Parallel Topological Sweep
Ming Ouyang*

Abstract
On input of a line arrangement, topological sweep outputs the line intersections in a topological order. The intersection of two lines is ready if all intersections to the left on these two lines have been processed. The classical algorithm processes the ready intersections one at a time. This article describes the first attempt to process the ready intersections in parallel. It is proved that, at the beginning of the sweep of a random arrangement, the expected number of ready intersections is a constant fraction of the number of lines. After the first batch, empirical data show that many intersections become ready batch after batch. Two new implementations are described. On arrangements of 300,000 lines, a new serial implementation is 3.92 times the speed of a serial implementation in the literature, and the first parallel implementation is 4.2 times the speed of the new one.

1 Introduction
Topological sweep [3] is a classical algorithm in computational geometry. The input is an arrangement of \( n \) lines in the plane. The intersection of two lines is a vertex. An arrangement is simple if any two lines intersect at a vertex, but no three do. The algorithm sweeps the arrangement — reporting the vertices — using \( O(n^2) \) time, which is asymptotically optimal. It is used in efficient algorithms for applications, such as data depth [4, 5, 7, 8, 9]. The algorithm is implemented in C by Rosenberger [3, 12] and in C++ using the LEDA library by Miller et al. [8]. It is extended to handle non-simple arrangements by Rafalin et al. [11] — they implement the extended method in C++ without using any standard libraries. The algorithm and implementations are serial in nature. Parallel topological sweep is needed for two reasons. First, in the past five decades, the performance of computers has more or less doubled every 18 months. This so-called Moore’s Law, however, is showing signs of plateauing. Second, experimental scientists, enabled by technology, are collecting more and larger data sets than ever. Analysis of large data sets, such as finding the Tukey median [8], is difficult without parallelization. The author is unaware of any prior attempt at parallelizing topological sweep.

Section 2 reviews the line-point duality and the classical algorithm. Section 3 examines how to parallelize it. Section 4 studies the expected concurrency in random arrangements. It is proved that \( \Omega(n) \) intersections are ready at the beginning of the sweep. Empirical data show that, on average, a constant fraction of the lines are engaged in ready pairs throughout the sweep. Section 5 describes a new serial implementation in C and the first parallel implementation in C and OpenMP. The new serial code is 3.92 times the speed of the Rafalin code [11]. The parallel code is 4.2 times the speed of the new serial code and more than 16 times that of the Rafalin code. Section 6 concludes with discussion.

2 Serial Topological Sweep
The description of the classical algorithm is expanded beyond that of [3] to include the dual space of point arrangement, which makes easy a proof in Section 4. Let \( A \) be a simple arrangement of \( n \) lines. As in [3], it is assumed that none of the lines is vertical. Some applications, such as data depth [5, 8], need to process the vertices of \( A \) in some order. They require that the vertices on the same line be listed monotonically — vertices on different lines may come in any order. Topological sweep produces such a topological sort of the vertices.

The line-point duality maps a line \( y = c_1 x + c_0 \) to the point \((c_1, c_0)\), and a point \((c_1, c_0)\) to the line \( y = -c_1 x + c_0 \). Fig. 1(a) shows two lines and their intersection. Fig. 1(b) is the dual arrangement. The duality preserves incidence — the dual line of the intersection point is incident upon the dual points of the lines. Sweeping a line can be construed as walking along its upper and lower sides to detect whether the line comes to an intersection as the upper or lower line. Fig. 1(c) shows walking the lines of Fig. 1(a) from left to right. For the upper line, \( y = -4x - 3 \), its lower sidewalk is blocked by the lower line, but its upper sidewalk overrides the intersection and continues to the right. For the lower line, \( y = -x + 2 \), its upper sidewalk is blocked by the upper line, but its lower sidewalk underpasses the intersection and continues to the right. The duals of the upper and lower sidewalks are, metaphorically, the left and right halves, respectively, of the dual point. Imagine the dual point as a clock. The dual of the upper sidewalk — moving from negative infinity to infinity — is a hand, the lower hand, that rotates from six o’clock to twelve o’clock, whereas the dual of the lower sidewalk is the upper hand that rotates from twelve o’clock to twelve o’clock.

*Department of Computer Science, UMass Boston
Figure 1: Line-point duality.

to six o’clock. During such a rotation, a hand must make a stop and point at any other point. When the upper hand of one point lines up with the lower hand of another point, the two hands merge and form the line dual of the primal vertex. As shown in Fig. 1(d), the upper hand of (−4, −3) rotates from twelve o’clock until it points at (−1, 2). The lower hand of (−1, 2) rotates from six o’clock until it points at (−4, −3). Not shown in Fig. 1(d), both the lower hand of (−4, −3) and the upper hand of (−1, 2) make a half-circle rotation.

The lines in the arrangement are sorted by their slopes. A topological sweep line — a cut — is monotonic in the y-direction, intersects each of the n lines once, and does not pass through any vertices. Fig. 2(a) shows the first cut of an arrangement of five lines, \( L_1, \ldots, L_5 \). The first cut walks the lines from left to right and stops before the vertices. An array, \( \text{cut}[i] \), stores the order of the lines along the cut. Initially, \( \text{cut}[1] = L_1 \), \( \text{cut}[2] = L_2 \), and so on. When the cut advances over a vertex, the two intersecting lines, which must have been adjacent in the cut, will swap their places.

The upper and lower horizon trees — UHT and LHT — are the main data structures. The solid lines in Fig. 2(b) and (c) are the initial UHT and LHT, respectively. The exposition of [3] illustrates the algorithm with the UHT. This article uses the LHT. When two lines meet in the LHT, the upper line has higher precedence — it continues to the right and blocks the lower line from proceeding. The precedence in the UHT is reversed. The trees are stored in two arrays, \( \text{uht}[i] \) and \( \text{lht}[i] \), where each element is the entity that blocks the line from proceeding. For example, in the LHT, \( L_2, L_3, \) and \( L_4 \) are blocked by \( L_1 \). Each line \( L_i \) has two obstacles, \( \text{uht}[i] \) and \( \text{lht}[i] \). The obstacle that is closer to the cut is stored in \( \text{closer}[i] \). This array \( \text{closer}[i] \) is a succinct representation of the set-intersection of the UHT and LHT (Fig. 2(d)). The crucial observation is that, when \( \text{closer}[\text{cut}[i]] \) is \( \text{cut}[i+1] \) and \( \text{closer}[\text{cut}[i+1]] \) is \( \text{cut}[i] \), the intersection of \( \text{cut}[i] \) and \( \text{cut}[i+1] \) is ready for the sweep line to cross. In Fig. 2(d), \( (L_1, L_2) \) is a ready pair, so is \( (L_4, L_5) \). Table 1 lists the initial values of the data structures. These data structures are simplified from those in [3], which store both the left and right endpoints of the UHT and LHT. Herein only the right endpoints are kept. The simplified version is sufficient to produce a topological sort of the vertices and conducive to fast implementation.

The ready pairs are stored in an array \( \text{ready}[i] \). A pair is represented by the rank in the cut of its first member. In Table 1, \( \text{ready}[1] = 1 \) and \( \text{ready}[2] = 4 \). During the sweep, one ready pair is removed from \( \text{ready}[i] \), and up to two new pairs may be added back to it. This array \( \text{ready}[i] \) can be managed either as a queue or a stack, because the pairs can be processed in any order. Herein lies the source of concurrency.

The upper and lower horizon trees correspond to the
lower and upper sidewalks, respectively, which in turn correspond to the upper and lower hands, respectively, in the dual space. For example, in the LHT, $L_1$ goes all the way to the right, so its lower hand in the dual space rotates nonstop from six o’clock to twelve o’clock. The lower hands of $L_2$, $L_3$, and $L_4$ stop at the dual of $L_1$, and that of $L_5$ stops at the dual of $L_4$. This correspondence will be used in Section 4.

![Table 1: The initial data structures for the arrangement in Fig. 2](image)

<table>
<thead>
<tr>
<th>i</th>
<th>cut[i]</th>
<th>uht[i]</th>
<th>lht[i]</th>
<th>closer[i]</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>2</td>
<td>$\infty$</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>5</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>5</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>5</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>$\infty$</td>
<td>4</td>
<td>4</td>
</tr>
</tbody>
</table>

Listing 1: Pseudo-code of constructing the initial LHT

Listing 1 is the pseudo-code for constructing the initial LHT, which is initialized with $L_1$ by setting $\text{lht}[1]$ to infinity. The major step is the clockwise traversal of the bay. Consider the scenario when $L_3$ enters the scene after $L_1$ and $L_2$ are already in place (Fig. 2(c)). The bay here consists of $\text{clockwise}$, $L_3$ and $L_1$ — see Fig. 3(a) for a more complicated example, where Bay consists of $L_5$, $L_4$, $L_3$, and $L_1$. The while loop on lines 6–13 performs the bay traversal and finds the next intersection of $L_3$, which is with $L_1$. Before $L_3$ is inserted, the bay consists of $L_2$ and $L_1$. After $L_3$ is inserted, $L_2$ drops below the horizon, and the bay consists of $L_3$ and $L_1$. $L_2$ is no longer visible to subsequent $L_j$’s. The same situation of dropping below the horizon happens to $L_3$ after $L_4$ is inserted. The time of constructing the LHT is $O(n)$ because the while loop will iterate at most $3(n−1)$ times, accounted for in three categories. First, there is one iteration per $L_i$ as the first iteration of the loop, for $n−1$ iterations in total. Second, there is one iteration every time when the if condition (line 8) is true, for at most $n−1$ iterations in total. When this happens, a previous line drops below the horizon. This can happen at most once for every line. Third, there is one iteration every time when the if condition (line 8) is false, for at most $n−1$ iterations in total. When this happens, the loop terminates via the break statement.

When the cut advances beyond a ready pair, the algorithm needs to update the horizon trees. This is illustrated with the ready pair $(L_4, L_5)$ in Fig. 2(d). The new UHT is shown in Fig. 2(e). The right endpoint of $L_5$ remains the same. As for $L_4$, the bay beyond $L_5$ is empty, so $\text{uht}[4]$ is set to infinity. The new LHT is shown in Fig. 2(f). The right endpoint of $L_4$ remains the same. As for $L_5$, the bay beyond $L_4$ — consisting of $L_1$ and $L_3$ — becomes visible. Thus, $L_5$ has to traverse the bay clockwise to find its new endpoint. Listing 2 is the pseudo-code for updating the LHT. The second member of the pair — the lower line — is designated as $L_i$. The bay traversal is performed by the while loop on lines 7–15. This traversal is similar to the one in the initial construction of the LHT. The complication is the if statement on lines 10–12. When the if condition is true, $L_j$ is located to the right of the cut, and its intersection with $L_i$ is duly considered. When it is false, the intersection of $L_i$ and $L_j$ is to the left of the cut and has already been processed, so the if statement has no else part. After the UHT and LHT are updated, the intersecting lines swap their places in the array cut[] and may be engaged in up to two new ready pairs.

Listing 2: Pseudo-code of updating the LHT

The time for processing one ready pair is $O(n)$. The total time for the complete sweep would be $O(n^2)$ but is only $O(n^2)$ via amortized analysis. Recall that the lines in the arrangement are sorted by increasing slopes. $L_i$ will participate in $n−i$ pairs as the upper line and will perform traversal of the UHT. It will participate in $i−1$ pairs as the lower line and will perform traversal of the LHT. Consider a fixed $L_i$. Aggregating over all of the bay traversals of the UHT and LHT, the while loops will iterate at most $3(n−1)$ times in total. Thus, the time for processing the intersections on one line is $O(n)$, for a total time of $O(n^2)$ for the sweep.

3 Parallel Topological Sweep

Multiple ready pairs can be processed in parallel. The task is to keep the horizon trees consistent. Fig. 3(a) is an LHT with three ready pairs: $(L_1, L_2)$, $(L_4, L_5)$, and $(L_6, L_7)$. Recall that in updating the LHT, the
right endpoints of the upper lines — \( L_1, L_4, \) and \( L_6 \) — remain unchanged. The lower lines need to traverse the bays beyond their partners to find new endpoints. When \( L_2 \) crosses beyond \( L_1 \), it will proceed to infinity. When \( L_5 \) crosses beyond \( L_4 \), it will meet \( L_3 \). Even if \( L_3 \) is absent from the arrangement, \( L_5 \) can not intercept \( L_2 \) before \( L_2 \) intersects with \( L_1 \). Otherwise, \( L_2 \) would not be a ready partner with \( L_1 \). Similarly, when \( L_7 \) crosses beyond \( L_6 \), it can not intercept \( L_5 \) nor \( L_2 \). Thus, the lines coming from below — \( L_2, L_5, \) and \( L_7 \) — may simultaneously traverse their bays for new intersections. These simultaneous traversals entail concurrent read of \( \text{ht}[] \) by multiple processors, which is innocuous. What is critical is that after a processor has finished its traversal, it must hold off updating its local region of \( \text{ht}[] \) until all traversals are done. Otherwise, there will be a race condition, as illustrated in the following scenario. Assume that \( L_5 \) finishes its traversal of \( \text{Bay}_3 \) and updates \( \text{ht}[5] \) to \( L_3 \) right away. At that moment, if \( L_7 \) has already moved from \( L_5 \) to \( L_4 \), no harm is done. If, however, \( \text{ht}[5] \) is overwritten before \( L_7 \) finishes with \( L_5 \), then \( L_7 \) will follow the new \( \text{ht}[5] \) to \( L_3 \) rather than \( L_4 \). This would break the algorithm. Thus, the processors must synchronize before they write to \( \text{ht}[] \). When they do write, they write to different parts of \( \text{ht}[] \) — there is no risk of concurrent write. Note that if \( (L_6, L_7) \) is sequentially processed after \( (L_4, L_5) \), then \( L_7 \) will visit only \( L_4 \) and \( L_3 \). With parallel processing, \( L_7 \) will visit \( L_5 \) in addition to \( L_4 \) and \( L_3 \). This is a source of parallel overhead.

Let the variable \( \text{numReady} \) be the number of ready pairs stored in the array \( \text{ready}[1..\text{numReady}] \). Listing 3 describes how to process all ready pairs in parallel. The \text{parFor} loop on line 1 is executed by \text{numReady} processors simultaneously. Each processor works on one pair. The clause \text{private()} on line 1 reserves two private variables for every processor that can be read and written without contention with other processors. The imperative \text{synchronize} on lines 5 and 10 stipulates that all processors must finish the proceeding steps before any of them proceed further. At line 11, each processor looks above and below to see if its two lines will be engaged in new ready pairs. Care must be taken that a new pair is identified exactly once. For example, assume two processors work on two adjacent pairs \((L_{\text{cut}}[i], L_{\text{cut}}[i+1])\) and \((L_{\text{cut}}[i+2], L_{\text{cut}}[i+3])\). If \( L_{\text{cut}}[i] \) and \( L_{\text{cut}}[i+3] \), which will have become \( L_{\text{cut}}[i+1] \) and \( L_{\text{cut}}[i+2] \), form a new pair, only one processor should add it to \( \text{ready}[] \). After peeking at each other’s data, the first processor leaves this task to the second processor. Furthermore, the new ready pairs must be collated and saved consecutively at the front of \( \text{ready}[] \). Because there may be zero, one, or two new pairs per existing one, the processors do not know in advance where to write in the array \( \text{ready}[] \). This can be solved with prefix sum. For example, let this sequence \([1 \: 0 \: 1 \: 0 \: 0] \) be the numbers of new pairs found by six processors. The exclusive prefix sum of this sequence is \([0 \: 1 \: 2 \: 2 \: 3] \), which can be computed in \([\log_2 6] \) steps using the parallel prefix sum algorithm [6]. Adding one to the exclusive prefix sum, the sequence \([1 \: 2 \: 2 \: 3 \: 3 \: 4] \) is the locations in the array \( \text{ready}[] \) where the processors can write down their ready pairs in parallel.

The computation in Listing 3 constitutes one stage of parallel topological sweep that processes one batch of ready pairs and produces the next batch. The algorithm repeats stage after stage until all \( \binom{n}{2} \) pairs are processed. Although parallelization incurs some overhead, the overall time remains \( O(n^2) \) if the parallel computation is serialized. Parallelization does not change the observation that the \text{while} loops for bay traversals will iterate at most \( 3(n-1) \) times on behalf of each line.

### 4 Expected Concurrency

This section studies the expected number of ready pairs at the first stage. The lines are generated via the dual. The dual points are uniformly distributed in the interior of the unit circle, excluding the origin. For each point, its polar angle is a uniform random number between 0 and \( 2\pi \). Its distance to the origin is the square root of a uniform random number in the open interval \((0,1)\). Square root is taken because the area is proportional to the square of the distance. The polar coordinates are converted to the Cartesian coordinates, which become the coefficients of the lines in the primal arrangement.
This section, however, works with the dual.

Let $x_1, \ldots, x_n$ be the sorted x-coordinates of the points $P_1, \ldots, P_n$. The $x_i$s are distinct because identical ones would result in parallel lines in the primal arrangement, which is assumed to be simple. The $n$ vertical lines, $x = x_i$, shred the unit circle into $n + 1$ vertical strips. At least $\lceil n/2 \rceil$ strips have widths less than $4/n$. Otherwise, the strips would be wider than the unit circle. Fig. 3(b) shows such a strip. Recall that walking a line from left to right corresponds to rotating two hands around the dual point. The UHT corresponds to the upper hands that start at twelve o’clock. The LHT corresponds to the lower hands that start at six o’clock. They rotate clockwise and stop at other points. The set-intersection of the UHT and LHT is to take the smaller rotation of the two hands. For example, if the upper hand has rotated to two o’clock and the lower hand to seven o’clock, the smaller rotation is $\pi/6$. In Fig. 3(b), if the duals of $P_i$ and $P_{i+1}$ form a ready pair at the first stage, their four hands have the following configuration. First, the upper hand of $P_i$ has rotated to $P_{i+1}$, and the antipodal image of its lower hand has rotated beyond $P_{i+1}$. Second, the lower hand of $P_{i+1}$ has rotated to $P_i$, and the antipodal image of its upper hand has rotated beyond $P_i$. If the pair is not ready, at least one of their hands has stopped at a third point that resides in one of the shaded pies below $P_i$ and above $P_{i+1}$.

Lemma 1 Assume $P_i$ and $P_{i+1}$ are in the bottom and top quarters, respectively, of their vertical lines. If the strip between $P_i$ and $P_{i+1}$ has a width less than $4/n$, the area of the shaded pie above $P_{i+1}$ is at most $1/n$. So is the area of the shaded pie below $P_i$.

Proof. When $P_{i+1}$ is in the top quarter of its vertical line, the length of the vertical side of the shaded pie above $P_{i+1}$ is at most $1/2$. The horizontal span of the arc is at most the width of the strip, $4/n$. Thus, the pie fits inside the right-angled triangle with an area of $1/n$. So is the pie below $P_i$. □

Let $X_i$, $i = 1, \ldots, n - 1$, be the indicator that the duals of $P_i$ and $P_{i+1}$ are a ready pair. Let $X$ be the random variable of the number of ready pairs at the first stage.

Theorem 2 If the $n$ dual points are uniformly distributed in the unit circle, the expected number of ready pairs at the first stage is $\Omega(n)$.

Proof. The probability $P_i$ and $P_{i+1}$ are in the bottom and top quarters, respectively, of their vertical lines is $1/4 \cdot 1/4 = 1/16$. If the strip has a width less than $4/n$, the probability a point is in a shaded pie below $P_i$ or above $P_{i+1}$ is less than $(2/n)/\pi = 2/(\pi n)$. The probability the duals of $P_i$ and $P_{i+1}$ are a ready pair is

$$\Pr(X_i = 1) \geq \frac{1}{16} \left(1 - \frac{2}{\pi n}\right)^{n-2}.$$

Although at least $\lceil n/2 \rceil$ strips have widths less than $4/n$, the leftmost and rightmost strips must be excluded. Both of them are demarcated by only one point.

$$E(X) = \sum_{i=1}^{n-1} E(X_i)$$

$$\geq \left(\frac{n}{2} - 2\right) \frac{1}{16} \left(1 - \frac{2}{\pi n}\right)^{n-2}$$

$$= \left(\frac{n}{2} - 2\right) \frac{1}{16} \exp \left((n-2) \ln \left(1 - \frac{2}{\pi n}\right)\right)$$

$$\geq \left(\frac{n}{2} - 2\right) \frac{1}{16} \exp \left(- \frac{2n - 4}{\pi n}\right)$$

$$\geq \left(\frac{n}{2} - 2\right) \frac{1}{16} e^{-2}.$$

□

By the theorem, the expected number of ready pairs is at least $0.01n$. Empirical data suggest there are more than that. Fig. 4(a) shows the boxplots of the numbers of ready pairs at the first stage for $n$ from 10,000 to 200,000. For each $n$, 100 random arrangements are generated, and their ready pairs at the first stage are counted. In the boxplots, the numbers of pairs are divided by $n$ and become fractions. The central mark of a box indicates the median, and the bottom and top edges of the box indicate the 25th and 75th percentiles, respectively. The medians are at least 0.297$n$ for all $n$.

The author does not know the expected number of ready pairs after the first stage. Fig. 4(b) shows the boxplots of the average numbers of ready pairs per stage for $n$ from 20,000 to 400,000. For each $n$, ten random arrangements are generated. For each arrangement, the number $\binom{n}{2}$ is divided by the number of stages, resulting in the average number of ready pairs per stage, which is then divided by $n$ and becomes a fraction. The medians are at least 0.153$n$. If the expected number of ready pairs per stage is $\Omega(n)$, parallel topological sweep will run in $O(n)$ time using $\lceil n/2 \rceil$ processors. Empirical data suggest that there are $3.3n$ stages on average.

5 Implementations

There are three implementations in the literature. The Rosenberger code [3, 12] is difficult to locate. The Miller code [8] has a broken URL. The Rafalin code [11] is downloaded from the Tufts University website [10]. It is slightly revised and brought up to the latest language standard. Two new implementations are developed. A new serial code is implemented in C. The first parallel code is implemented in C and OpenMP. The two new implementations are available on GitHub,
github.com/mingouyang/parTopoSwp. The code is compiled with the Intel C compiler (icc and icpc 19.0.5.281) with the optimization flags -03 -xHost -ipo. The computation is performed on a CentOS 7 server with two Intel Xeon Skylake 2.70 GHz 18-core CPUs and 384GB DDR4 2,666 MHz RAM.

Fig. 4(c) compares the performance of the three implementations. For each implementation at each value of \( n \), the average runtime of ten random arrangements is plotted. The Rafalin code solves \( n = 100,000 \) in 692.2 second. The new serial code solves \( n = 200,000 \) in 742.8 second. On average, the new serial code is 3.92 times faster than the Rafalin code. The parallel code is executed with 64 OpenMP threads. It solves \( n = 400,000 \) in 678.3 second. It is more than 16 times faster than the Rafalin code for large \( n \).

Fig. 4(d) shows the speedup curve of the parallel code using 2, 4, 8, 16, 32, and 64 threads when \( n \) is fixed at 300,000. The baseline is the new serial code — it uses one thread. Speedup is calculated as the serial runtime divided by the parallel runtime. When the number of threads is two and four, the parallel code runs slower than the serial code. This comes from the parallel overhead described in Section 3 as well as the penalty of thread synchronization. At each synchronization point, some threads will be waiting for the others to finish their work. Their idling is another form of parallel overhead. With 64 threads, the parallel code is 4.2 times faster than the serial code. The server has 36 physical cores. The hyper-threading technology of Intel allows two threads to share a physical core. Thus, the hardware may support up to 72 threads. When sharing cores, however, the threads rarely run as fast as when each thread occupies a physical core exclusively. It is likely that if there are 64 physical cores, the parallel code may reach higher performance.

6 Discussion

Topological sweep [3] is a building block of some efficient algorithms. The present work is the first to parallelize it. The classical algorithm processes the ready pairs one at a time. Herein it is shown that ready pairs can be processed in parallel. For random arrangements, it is proved that the expected number of ready pairs at the beginning is \( \Omega(n) \). Empirical data suggest that the average number of ready pairs for the rest of the parallel stages is also \( \Omega(n) \). If this is proved, topological sweep can be done in expected linear time using \( \lceil n/2 \rceil \) processors. Arrangements that constrict concurrency can be constructed. The number of parallel stages is the maximal monotone path length, which is \( \Omega(n^{2-o(1)}) \) [1, 2].

Three implementations are compared. The code by Rafalin et al. [11] is designed to handle degenerate arrangements. A new serial code for simple arrangements is implemented that runs as fast as the author can make it. It will underreport the vertices of degenerate arrangements. When compiled with the same compiler and executed on the same hardware, the new serial code is 3.92 times the speed of the Rafalin code. The speedup is attained by ignoring degeneracy as well as by streamlined computation. Based on the new serial code, the first parallel code is implemented in C and OpenMP. When executed with 64 threads, it is 4.2 times the speed of the serial code and more than 16 times that of the Rafalin code. Both new implementations are available on GitHub. The parallel code is merely a proof of concept. For future work, the code can be sped up with advanced techniques of parallel programming, such as dynamic load balancing and non-uniform memory access tuning.

Graphics processing units have thousands of vector processors. They are designed for massive single-instruction-multiple-data (SIMD) computation. The parallel traversals of the horizon trees loosely fit the SIMD paradigm. The complication is that the bays to be explored have different sizes. Some processors may finish their work before the others. This divergence in computation is a source of SIMD overhead. It is an interesting problem to reorganize the traversals so that such overhead is reduced.
References


