Almost-Sort) on the next layer will still evaluate over the whole period $[0, 2\pi)$. Only once all comparisons are made in the merge do we delete points that do not dominate in the bucket range. Additionally, when we place points into buckets, we are comparing with an envelope of a sample, so if the point's sinusoidal wave is completely dominated, we just discard it altogether.

3) We have to add in checks across buckets to make sure that points are constrained to those domains. In AlmostHull, this entails doing another run of the algorithm on conflicts, points that were placed in other buckets but after solving within the bucket they stretch into another buckets domain. These conflicts are placed in another array, and then the version in the conflict array is merged into the originals array. This merge can be done the same way that the pairwise merges are done in angular elimination, but since there are only two envelopes to compare for each bucket, it will be total $O(n)$ work. In FullHull (our adaptation of Full-Sort), this entails checking across the bucket boundaries after using the MultiwayMerge, exactly as is done in that algorithm.

Otherwise, the algorithms are identical, and therefore the exact same probabilistic guarantees hold. In terms of collisions, the additional invocation of AlmostHull adds at most a constant factor. In fact, the collisions will in practice be likely significantly less as we will remove points from consideration if they are dominated by the envelope formed by the sample used for bucketing.

7 Conclusion

Boring — no open questions!

We have presented three parallel algorithms for the binary-forking model which can be used to find the convex hull of a set of points in 2D. All three use Hough transforms to find the upper envelope of a set of sine waves corresponding to the input points, or can be converted to use point-line duality instead. We present the first deterministic cache-oblivious CREW algorithm that finds the convex hull of a set of 2D points cache and work-optimally while matching the best known span bound for any deterministic convex hull algorithm for the binary-forking model. We also present a parameterized work-span tradeoff algorithm that allows us to achieve $O(k \log n)$ span at the cost of $O\left(n^{1+\frac{1}{k}} \log n\right)$ work for any integer $k \in [1, \log n]$. Finally, we present a randomized algorithm which achieves $O(\log n)$ span and $O(n \log n)$ work, both with high probability in n.

References

1 Umut A Acar, Guy E Blelloch, and Robert D Blumofe. The data locality of work stealing. In *Proceedings of the twelfth annual ACM symposium on Parallel algorithms and architectures*, pages 1–12, 2000.

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 *Proceedings of the ACM Symposium on Parallelism in Algorithms and Architectures*, pages 84–95, 2014.

3 Zafar Ahmad, Rezaul Chowdhury, Rathish Das, Pramod Ganapathi, Aaron Gregory, and Mohammad Mahdi Javanmard. Low-span parallel algorithms for the binary-forking model.

In *Proceedings of the 33rd ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '21, page 22–34, ~~New York, NY, USA~~, 2021. ~~Association for Computing Machinery~~. doi:10.1145/3409964.3461802.

4 Selim G Akl and Godfried T Toussaint. Efficient convex hull algorithms for pattern recognition applications. In *Proceedings of the International Conference on Pattern Recognition, 4th*, pages 483–487, 1979.

5 Nancy M Amato, Michael T Goodrich, and Edgar A Ramos. Parallel algorithms for higher-dimensional convex hulls. In *Proceedings 35th Annual Symposium on Foundations of Computer Science*, pages 683–694. IEEE, 1994.

6 A.M. Andrew. Another efficient algorithm for convex hulls in two dimensions. *Information Processing Letters*, 9(5):216–219, 1979. ~~URL: https://www.sciencedirect.com/science/article/pii/0020019079900723~~, doi:~~https://doi.org/~~10.1016/0020-0190(79)90072-3.

7 Lars Arge and Peter Bro Miltersen. *On Showing Lower Bounds for External-Memory Computational Geometry Problems*, page 139–159. American Mathematical Society, USA, 1999. Where ?

8 Mikhail J Atallah, Richard Cole, and Michael T Goodrich. Cascading divide-and-conquer: A technique for designing parallel algorithms. *SIAM Journal on Computing*, 18(3):499–532, 1989.

9 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. ~~URL: https://www.sciencedirect.com/science/article/pii/0743731586900110~~, doi:~~https://doi.org/~~10.1016/0743-7315(86)90011-0.

10 C Bradford Barber, David P Dobkin, and Hannu Huhdanpaa. The quickhull algorithm for convex hulls. *ACM Transactions on Mathematical Software (TOMS)*, 22(4):469–483, 1996.

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 *Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures*, pages 145–156, 2016.

12 Omer Berkman, Baruch Schieber, and Uzi Vishkin. A fast parallel algorithm for finding the convex hull of a sorted point set. *International Journal of Computational Geometry & Applications*, 6(02):231–241, 1996.

13 Guy E Blelloch, Rezaul Chowdhury, Phillip B Gibbons, Vijaya Ramachandran, Shimin Chen, and Michael Kozuch. Provably good multicore cache performance for divide-and-conquer algorithms. In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms*, pages 501–510, 2008.

14 Guy E Blelloch, Jeremy T Fineman, Phillip B Gibbons, and Harsha Vardhan Simhadri. Scheduling irregular parallel computations on hierarchical caches. In *Proceedings of the twenty-third annual ACM symposium on Parallelism in algorithms and architectures*, pages 355–366, 2011.

15 Guy E Blelloch, Jeremy T Fineman, Yan Gu, and Yihan Sun. Optimal parallel algorithms in the binary-forking model. In *Proceedings of the ACM Symposium on Parallelism in Algorithms and Architectures*, pages 89–102, 2020.

16 Guy E Blelloch and Phillip B Gibbons. Effectively sharing a cache among threads. In *Proceedings of the ACM Symposium on Parallelism in Algorithms and Architectures*, pages 235–244, 2004.

17 Guy E Blelloch, Yan Gu, Julian Shun, and Yihan 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.

18 Robert D Blumofe and Charles E Leiserson. Space-efficient scheduling of multithreaded computations. *SIAM Journal on Computing*, 27(1):202–229, 1998.

19 T. M. Chan. Optimal output-sensitive convex hull algorithms in two and three dimensions. *Discrete Comput. Geom.*, 16(4):361–368, ~~apr~~ 1996. doi:10.1007/BF02712873.

20 Danny Z. Chen. Efficient geometric algorithms on the erew pram. *IEEE transactions on parallel and distributed systems*, 6(1):41–47, 1995.

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

22 James W Cooley and John W Tukey. An algorithm for the machine calculation of complex Fourier series. *Mathematics of Computation*, 19(90):297–301, 1965.

23 Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press, 2009.

24 Eric Dumonteil, Satya N Majumdar, Alberto Rosso, and Andrea Zoia. Spatial extent of an outbreak in animal epidemics. *Proceedings of the National Academy of Sciences*, 110(11):4239–4244, 2013.

25 Martin Farach-Colton, Meng Li, and Meng-Tsung Tsai. Streaming algorithms for planar convex hulls, 2018. `arXiv:1810.00455`.

26 `http://docs.oracle.com/javase/tutorial/essential/concurrency/forkjoin.html`, (Oracle Java Documentation).

27 Matteo Frigo, Charles E. Leiserson, Harald Prokop, and Sridhar Ramachandran. Cache-oblivious algorithms. *ACM Trans. Algorithms*, 8(1), ~~jan~~ 2012. `doi:10.1145/2071379.2071383`.

28 Matteo Frigo, Charles E Leiserson, and Keith H Randall. The implementation of the cilk-5 multithreaded language. In *Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*, pages 212–223, 1998.

29 Mujtaba Ghouse and Michael T. Goodrich. Fast randomized parallel methods for planar convex hull construction, 1991. Where ?

30 Mujtaba R Ghouse and Michael T Goodrich. In-place techniques for parallel convex hull algorithms ~~(preliminary version)~~. In *Proceedings of the third annual ACM symposium on Parallel algorithms and architectures*, pages 192–203, 1991.

31 Mujtaba R Ghouse and Michael T Goodrich. Fast randomized parallel methods for planar convex hull construction. *Computational Geometry*, 7(4):219–235, 1997.

32 Michael T. Goodrich. Finding the convex hull of a sorted point set in parallel. *Information Processing Letters*, 26(4):173–179, 1987. ~~URL: https://www.sciencedirect.com/science/article/pii/0020019087900020,~~ doi:~~https://doi.org/~~10.1016/0020-0190(87)90002-0.

33 Michael T Goodrich, Riko Jacob, and Nodari Sitchinava. Atomic power in forks: A super-logarithmic lower bound for implementing butterfly networks in the nonatomic binary fork-join model. In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms*, pages 2141–2153. SIAM, 2021.

34 M.T. Goodrich, Jyh-Jong Tsay, D.E. Vengroff, and J.S. Vitter. External-memory computational geometry. In *Proceedings of 1993 IEEE 34th Annual Foundations of Computer Science*, pages 714–723, 1993. `doi:10.1109/SFCS.1993.366816`.

35 Ronald L. Graham. An efficient algorithm for determining the convex hull of a finite planar set. *Inf. Process. Lett.*, 1:132–133, 1972.

36 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, 2003.

37 J. JaJa. *An Introduction to Parallel Algorithms*. Addison Wesley, 1997.

38 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. ~~URL: https://www.sciencedirect.com/science/article/pii/0020019073900203~~, doi:~~https://doi.org/~~10.1016/0020-0190(73)90020-3.

39 Michael Kallay. The complexity of incremental convex hull algorithms in rd. *Information Processing Letters*, 19(4):197, 1984.

40 David G Kirkpatrick and Raimund Seidel. The ultimate planar convex hull algorithm? *SIAM journal on computing*, 15(1):287–299, 1986.

41 Tim Mercy, Wannes Van Loock, and Goele Pipeleers. Real-time motion planning in the presence of moving obstacles. In *2016 European Control Conference (ECC)*, pages 1586–1591, Aalborg, Denmark, 2016. IEEE.

42 Russ Miller and Quentin F. Stout. Efficient parallel convex hull algorithms. *IEEE transactions on Computers*, 37(12):1605–1618, 1988.

43 K Murakami, H Koshimizu, and K Hasegawa. An algorithm to extract convex hull on theta-rho H hough transform space. In *9th International Conference on Pattern Recognition*, pages 500–501, Los Alamitos, CA, USA, 1988. IEEE Computer Society.

44 Mark H. Overmars and Jan van Leeuwen. Maintenance of configurations in the plane. *Journal of Computer and System Sciences*, 23(2):166–204, 1981. ~~URL: https://www.sciencedirect.com/science/article/pii/002200008190012X,~~ doi:~~https://doi.org/~~10.1016/0022-0000(81)90012-X.

45 F. P. Preparata1977 and S. J. Hong. Convex hulls of finite sets of points in two and three dimensions. *Commun. ACM*, 20(2):87–93, feb 1977. doi:10.1145/359423.359430.

46 Vijaya Ramachandran and Elaine Shi. Data oblivious algorithms for multicores. In *Proceedings of the 33rd ACM Symposium on Parallelism in Algorithms and Architectures*, pages 373–384, New York, NY, USA, 2021. Association for Computing Machinery.

47 John H Reif and Sandeep Sen. Optimal randomized parallel algorithms for computational geometry. *Algorithmica*, 7(1):91–117, 1992.

48 Neeraj Sharma and Sandeep Sen. Efficient cache oblivious algorithms for randomized divide-and-conquer on the multicore model, 2012. URL: https://arxiv.org/abs/1204.6508, ~~doi:10.48550/ARXIV.1204.6508.~~

49 https://msdn.microsoft.com/en-us/library/dd460717, (TPL).

50 https://www.threadingbuildingblocks.org, (TBB).

51 Mark Wright, Andrew Fitzgibbon, Peter Giblin, and Robert Fisher. Convex hulls, occluding contours, aspect graphs and the hough transform. *Image Vision Comput.*, 14:627–634, 01 1996.

52 Chang-Wu Yu and Gen-Huey Chen. Efficient parallel algorithms for doubly convex-bipartite graphs. *Theoretical Computer Science*, 147(1-2):249–265, 1995.

A Algorithm Pseudocodes

We present pseudocode for the algorithms described above.

B Primitives

We have several primitive functions which provide the basis for many of our algorithms presented. In the interest of implementation and completeness, we also provide pseudocode for them here.

Data: A, a collection of sinusoidal envelopes, $L_1, L_2, ..., L_r$ where $n \leq 3r^6$
Result: The combined upper envelope of A
if $n \leq 24$ **then** apply any serial upper envelope algorithm and return;
if $n \leq 3r^3$ **then** $k \leftarrow 1, A_1 \leftarrow A$ **else**
  Form a sample S of every r^2th curve (by associated angle) in each L_i, for a total of n_i/r^2 elements from each, where n_i is the size of list L_i;
  Compute ranks of elements of S using associated angle;
  Form a sample P of every $2r$th member in S by rank for a total of $\leq n/2r^3$ elements;
  Using P as a set of pivots, partition A into $k = |P| + 1$ subsets $(A_1, ...A_k)$. Include the curves which their right and left pivot angles pass through (See Figure 6);
end
parallel foreach *subset* A_i *of* A *(from* 1 *to* k*)* **do**
  Separate W_i into smaller subsets A_{ij} such that each contains elements from at most $\sqrt{r}$ different lists;
  parallel for each A_{ij} **do** MulitwayMerge(A_{ij}) ;
  Run MulitwayMerge(A_i) using the sorted A_{ij} as lists;
end
parallel foreach *pivot angle* p_i *of* P **do**
  Determine the intersection angle between the last element of A_i and the first element of A_{i+1} and set the A_{i+1} element's starting angle to it.
end

Algorithm 2 Divide-and-Conquer with Multiway Merge

Data: W, a set of n sine waves of $x\cos(\theta) + y\sin(\theta)$, *wave*, another sine wave of the same form
Result: The range over which *wave* dominates W and the wave that dominates it to the right
if *W.size = 1* **then**
  $I \leftarrow Intersections(wave, W[0])$;
  $S.next \leftarrow W[0]$;
  if wave $\neq$ MaxToRight(I[0]) **then** $I \leftarrow [I[1], I[0]]$;
  $S.range \leftarrow I$;
  return S;
end
$W_1 \leftarrow [W[0]...W[n/2-1]]$;
$W_2 \leftarrow [W[n/2]...W[n-1]]$;
spawn;
if new thread **then** $S_1 \leftarrow DominatingRange(W_1, n/2, wave)$;
else $S_2 \leftarrow DominatingRange(W_2, n/2, wave)$;
$S.range \leftarrow CommonRange(S_1.range, S_2.range)$;
$S.next \leftarrow MaxVal(S_1.next, S_2.next, S.range[1])$;
return S;

Algorithm 3 Dominating Range

Data: W, a set of n sine waves of $x\cos(\theta) + y\sin(\theta)$
Result: The upper envelope of W
parallel foreach *wave in W* **do**
    $IntersectInfo = DominatingRange(W, n, wave)$;
    $wave.range = IntersectInfo.range$;
    $wave.next = IntersectInfo.range$;
    if wave.range is empty **then** $wave.rank \leftarrow n$;
    else $wave.rank \leftarrow 0$;
end
ListRanking(W);
E = array of size n;
parallel foreach *wave in W* **do**
    if wave.rank < n **then** E[wave.rank] = wave;
end
return E;

Algorithm 4 Base Case for Angular Elimination

Data: W, a set of n sine waves of the form $x\cos(\theta) + y\sin(\theta)$, a parameter k
Result: The upper envelope of W
if $k \leq 1$ **then** **return** BaseCase(S,n);
parallel foreach *subset S of size* $n^{(k-1)/k}$ **do**
    $S \leftarrow AngularElimination(S, n^{(k-1)/k}, k-1)$;
end
parallel foreach *wave in W* **do**
    $P \leftarrow$ array of size $n^{\frac{1}{k}}$;
    parallel foreach *envelope S in P* **do**
        $L \leftarrow$ the curve in S containing wave.range[0];
        $R \leftarrow$ the curve in S containing wave.range[1];
        $W_i \leftarrow$ the curves of S between L and R;
        $P[i] \leftarrow DominatingRange(W_i, n^{\frac{k-1}{k}}, wave)$;
    end
    $wave.range \leftarrow$ *CommonRange* among all P[i].range;
    $wave.next \leftarrow$ *MaxToRight* among all P[i].next;
    if wave.range is empty **then** $wave.rank \leftarrow n$;
    else $wave.rank \leftarrow 0$;
end
$ListRanking(W)$;
$E \leftarrow$array of size n;
parallel foreach *wave in W* **do**
    if wave.rank < n **then** $E[wave.rank] \leftarrow wave$;
end
return E;

Algorithm 5 Angular Elimination

Data: B, a set of sine waves of the form $x \cos\theta + y \sin\theta$,
n_c : size of array to sort (only $B[l_0, ..., l_0 + nc - 1]$ is occupied)
n : size of the array at the highest level of recursion
l_0 : location where the array to sort begins
d : depth of current call in recursion tree
m : multiple of extra memory to use
$B[l_0, ..., l0 + n_c m - 1]$: contains array to be sorted
$C[l_0, ..., l0 + n_c m - 1]$: ancillary space
$C_2[l_0, ..., l0 + n_c m - 1]$: secondary ancillary space
$D[l_0, ..., l0 + n_c m - 1]$: where prefix sums will be stored (for indexing)
Result: The upper envelope of W

```
if d ≥ log log log n then return MultiwayMerge(B[l0, l0 + nc − 1];
s ← log^3 nc;
P ← sample with repetition of size (√nc + 1)s from B[l0, l0 + nc − 1];
P ← AngularElimination(P, 3);
P[0] ← 0, P[(√nc + 1)s] ← 2π;
Space out the elements of P as evenly as possible. Fill the gaps with duplicates of the points
  but with split ranges;
{ Bucketing }
parallel foreach a ∈ B[l0, l0 + nc − 1] do
    Find some i where a begins dominating the pivot envelope between P[i · s] and
      P[(i + 1) · s if none exists do nothing; Choose a random number j ∈ [0, ...m√nc − 1;
    Attempt C[im√nc + j] ← a, doing nothing if collision
end
parallel for i ← l0 to l0 + nc − 1 do B[i] ← null ;
{Compacting}
parallel foreach i ∈ [0, nc − 1 do
    low ← l0 + im√nc;
    hi ← l0 + (i + 1)m√nc − 1;
    D[low, hi] ← Indicator-Prefix-Sum(C[low, hi);
    parallel foreach j ∈ [low, ...hi] do
        D[j] = D[j] + im√nc;
    end
end
parallel foreach i ∈ [l0, l0 + nc m − 1 do
    if C[i] is not null then
        B[D[i]] ← C[i];
        C[i] ← null;
    end
end
{Solving Buckets and Comparing }
parallel foreach i ∈ [0, ...√nc − 1] do
    AlmostHull(√nc, n, l0 + im√nc, d + 1, m, B, C, D);
end
parallel foreach i ∈ [0, ...√nc − 1] do
    parallel foreach j ∈ [0, ...√nc − 1] do
        if i = j then do nothing;
        Determine if the ith bucket envelope dominates the jth bucket within its bucket. If
          so, randomly assign it to a corresponding slot in C2 (just as before, do nothing if
          already occupied).
    end
end
parallel foreach i ∈ [0, ...√nc − 1] do
    Use AlmostHull on the bucket conflicts in C2, then merge the conflict bucket with the
      original bucket.
end
C ← B; D ← Indicator-Prefix-Sum(C);
parallel for i ∈ [0, ...nc m − 1] do if C[i] is not null then B[D[i]] ← C[i]
```

Algorithm 6 AlmostHull

```
Data: A, a set of sine waves of the form x cos θ + y sin θ
Result: The upper envelope of W
m ← log n log log log n/ log log n; Allocate arrays B, C, C_2, D of size nm; B[0, ..., n − 1] ← A;
Almost − Hull(n, n, 0, 0, m, B, C, C_2, D);
num_sorted ← smallest i s.t. B[i] = null;
Set all elements of C and D to null;
Allocate arrays E, F of size n ;
parallel foreach a ∈ A[0, ..., n − 1] do
    Find smallest i s.t. B[i] ≤ a < B[i + 1] (comparing start angle);
    if a ≠ B[i] then
        Choose a random number j ∈ [0, ..., m − 1];
        Attempt C[i · m + j] ← a; do nothing if collision;
    end
end
D ← Indicator-Prefix-Sum(C);
parallel foreach i ∈ [0, ...n] do
    if D[i · block_size] ≠ D[(i + 1) · block_size − 1 then E[i] = 1l
end
F ← Prefix − Sum(E);
Set all elements of C and D to null;
block_size ← log^3 nlog^2 loglogn/log^3 log n;
parallel foreach a ∈ A[0, ..., n − 1] do
    Find smallest i s.t. B[i] ≤ a < B[i + 1], (comparing start angle);
    if a ≠ B[i] and E[i] = 1 then
        Choose a random number j ∈ [0, ..., block_size − 1];
        Attempt C[(F[i] − 1) · block_size + j] ← a; do nothing if collision;
    end
end
D ← Indicator-Prefix-Sum(C);
parallel foreach i ∈ [0, ...num_sorted − 1 do
    if E[i] = 0 then E[i] = 1;
    else E[i] = D[F[i] · block_size − 1] − D[F[i]]
end
Set all elements of C and D to null;
parallel foreach i ∈ [0, ...n − 1] do
    parallel foreach j ∈ [0, log n − 1 do
        Choose a random number k = H(j, (F[i] − E[i])m, F[i]m − 1);
        Attempt C[k] ← a; do nothing if collision;
    end
    chunk_size ← n log log log n/ log log n;
    foreach i ∈ [0, ...n/chunk_size − 1 do
        parallel foreach a ∈ A[i · chunk_size, ..., (i + 1) · chunk_size − 1 do
            Find smallest i s.t B[i] ≤ a < B[i + 1] (comparing start angle);
            Keep-Single(C,a,n,H,(F[i]-E[i])m, F[i]m -1)
        end
    end
end
parallel foreach i ∈ 0, ...num_sorted − 1 do
    D[(F[i] − E[i])m, ...F[i]m − 1] ← Indicator-Prefix-Sum(C[(F[i]-E[i])m,..., F[i]m -1]);
    parallel foreach j ∈ [(F[i] − E[i])m, F[i]m − 1] do
        D[j] ← D[j] + (F[i] − E[i])m;
    end
end
parallel foreach i ∈ [0, ...nm − 1] do
    if C[i] ≠ null then {A[D[i]] ← C[i]; C[i] ← null}
end
parallel foreach i ∈ [0, ...num_sorted − 1] do
    lo ← (F[i] − E[i])m; hi ← lo +E[i];
    A[lo,.., hi] ← Multiway-Merge(A[lo,.., hi]);
end
parallel foreach i ∈ [0, ...num_sorted − 1] do
    lo ← (F[i] − E[i])m; hi ← lo +E[i];
    Check across boundaries of lo and hi to adjust start positions.
end
D ← Indicator-Prefix-Sum(A);
parallel for i ∈ [0, ...nm − 1] do if A[i] ≠ null then C[D[i]] ← A[i]; A[i] ← null;;
A ← C;
```

Algorithm 7 Full-Hull

Data: w_1 and w_2, two sine waves and θ an angle $[0, 2\pi]$
Result: The wave with the larger value at the angle
function *MaxVal*(w_1, w_2, θ)**:**
  $r_1 \leftarrow w_1[x] \cos\theta + w_1[y] \sin\theta$;
  $r_2 \leftarrow w_2[x] \cos\theta + w_2[y] \sin\theta$;
  if $r_1 > r_2$ **then return** w_1;
  else return w_2;

Algorithm 8 MaxVal (For MinVal replace ">" with "<")

Data: w_1 and w_2, two sine waves and θ an angle $[0, 2\pi]$
Result: The wave with the larger slope at the angle
function *MaxSlope*(w_1, w_2, θ)**:**
  $r_1 \leftarrow -w_1[x] \sin\theta + w_1[y] \cos\theta$;
  $r_2 \leftarrow -w_2[x] \sin\theta + w_2[y] \cos\theta$;
  if $r_1 > r_2$ **then return** w_1;
  else return w_2;

Algorithm 9 MaxSlope (For MinSlope replace ">" with "<")

Data: w_1 and w_2, two sine waves and θ an angle $[0, 2\pi]$
Result: The wave with the larger value at the angle and rightward
function *MaxToRight*(w_1, w_2, θ)**:**
  $r_1 \leftarrow w_1[x] \cos\theta + w_1[y] \sin\theta$;
  $r_2 \leftarrow w_2[x] \cos\theta + w_2[y] \sin\theta$;
  if $r_1 > r_2$ **then return** w_1;
  if $r_1 = r_2$ **then return** MaxSlope(w_1, w_2, θ);
  else return w_2;

Algorithm 10 MaxToRight (For MaxToLeft replace MaxSlope with MinSlope)

Data: w_1 and w_2, two sine waves
Result: The angles $[0, 2\pi)$ at which they intersect
function *Intersections*(w_1, w_2)**:**
  $\theta \leftarrow \tan^{-1}(\frac{w_1[x]-w_2[x]}{w_2[y]-w_1[y]})$;
  if $\theta < 0$ **then** $\theta \leftarrow \theta + \pi$; **return** $[\theta, \theta + \pi]$;

Algorithm 11 Intersections

```
Data: I_1 and I_2, two intervals [0, 2π)
Result: The wave with the larger value at the angle
function CommonRange(I_1, I_2):
    if I_1[0] < I_1[1] then
        if I_2[0] < I_2[1] then
            I ← [max(I_1[0], I_2[0]), min(I_1[1], I_2[1])];
            if I[0] > I[1] then  return [0,0];
            elsereturn I;
        end
        else
            I'_2 ← [0, I_2[1] − I_2[0] + 2π];
            I'_1 ← [I_1[0] − I_2[0] + 2π, I_1[1] − I_2[0] + 2π];
            I' ← CommonRange(I'_1, I'_2);
            return [I_2[0] + I'[0]  (mod 2π), I_2[0] + I'[1]  (mod 2π)];
        end
    end
    else
        if I_2[0] < I_2[1] then return CommonRange(I_2, I_1);
        else
            I'_2 ← [0, I_2[1] − I_2[0] + 2π];
            I'_1 ← [I_1[0] − I_2[0] + 2π, I_1[1] − I_2[0] + 2π];
            I ← CommonRange(I'_2, I'_1);
            return [I_2[0] + I'[0]  (mod 2π), I_2[0] + I'[1]  (mod 2π)];
        end
    end
```

Algorithm 12 CommonRange