Cache-Oblivious Parallel Convex Hull in the Binary Forking Model

Reilly Browne * Rezaul Chowdhury † Shih-Yu Tsai ‡ Yimin Zhu §

May 23, 2023

Abstract

We present two cache-oblivious sorting-based convex hull algorithms in the Binary Forking Model. The first is an algorithm for a presorted set of points which achieves $O(n)$ work, $O(\log n)$ span, and $O(n/B)$ serial cache complexity, where B is the cache line size. These are all optimal worst-case bounds for cache-oblivious algorithms in the Binary Forking Model. The second adapts Cole and Ramachandran's cache-oblivious sorting algorithm, matching its properties including achieving $O(n \log n)$ work, $O(\log n \log\log n)$ span, and $O(n/B \log_M n)$ serial cache complexity. Here M is the size of the private cache.

1 Introduction

Finding the convex hull of a set of n points in the plane 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 [3, 24, 35]. In the serial setting, there have been several efficient algorithms for constructing convex hulls [20, 25, 32, 38], both in 2-dimensions and in the more general problem of d dimensions. In the parallel setting, a bunch of efficient convex hull algorithms have been developed for a set of d-dimensional points [4, 7, 8, 18, 33, 36, 39]. There are also many parallel convex hull algorithms specifically for presorted or unsorted points in 2 dimensions [11, 21, 29, 36].

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*** [14]. In particular, we present ***cache-oblivious*** [27] ***parallel*** algorithms for both presorted (by x-coordinate) and unsorted points with optimal worst-case ***work*** (see next paragraph) with respect to the size of the input. We achieve optimal ***span*** (see next paragraph) in the presorted case and match the best-known span for sorting in the unsorted case for the binary-forking model.

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 [1, 10, 12–14] realistically captures the performance of parallel algorithms designed for modern multicore shared-memory machines. Its formal definition is in [14]. 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 [1, 2, 12, 15, 19, 23] are

*Department of Computer Science, Stony Brook University, rjbrowne@cs.stonybrook.edu
†Department of Computer Science, Stony Brook University, rezaul@cs.stonybrook.edu
‡Department of Computer Science, Stony Brook University, shitsai@cs.stonybrook.edu
§Department of Computer Science, Stony Brook University, Yimin.Zhu@stonybrook.edu

widely used in many parallel programming languages or environments such as Intel TBB [43], the Microsoft Task Parallel Library [42], Cilk [28], and the Java fork-join framework [26].

The binary-forking model is an ideal candidate for modeling parallel computations on modern architectures when compared with the closely related PRAM model [34]. 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 [14]. 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 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.

There are several ways to expand the cache-oblivious model to parallel computation. The performance of our algorithms are analyzed for private caches, using the same model as in Cole and Ramachandran [22]. It is similar to the parallel external memory (PEM) model of Arge et al. [5], with the primary difference being in the type of parallelism used. The PEM model uses bulk-synchrony for synchronization whereas we use binary forking. However, the use of private caches is almost identical. Each processor has a private cache of size M which consists of blocks of size B and the main memory of arbitrary size is shared amongst all processors. Several problems have been studied in the PEM model, particularly geometric problems including convex hull [41]. Sitchinava [41] showed that Atallah and Goodrich's [9] $O(n \log n)$ work and $O(\log n)$ span algorithm on presorted points matches the cache misses of sorting for the PEM model. This implies a convex hull algorithm for unsorted points that matches sorting bounds. However, in the binary-forking model and the binary fork-join model, the span of their algorithm becomes $O(\log^2 n)$ which is dominated by thread-spawning at each level in the $O(\log n)$ height recursion. We improve upon this result by presenting an algorithm with $O(\log n)$ span and $O(n)$ work for presorted points while preserving cache-obliviousness.

In terms of cache analysis for convex hulls, one of the earliest serial algorithms developed by Graham [32], when combined with a cache oblivious sorting algorithm, could achieve $O(n/B \log_M n)$ cache misses. Arge and Miltersen [6] 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. [31] for the external memory model. In terms of parallel cache-oblivious algorithms, Sharma and Sen [40] 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.

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

- For finding the convex of a set of presorted points (by x-coordinate), we give a deterministic cache-oblivious algorithm that uses $n^{1/5}$-way divide and conquer, performing $O(n)$ work in $O(\log n)$ span and incurring $O(n/B)$ cache misses. It is optimal across work, span and cache complexity for the problem of finding the convex hull of presorted points in the binary-forking model.

- For finding the convex hull of unsorted points, we give 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. To the best of our knowledge, this is the first deterministic

cache-efficient 2D convex hull algorithm for unsorted points in the binary-forking model with the lowest span. Its span outperforms the recently proposed randomized incremental convex hull algorithm for the binary-forking model as well [18].

- All our bounds hold for the binary fork-join model as well since neither algorithm uses the atomic test-and-set operation which is unique to the binary-forking model.

2 A Cache-Optimal $O(\log n)$ Span Algorithm for Presorted Points

We present a cache-oblivious divide-and-conquer algorithm for finding the upper convex hull of an array of presorted points. It achieves $O(\log n)$ span with $O(n)$ work and $O(n/B)$ cache misses, which are optimal bounds for work, span, and cache efficiency for the convex hull problem in the binary-forking model. As is common practice, we describe a procedure which finds the "upper hull", the set of extreme points which lie above the line from the leftmost and rightmost points. Repeating this procedure on the point set rotated by 180 degrees yields the entire convex hull. We consider our algorithm to be one of several to build off of the recursive structure of Atallah and Goodrich's [8] $O(n \log n)$ work algorithm. Atallah and Goodrich's algorithm is a $\sqrt{n}$-way divide-and-conquer algorithm. To perform the merges, it uses Overmars and Van Leuwen's [37] technique for finding the upper common tangent of two disjoint convex polygons to compare the $\sqrt{n}$ subproblem solutions and determine which points from each should remain. At each layer in this recursion, $\binom{\sqrt{n}}{2} = O(n)$ applications of this technique are used at the cost of $O(\log n)$, giving the recurrence $T(n) = \sqrt{n}T(\sqrt{n}) + O(n \log n)$ for the total work. The span of this method is $O(\log n)$, even under the more restrictive binary-forking model where $O(\log n)$ is optimal for any problem requiring $\Omega(n)$ work, as spawning $\Omega(n)$ threads incurs $\Omega(\log n)$ span. However, here the work remains suboptimal.

Goodrich [30] presented a method to obtain $O(n)$ work using a tree data structure which they refer to as a "hull tree". The hull tree data structure stores the convex hull of a set of points in a binary tree that permits a simulation of the Overmars and Van Leuwen's procedure and an efficient means of "trimming off" points that have been eliminated and unifying subproblem solutions under a new tree. The issue is that using this data structure incurs many cache misses because the tree cannot guarantee any data locality. Chen [21] also use a tree-based approach to achieve $O(n)$ work, which encounters the same issue.

The first step to ensure data locality while reducing to $O(n)$ work is to decrease the number of subproblems so that the pairwise hull comparisons do not take $O(n \log n)$ work. In fact, we want this work from these comparisons to total $O(n/B)$. This can be done by choosing a different exponent. We will use $n^{1/5}$-way divide-and-conquer, which means that comparing the subproblem solutions will take $O(n^{2/5} \log n)$ work, which allows us to use the tall cache assumption ($M = \Omega(B^2)$) to bound the total work from these comparisons by $O(n/B)$. Once the subproblems are compared, we can eliminate all of the points which have been determined to be within the hull by shifting all of the remaining points to the front of the array (using a prefix sum). This can be done cache-obliviously [16], incurring $O(n/B)$ cache misses for each recursion layer. However, this comes out to be $O(n \log \log n)$ work total. So, if we are to maintain data locality, we have to determine a way to eliminate points without having to shift everything down and incur too much work.

Instead of deleting points immediately, Algorithm 1 uses an array of size $O(\frac{n}{\log n})$ which stores a set of x-coordinate intervals corresponding to the points which must be deleted. We store these intervals as a list of "starts" and "ends" which correspond to the beginning and ending of each deletion interval all sorted by x-coordinate. For an interval I, consider the largest interval I' for which $I \subseteq I'$. As an invariant, we will store in $start(I)$ a pointer to $start(I')$ and in $end(I)$ a pointer to $end(I')$. This will crucially allow us to get from a point that is within one of these intervals to the nearest point which is not within any of these deletion intervals in $O(1)$ time, which lets us simulate the Overmars and Van Leuwen [37] technique on subproblem solutions. By primarily operating on this array of intervals, we can avoid incurring $O(n)$ work at every layer and instead amortize to $O(n)$ total work across all layers.

Data: A, a subset of points that are sorted by their x-coordinate, n the number of points in A, N the number of points in the original input, which is a superset of A, E an allotted section of global array D, n_E the size of alloted section E
Result: The upper hull of A
if $n < \log N$ **then** solve using a serial algorithm (such as Graham scan) ;
else
Split A into $n^{1/5}$ subproblems of size roughly $n^{4/5}$. Allot a contiguous segment of $\frac{n_E - 2n^{1/5}}{n^{1/5}}$ slots of E to each subproblem (in x-order). Solve these recursively;
Find the common tangents between each pair of subproblems using the modified Overmars and Van Leuwen technique (See Theorem 2.1). Store in a local array of tangents, T ;
Comparing the elements of T for each subproblem, determine the local set of new dividers, D'. Additionally store the dividers in the $2n^{1/5}$ slots at the end of E and another copy in D';
Merge the list of old and new dividers by x-order in E;
For all $d \in D'$, find d's copy in E. Store in $d.RANK$ the copy's rank in E;
For each (contiguous) left-right pair $(d_l, d_r) \subset D'$, mark the dividers $E[d_l.RANK, d_r.RANK]$ by setting their WRAPPER to $E[d_l.RANK]$ if a LEFT divider, $E[d_r.RANK]$ if RIGHT ;
For each element d in D', mark the points in A between its E copy's LANDING flag and the E copy's neighbor ($E[d.RANK + 1]$ for LEFT, $E[d.RANK - 1]$ for RIGHT);
if $n = N$ **then** Use prefix sums to determine the ranks of the unmarked (extreme) points and delete the marked points;
end

Algorithm 1: Cache-Optimal algorithm for Presorted Points

2.1 Data Structure Invariants

We use three data types in our arrays: **points**, **dividers**, and **tangents**. Each type is constant-sized, so the arrays have space linear in their respective numbers of elements.

Points are based on the input, with some extra attributes assigned to each point so that if it is not an extreme point, it can be marked instead of immediately removed. In addition to x- and y-coordinates, the point contains a RANK integer flag which is initialized to 1 and two pointers to dividers called LEFTPAR and RIGHTPAR. LEFTPAR and RIGHTPAR are initialized to NULL, but when they are set, they indicate that the point is within some deletion interval I. LEFTPAR points to a divider corresponding to the start of I, whereas RIGHTPAR points to a divider corresponding to the end of I.

Dividers represent the boundaries deletion intervals. If the divider represents the beginning of some interval I, we will refer to it as L_I. If the divider refers to the end of I, then we will refer to it as R_I. The divider contains an x-coordinate, a pointer, LANDING, that points to the point element corresponding to the start/end of the interval (but not within the interval), a pointer to another divider WRAPPER, an integer flag RANK and a bit flag indicating whether the divider is a "LEFT" divider (the start of an interval) or a "RIGHT" divider (the end of an interval). All of these are initialized to 0 or NULL.

Tangents represent an upper common tangent between the upper hulls of two subsections. An upper common tangent between two hulls is the line that lies above all points in each hull and which intersects the boundaries of each hull in exactly one contiguous section. This is typically expressed as a pair of extreme points, one from each hull. These are used to determine which points would be deleted if the two hulls were merged together. In our representation, each tangent simply contains 2 pointers to points, one for the point of tangency of the leftward of the two hulls, LEFTPT, and one for the point of tangency of the rightward of the two hulls, RIGHTPT. Both pointers are set to NULL to start.

The algorithm will utilize 4 arrays of these types: 2 permanent arrays (of points, A, and of dividers D) and 2 temporary arrays specific to the particular recursions (of tangents, T, and of dividers, D').

There are 5 invariants that are maintained with respect to D that will allow us to bound our work by $O(n)$ and our cache misses by $O(n/B)$.

Invariant 1 The size of D must be $O(\frac{n}{\log n})$.

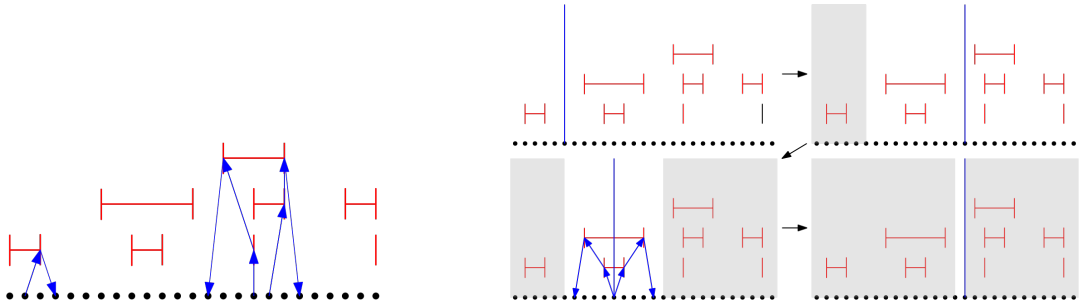


Figure 1: (a) Example paths from marked points to points that have not been marked; (b) Simulating the Overmars and Van Leuwen [37] procedure using the divider array. Only one of the two hulls are depicted. The blue line depicts the current point of consideration and the green the previous point of consideration.

Invariant 2 For any interval I with dividers $L_I, R_I \in D$, and the largest subset I' such that $I \subseteq I'$, $L_I.WRAPPER = L_{I'}$ and $R_I.WRAPPER = R_{I'}$. In the case of ties, take I' to be that was most recently added.

Invariant 3 For any two intervals I_1, I_2 with dividers in D, either $I_1 \cap I_2 = \emptyset$, $I_1 \subseteq I_2$, or $I_2 \subseteq I_1$.

Invariant 4 For any point p which is in at least interval, for one such interval I, $p.LEFTPAR = L_I$ and $p.RIGHTPAR = R_I$.

Invariant 5 For any interval I for which $L_I.WRAPPER = L_I$ and $R_I.WRAPPER = R_I$, then neither $L_I.LANDING$ nor $R_I.LANDING$ are elements of any interval.

We give an example depiction of this structure in Figure 1a. We show this in the context of paths from points that are marked for deletion (lie in some interval) to a nearest point to the left or right which is not in that interval. Assuming the invariants hold, we can claim the following theorem.

Theorem 2.1. *Given two convex hulls C_1, C_2, that are both expressed by an array of points A and an array of dividers D, and that are separable by a vertical line, the upper common tangent of C_1 and C_2 can be computed in $O(\log n)$ time serially.*

Proof. We assume that A and D abide by the 5 invariants listed above. We will simulate Overmars and Van Leuwen [37] on A, making use of D in the case that we accidentally land on a point which has been marked for deletion i.e. is not in either C_1 or C_2. Since we can proceed as usual if there are no dividers (and hence no intervals of deletion), we will consider only the case in which there have been points marked for deletion.

For any unmarked point, we will consider the *left path* and the *right path* in the divider structure. These paths follow down pointers from the source, so we will repreent the points in the path using $\rightarrow$ as a delimiter, indicating we are going to the point pointed to by that flag. These paths land on the nearest unmarked point to the left and right respectively. The left path is either $p \rightarrow LEFTPAR \rightarrow LANDING$ or, if $p.LEFTPAR$ points to the L_I for an interval I which is the subset of a larger interval I', then the path is $p \rightarrow LEFTPAR \rightarrow WRAPPER \rightarrow LANDING$. By invariant 5, we know that this path will land on a point which has not been marked. Specifically, this point is the point which is immediately outside I's (I''s) interval on the left side. Since $p \in I$ ($p \in I'$), any point between the end of the left path and p must be marked for deletion. Therefore, the left path gives the nearest point to the left. By symmetry, the same applies for the right path, given as $p \rightarrow RIGHTPAR \rightarrow LANDING$ or $p \rightarrow RIGHTPAR \rightarrow WRAPPER \rightarrow LANDING$. These paths are shown in blue in Figure 1a.

Using the left and right paths, we can deal with the case of points marked for deletion. The Overmars and Van Leuwen [37] technique relies on the ability to eliminate half of the remaining candidate points (for

the line of tangency, not the hull) from either C_1, C_2 or both at every step, which we also guarantee. This gives us the same $O(\log n)$ time guarantee since determining the points to eliminate from consideration for the upper common tangent takes $O(1)$ time.

In the first part of the Overmars and Van Leuwen [37] technique, the point with maximum y-coordinate, p_{top}, is found for each hull using a binary search. Consider C_1. Each step in this search determines whether p_{top} is to the left or right of the current point, p, by comparing it to its predecessor and successor in C_1. If the successor to p in the array A is marked for deletion, take the right path from this to get the true successor of p. For the predecessor, take the left path. If p itself is marked for deletion, then consider both the point at the end of the left path and the point at the end of the right path. These comparisons will either lead to the candidate set contracting to either the left or right of both or we will know that the top is one of the two, since the top point cannot be in the deleted interval. In either case, at least half of the points are eliminated from consideration since p lies at the midpoint of the candidates. We show an example search in Figure 1b.

The second part of the technique is another binary search, but this one relies on both hulls. At every step, we consider the midpoint of the remaining candidates in C_1 (and also for C_2). Just as before, if this point is marked, we instead consider the points at the end of the left path and the right path. The details of how to compare two points that are being considered (one from C_1 and one from C_2) are given in [37]. When the comparison is done, we can eliminate all candidates to the left of or to the right of at least one of the considered points. This holds for our case even if we have to consider 2 points from C_1 and two points from C_2. Let l_1, l_2 be the points from following the left paths and r_1, r_2 be the points from following the right paths. We represent the comparisons as pairs $(l_1, l_2), (l_1, r_2), (r_1, l_2), (r_1, r_2)$. Each comparison specifies that for at least one of the points, all points in its respective hull to either the left or the right cannot be a point of tangency. Whichever way the comparisons lie, one of the two hulls C_i will have both l_i and r_i able to eliminate points. If this is the case, then the argument from the search for p_{top} applies here. If only one of C_1 or C_2 has 2 points to consider from landing in an interval, without loss of generality consider C_1 to have the 2 points of consideration. Either both points in C_1 can eliminate candidates in the search, which gives us the same as the previous case or the point from C_2 can eliminate half to its left or right. In either case, half of the remaining candidates from C_1 or C_2 have been eliminated. If both C_1 and C_2 have 1 point to consider, it follows directly from the original Overmars and Van Leuwen technique.

Therefore, we can guarantee that only $O(\log n)$ comparisons need to be made to find the upper common tangent between C_1 and C_2. □

2.2 Maintaining the invariants

Now that we have a method to compare the hulls assuming the invariants hold, we will show how to merge the hulls, keeping in mind that we also must maintain the invariants. Lemma 2.2 will show how to maintain invariants 1 and 3, while Lemma 2.4 will show how to maintain invariants 2, 4, and 5.

Lemma 2.2. *The set of new dividers D' for a given recursion layer (of size n) can be determined in $O(\log n)$ span with $O(n^{2/5} \log n)$ work (and cache misses)*

Proof. We will first calculate the $O(n^{2/5})$ common tangents, storing them in a 2D array T. We use the smallest and largest sloped tangents (as in Atallah and Goodrich [8]) to determine which points should be removed from each subproblem hull, or subhull. For each subhull, there will be some section starting with the leftmost point and some section starting with the rightmost point which will need to be removed, leaving some contiguous set of points in the center, or no points from this hull. If there is such an set, we use a RIGHT divider with LANDING set to the first (leftmost) unremovable point and a LEFT divider with LANDING set to the last (rightmost) unremovable point. If there are no unremovable points from this subhull, then do not add any divider. We can omit dividers for the leftmost and rightmost points among all of these subhulls, since they will not correspond to intervals of deletion (they are extreme points). We can store these all in an array of size $2n^{1/5} - 2$, with gaps allowed for nonexisting dividers. Set the WRAPPER for each of these dividers to itself.

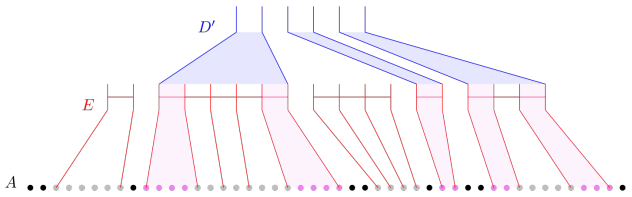


Figure 2: Each pair in D' binary searches for its copy, marks its subsidiaries and then marks the outermost layer of points.

At this point, $D' \cup D$ has satisfied invariants 1 and 3. 1 is satified since D must be of size at most $O(\frac{n}{4/5 \log n})$ and the addition of $2n^{1/5} - 2$ dividers does not affect this asymptotically. 3 is satisfied by induction, since when D is empty, this holds since it holds for D' alone, and assuming it holds for $D \neq \emptyset$, since D' does not have dividers that "land" inside previously marked points, they cannot intersect with any previous intervals corresponding to D without entirely containing such an interval. □

Lemma 2.3. *The set of new dividers D' for a given recursion layer (of size n) can be integrated into the full set of dividers D in $O(\log n)$ span with $O(\frac{n}{\log n})$ work and $O(\frac{n}{B \log n})$ cache misses.*

Proof. We can integrate these using a merge routine. Specifically, we will be merging a copy of D' into E, a set of contiguous slots in D that have been allotted to this subproblem. After alloting for each of the subproblems, there are $2n^{1/5}$ slots remaining at the end, allowing us to copy D' there. Before doing so, we will compress D' so they occupy contiguous positions. Only then will we copy them over to the end of E. These both can be done using prefix sums [16], which takes $O(n^{1/5})$ work and $O(n^{1/5}/B)$ cache misses. These calculations will use the RANK flag. □

Lemma 2.4. *All non-extreme points can be marked for deletion with $O(n)$ work and $O(n/B)$ cache misses, amortized.*

Proof. Now that the array D' has copies of its elements integrated into E, we can start by satisfying invariant 2. For each adjacent pair in D' corresponding to an interval of deletion I, conduct binary searches on E for their copies. Between these two copies will be a set of dividers corresponding to intervals which are subsets of I. To mark these, we will store the rank in E of the copies in the RANK flags of the D' elements. For the dividers in E which are LEFT, set their WRAPPER pointers to the LEFT element of the D' pair, specifically the copy in E. For the RIGHT divider, do the same with the RIGHT copy. This will satisfy invariant 2 by the same principle of induction as before. The marking procedure is cache efficient, incurring $O(\frac{n}{B \log n})$ cache misses at each recursion layer. The searches are not, but only incur $O(n^{1/5} \log n)$ work, which is dominated by comparing the subhulls.

At this point, invariant 5 has already been satisfied. Now that the WRAPPER flags have been set for intervals that are subsets, we can note that from before the LANDING flags for D' elements will not land inside previously marked points. This means that the only way a divider d could have a LANDING which has been marked for deletion is if that landing were in a new interval, in which case the interval of d, I, is a subset of the interval that marked $d.LANDING$, by 3.

All that remains is invariant 4. To maintain this, we mark the points for deletion. To ensure that we do not mark twice, we only consider dividers d in D'. Let d' be the corresponding divider to d that together represent interval I. Looking at d's copies in E, we take the neighboring divider: $E[d.RANK + 1]$ if d is a LEFT divider and $E[d.RANK - 1]$ if d is a RIGHT divider. These dividers have landing spots, between which are points will be marked for deletion (exclude the landing spot of a D'). Consider the d to be LEFT. Set LEFTPAR to $E[d.RANK]$ and RIGHTPAR to $E[d'.RANK]$. Do the reverse if d is RIGHT. This will satisfy invariant 4. We will mark the points in parallel, in groups of size at most $\log n$.

Since only points in "new" intervals are marked, this means that each point is marked at most 1 time, allowing us to amortize the work of marking to $O(n)$. We address some issues due to data alignment in 2.4,

but note here that the marking of points is done to adjacent points. We can charge 2 cache misses from each of these to the elements of D', which allows us ignore the issues of alignment on the ends of these intervals. This allows us to consider the "middles" of these intervals, which are access cache-efficiently. These will amortize to $O(n/B)$. □

2.3 Work, Span, and Cache Analysis

Theorem 2.5. *Algorithm 1 achieves $O(\log n)$ span with $O(n)$ work and incurring $O(n/B)$ cache misses*

Proof. First, we show span. To start, Algorithm 1 finds the convex hull of $O(\frac{n}{\log n})$ subhulls at the lowest level in the recursion layer. Each individual subhull is done serially using a serial algorithm. The Graham scan [32] is ideal since it achieves $O(n/B)$ cache misses for presorted input. With each subhull having size at most $\log n$, this takes $O(\log n)$ span. From here, merges are used. The merges start with finding the new dividers D', which takes $O(\log n)$ span from Lemma 2.2. From there, integrating them into the full set of dividers takes $O(\log n)$ span from Lemma 2.3. Finally, marking the points for deletion takes $O(\log n)$ span due to the binary searches. Marking the points and old dividers takes no more than $O(\log n)$ span since each point/divider can be done in parallel. this gives us the following recursion which solves to $O(\log n)$:

$$T_\infty(n) = T_\infty(n^{1/5}) + O(\log n)$$

The final step, using a prefix sum to align the unmarked points into adjacent array positions, takes only an additional $O(\log n)$ span [16].

For the work, the total work is dominated by the final prefix sum, $O(n)$ [16]. The $O(\frac{n}{\log n})$ base cases cost $O(n)$ work in total when Graham's algorithm is used. From Lemma 2.4, we know that marking points for deletion costs only $O(n)$ work amortized. From here, it suffices to show that all other recursive options cost $O(\frac{n}{\log n})$ work per recursion layer. The work from Lemmas 2.2 and 2.3 are dominated by $O(\frac{n}{\log n}$, which leaves the work on the divider array leading up to point marking. Setting the old dividers' WRAPPER flags is linear in the size of E, the available section of divider array. This is $O(\frac{n}{\log n})$ where n is the size of this recursion instance. Therefore, for the recusive steps, we have the following recursion for work:

$$T_1(n) = n^{4/5}T_1(n^{4/5}) + O\left(\frac{n}{\log n}\right)$$

With a base case of $T(\log n) = O(1)$ (we accounted for the work of the first layer earlier), we get a total work of $O(\frac{n \log\log n}{\log n}$ which is clearly dominated by $O(n)$.

For cache misses, we know that the serial algorithm on the $O(\log n)$ sized subproblems incurs $O(n/B)$ cache misses. From Lemma 2.4, we know that marking the points can be amortized to $O(n/B)$ by also charging some to the particular recursive layer. The recursive layers are cache efficient with the exception of $O(n^{2/5}\log n)$ binary searches (and other factors dominated by this). The cache efficient routines incur at most the work divided by B, which is $O(n/B)$. The binary searches incur misses until the recursion size is M or $\log n$, whichever is greater. If $\log n$ is greater, then simply bounding by the total work done in the recursions suffices, since for any constant c, $c\frac{n\log\log n}{\log n} \leq c\frac{n\log M}{M} \leq c\frac{n}{M^{1/2}}$ and by tall cache assumption this is $O(n/B)$.

If $M > \log n$, then we consider the following recursion:

$$Q(n) = n^{1/5}Q(n^{4/5}) + O(n^{2/5}\log n)$$

The solution to this recurrence is nontrivial, but we can use induction to show that $Q(n) = O\left(\frac{n}{B}\right)$. This is clearly satisfied in the base case since $Q(M) = O(1) \leq c_1 M/B$. For the inductive step, simply observe that $Q(n) = n^{1/5}Q(n^{4/5}) + c_2 n^{2/5}\log n$ which from the inductive assumption yields: $n^1/5(c_1\frac{n^{4/5}}{B}) + c_2 n^{2/5}\log n = c_1 n/B + c_2 n^{2/5}\log n$. From the tall cache assumption, $c_1 n/B$ dominates $c_2 n^{2/5}\log n$, therefore $Q(n) = O(\frac{n}{B})$ We complete the analysis of cache misses in the parallel setting in Section 2.4. □

2.4 Additional Cache Misses from Work-Stealing and False Sharing

Theorem 2.6. *Algorithm 1 incurs $O(\frac{n}{B} + S \cdot \frac{M}{B})$ cache misses (including false shares) under a work-stealing scheduler, where S is the number of steals.*

In the previous discussion of the number of cache misses incurred across the runtime of the algorithm, the only remaining source of cache misses is the scheduler. We use the same notion of work stealing as in Cole and Ramachandran [22], ie that in parallel sections, each processor places tasks into a work queue, and where idle processors "steal" work from the tops of these queues to decrease runtime. Associated with work-stealing is an additional type of cache misses, known as false sharing. False sharing occurs in private cache settings when a processor P_1 writes to a block which had previously been stored in the cache of another processor P_2 and then P_2 now has to access that block. Each step in Algorithm 1 corresponds to a set of parallel tasks, with synchronization after the conclusion of each step to ensure correctness. Different tasks only share data in the binary searches, where processors may concurrently reading from the same locations, but do not incur any false shares since nothing is updated. However, false shares can result from data alignment.

For cache-oblivious algorithms, there is no guarantee that subproblems will be partitioned along block boundaries. This can incur at most 1 additional cache miss per boundary due to false sharing. If we implement each step recursively, then we can ensure that these boundaries will only contribute a constant factor of additional false shares before steals. This applies to marking points and setting the WRAPPER flags of the dividers, but is redundant for the merging of dividers as Blelloch and Gibbons [17] is already implemented recursively.

Steals are another source of false shares. All of our steps which write to data do so in contiguous array locations within each parallel task. Tasks are ordinarily distributed to processors such that the write locations for successive tasks will be contiguous to each other. However, this contiguity can be disrupted by a steal. Even though a thief may update a piece of memory that won't be needed by the original processor, it may be within a block that is within the original processor's private cache, incurring a false share. Any particular write can affect at most $B - 1$ other processors which are operating in the same area, meaning that false shares only contribute $O(S \cdot B) \subset O(S \cdot \frac{M}{B})$ additional cache misses.

Aside from false shares, the only additional cache misses incurred due to steals are those incurred due the stealing processors accessing data that ordinarily would have been accessed by the original processor in order. This could be as bad as a stealing processor completely reloading as many blocks as could fit into the cache, implying at worst $O(S \cdot \frac{M}{B})$ cache misses. So in total, when using any work-stealing scheduler, there is at most $O(\frac{n}{B} + S \cdot \frac{M}{B})$ cache misses incurred by Algorithm 1.

Corollary 2.7. *Algorithm 1 incurs $O(n/B)$ cache misses w.h.p in p with the RWS scheduler when the number of processors is $p = O(\frac{n}{M \log n})$.*

We additionally consider a randomized work-stealing scheduler, the same as is described in Cole and Ramachandran [22] and which was originally described by Blumofe and Leiserson [19]. The randomized work-stealing scheduler (RWS) is a work-stealing scheduler where "idle" processors steal the head task from a randomly selected processor. If this fails, as in the task queue selected is empty, the scheduler simply tries again until either a task is found or the parallel section is completed.

Acar, Blelloch, and Blumofe [1] show that w.h.p. in p, the number of steals incurred during execution using the RWS is $S = O(p \cdot T_\infty)$. Therefore, if $p = O(\frac{n}{M \log n})$, then $S = O(\frac{n}{M})$, which when applied to the total number of cache misses yields $O(\frac{n}{B} + \frac{n}{M} \cdot \frac{M}{B}) = O(n/B)$.

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

The current state of the art for deterministic parallel sorting algorithms in the binary-forking model with regards to span is Cole and Ramachandran's [22] $O(\log n \log\log n)$ span cache oblivious algorithm. Since the algorithm is cache optimal, costing only $O((n/B)\log_M(n))$ cache misses, this means that using their algorithm to sort an instance of unsorted points, followed by our Algorithm 1 will yield a work and cache-optimal algorithm with $O(\log n \log\log n)$ span.

However, considering that implementation of Cole and Ramachandran's sorting algorithm is already required for this, a simpler approach for unsorted points is to modify the internal structure of the sorting algorithm. Overall, this modification is much simpler than implementing the presorted algorithm after already implementing Cole and Ramachandran's. We propose a divide-and-conquer algorithm for unsorted points which performs $O(n \log n)$ work in $O(\log n \log \log n)$ span while incurring only $O((n/B) \log_M n)$ cache misses. Our modifications are to account for the fact that sorting alone does not remove points which are not extreme points of the convex hull. 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 [6], when output sensitivity is not a consideration). The cache complexity of our algorithm matches Sharma and Sen's [40] randomized cache oblivious convex hull algorithm but does so in the worst case rather than in expectation.

Cole and Ramachandran's [22] cache-oblivious sorting algorithm is a divide-and-conquer algorithm with a relatively complicated recursive structure. It takes as input r sorted lists of elements with total size n. Depending on the relative size between n and r, the algorithm reorganizes the data so that elements of similar rank are grouped together into subproblems. This is accomplished by using a set of pivots which is of size approximately n/r^3 if r is sufficiently large. These subproblems are divided further into smaller subproblems which are then merged together. At this point the entire input has been sorted because the initial pivot-based partition ensures that for any two (larger) subproblems the ranks of all elements in one are larger than those of all elements in the other.

Our algorithm determines what we refer to as the "quarter-hull", and is thus iterated 4 times to produce the complete convex hull. We define the "quarter-hull" of a point set S as the counter-clockwise section of the convex hull between the point in S with the largest y-coordinate and the point in S with smallest x-coordinate. Since the convex hull is preserved under dimension-preserving linear transformations, it is clear that this algorithm can be applied to all 4 "quadrants" of the convex hull by rotation. It can also be the case that the smallest x-coordinate and largest y-coordinate points are the same point, in which case the quarter hull is only that singleton point and can be returned immediately. In case of a tie for x-coordinate, use the largest y-coordinate and vice versa.

The data structure is much simpler than in the presorted algorithm. For each point, p, we simply need to include an additional (x, y) coordinate pair, which we will refer to as the TAIL, and two integers RANK, and HOME. If the TAIL lies to the left and below p, this indicates that it is an extreme point, otherwise it is not. Implicitly, the TAIL defines an "active range" for p starting with it predecessor in the hull. However, the TAIL is not a pointer, it explicitly stores the coordinates of the predecessor in the hull. We will refer to p itself as the HEAD of the point. RANK will be used to perform all prefix sums and HOME will mark the "true" subset that it would be placed in if the TAIL were the same as the point itself.

Theorem 3.1. *For n points in the plane, Algorithm 2 (Multiway Merge) finds their quarter hull in $O(n \log n)$ work and $O(\log n \log \log n)$ span.*

Proof. Our algorithm differs from Cole and Ramachandran's [22] in two key ways. The first is the size of the subsets A_i, since ours includes points which may be on the other side of the pivot but their TAILs are on the correct side, there can be at most r more elements in each of our subsets than in the original sorting algorithm. This is because each list can only contribute 1 copy of a point which "belongs" in another subset since the lists are given as quarter hulls. However, since the size of their subsets is at most $3r^3 - r^2 - r$ (From Lemma 2.1 of [22]) and the bound required for the recursion to hold is that the subsets be of size at most $3r^3$, this will not be an issue. Ours will still work since $3r^3 - r^2 \leq 3r^3$. This overlap also presents marginal difficulty in the implementation of the partition step. Within each of the r lists, it is generally the case that a particular pivot will lie between the HEAD and TAIL of a point, leading to that point being represented in more than one subset in the partition. To achieve this, Cole and Ramachandran find the element of each list which is just after the point in the ordering to determine which subsection of values needs to be copied into each bucket. We can simply buffer these ranges for each bucket and we will be sure to capture all the required points, incurring specifically the extra r for each bucket.

The other difference is that we have to check the boundaries of each subset to make sure that the last point in some quarter hull A_i is not invalidated by A_{i+1} and vice versa. That is determined by observing

Data: A, a collection of quarter hulls, $L_1, L_2, ..., L_r$ where $n \leq 3r^6$
Result: The combined quarter hull of A
if $n \leq 24$ **then** apply any serial convex hull 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 point in each L_i, for a total of n_i/r^2 points from each quarter hull, where n_i is the number of extreme points defining quarter hull L_i;
  Compute ranks of elements of S using TAIL;
  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 x-coordinate pivots (by TAIL), partition A into $k = |P| + 1$ subsets $(A_1, ...A_k)$. Include the points for which the x-coordinate interval between its TAIL and itself contains some overlap with the interval between pivots $P[i-1], P[i]$ (See Figure 6);
end
parallel foreach *subset* A_i *of* A *(from* 1 *to* k*)* **do**
  Separate A_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 $p \in [A_1, A_2, ...A_k]$ **do**
  Mark any point for which its active range no longer overlaps with the subset it is assigned to.
end
Using prefix sums on each subset, eliminate any marked points. **parallel foreach** $i \in [1, k]$ **do**
  Determine if the first element of A_{i+1} invalidates the last element of A_i. If so, set the TAIL of the first element of A_{i+1} to the second to last element of A_i. Otherwise, set its TAIL to the last element of A_i.
end
Using prefix sums, reassign all the identified extreme points into adjacent sorted positions in an output array.

Algorithm 2: Divide-and-Conquer with Multiway Merge

the quarterhull of the boundary points and their TAILs (see Figure 3b). Update the TAILs of the points to be equal to the predecessor along this quarter-hull. If a point is deleted, update it such that the new first point in A_{i+1} has the new last point in A_i as its TAIL.

We do not need to check any more than the boundaries. Since we delete from the subsets points whose intervals are outside the subset's interval induced by the pivots, we know that the only points that can have intervals which are partially outside the subset's interval are at the boundaries. We also know that every point's interval is disjoint from those of the other points except for the x-coordinate at which one point's TAIL coincides with the HEAD of another.

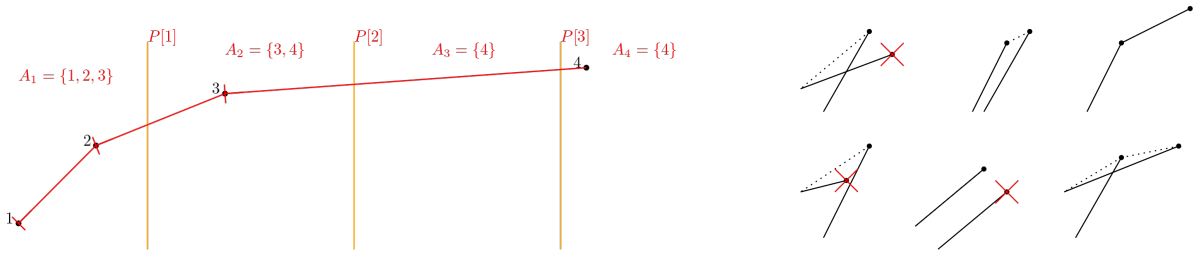
This check takes $O(\log r) \leq O(\log n)$ span and performs $O(n/r^3) \leq O(\sqrt{n})$ work, which is dominated by the work and span of the rest of the algorithm. Therefore, the work and span bounds of our algorithm are equivalent to Cole and Ramachandran's sorting algorithm [22], $O(n \log n)$ and $O(\log n \log \log n)$, respectively. □

Theorem 3.2. *For n points in 2D, Algorithm 2 finds their convex hull while incurring at most* $O((n/B) \log_M n)$ *cache misses.*

Proof. This follows directly from the Cole and Ramachandran bounds. We use the same tall cache assumption as before, that $M = \Omega(B^2)$. At each recursion layer, at most $O(n/B)$ cache misses are incurred. This is trivially true for the elimination of points who are not active within their subset's ranges (steps 12 and 13 in Algorithm 2). It is also trivial for the elimination phase at the end of the execution (step 15).

The additional copies of points which belong in multiple buckets also incur at most $O(n/B)$ additional cache misses. The reads required to make these additional copies are contiguous with the other points which are being placed into that bucket, so it can be at worst an additional $O(n/B)$ cache misses, since it would have the exact same misalignment issues as Cole and Ramachandran's.

For the comparisons in Step 14, we access at most s/r memory locations, where s is the size of the sample S. Since $s/r < s \cdot r$, then even if every memory access in the Step 14 is a cache miss, we will not exceed $O(n/B)$ cache misses at each recursion layer. Hence, the recursion for Cole and Ramachandran's



(a) (b)

Figure 3: (a) Showing how the pivots work in Algorithm 2; (b) Possible configurations of points on the boundaries between subsets in Algorithm 2. TAILs before the check are shown with solid lines and updated TAILs are with dotted lines.

sorting algorithm also applies here: $Q(n,r) = O(n/B) + \sum_{i,j} Q(n_{ij}, \sqrt{r}) + \sum Q(n_i, \sqrt{r})$ which solves to $Q(n,r) = O(\frac{n \log n}{B \log M})$ □

Theorem 3.3. *Using any work-stealing scheduler, Algorithm 2 incurs at most* $O((n/B)\log_M n + S \cdot \frac{M}{B})$ *cache misses where* S *is the number of steals.*

Proof. This is the exact same bound as is given in Cole and Ramachandran [22], so it suffices to show that at most $O(M/B)$ cache misses are incurred when a processor steals one of the modified tasks. The sampling and partition phase (Steps 2-6 in Algorithm 2) is very similar to Cole and Ramachandran's, and incurs the same number of cache misses due to steals and false sharing asymptotically. This follows from the fact that each parallel task from the modified procedure in Step 6 only makes a constant number of writes, incurring at most $O(S \cdot B) \subset O(S \cdot \frac{M}{B})$ (due to the tall cache assumption) false shares.

The decomposition into even smaller subproblems (Steps 7-10) uses the same structure as Cole and Ramachandran's sorting algorithm and thus is already know to incur an additional $O(S \cdot \frac{M}{B})$ cache misses. All that remains is the elimination of points which extend outside the bucket they were placed in during partitioning (Steps 12-13), checking the boundaries (Step 14), and the final deletion (Step 15). Both rounds of deletions also perform only a constant number of writes, which means that only $O(S \cdot \frac{M}{B})$ false shares can be incurred. The exact same bound applies to the boundary checks.

Aside from the cache misses incurred by false shares, the cache misses incurred from performing a steal can be at most M/B, since any more cache misses would be incurred without steals. This means that in total, there will be $O(S \cdot \frac{M}{B})$ additional cache misses due to the use of a work-stealing scheduler. □

Corollary 3.4. *Algorithm 1 incurs* $O(n/B \log_M(n))$ *cache misses w.h.p in* n *with the RWS scheduler when the number of processors is* $p = O\left(\frac{n}{M \log M} \cdot min\left(\frac{1}{\log \log n}, \frac{M \log B}{B^2}\right)\right)$.

Proof. This comes directly from our Theorem 3.3 and Theorem 1.2 [22]. Since our algorithm matches the false sharing and cache miss bounds of Cole and Ramachandran's, their analysis directly carries over. □

4 Conclusion

We have presented two cache-oblivious algorithms for the binary-forking model which find the convex hull of a set of points in 2 dimensions. We presented the first cache-oblivious algorithm for presorted points which achieves optimal span $O(\log n)$ with optimal work, $O(n)$, while incurring the optimal amount of cache misses $O(n/B)$. The current state of the art for sorting in the binary forking model achieves $O(\log n \log \log n)$ span with $O(n \log n)$ work and while incurring $O(n/B \log_M(n)$ cache misses. Here only span is suboptimal when considering only the size of the input (as opposed to also considering the size of the output). By modifying this algorithm, we provide a simpler method to achieve these bounds for convex hulls compared to sorting

and then applying any algorithm for presorted points that we are aware of. We believe our techniques for converting merge-based sorting algorithms into convex hull algorithms can be generalized to many merge-based sorting algorithms. Further work would entail expanding the algorithm for presorted input to the more general case of simple polygons, reducing the span of sorting in the binary forking model and developin a cache-oblivious algorithm which achieves work which is optimal with respect to the size of the output as well as the input.

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] 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.

[4] 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.

[5] Lars Arge, Michael T. Goodrich, Michael Nelson, and Nodari Sitchinava. Fundamental parallel algorithms for private-cache chip multiprocessors. In *Proceedings of the Twentieth Annual Symposium on Parallelism in Algorithms and Architectures*, SPAA '08, page 197–206, New York, NY, USA, 2008. Association for Computing Machinery. doi:10.1145/1378533.1378573.

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

[7] 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.

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

[9] Mikhail J Atallah and Michael T Goodrich. Parallel algorithms for some functions of two convex polygons. *Algorithmica*, 3(1):535–548, 1988.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] Guy E Blelloch, Phillip B Gibbons, and Harsha Vardhan Simhadri. Low depth cache-oblivious algorithms. In *Proceedings of the twenty-second annual ACM symposium on Parallelism in algorithms and architectures*, pages 189–199, 2010.

[17] Guy E. Blelloch, Phillip B. Gibbons, and Harsha Vardhan Simhadri. Low depth cache-oblivious algorithms. In *Proceedings of the Twenty-Second Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '10, page 189–199, New York, NY, USA, 2010. Association for Computing Machinery. doi:10.1145/1810479.1810519.

[18] 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.

[19] Robert D. Blumofe and Charles E. Leiserson. Scheduling multithreaded computations by work stealing. *J. ACM*, 46(5):720–748, sep 1999. doi:10.1145/324133.324234.

[20] 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.

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

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

[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 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.

[30] Michael T Goodrich. Finding the convex hull of a sorted point set in parallel. *Information Processing Letters*, 26(4):173–179, 1987.

[31] 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.

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

[33] 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.

[34] J. JaJa. *An Introduction to Parallel Algorithms*. Addison Wesley, 1997.

[35] 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.

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

[37] 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.

[38] 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.

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

[40] 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.

[41] Nodari Sitchinava. Computational geometry in the parallel external memory model. *SIGSPATIAL Special*, 4(2):18–23, jul 2012. doi:10.1145/2367574.2367578.

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

[43] https://www.threadingbuildingblocks.org, (TBB).