Extending Segment Tree for Polygon Clipping and Parallelizing using OpenMP and OpenACC Compiler Directives

Buddhi Ashan M. K.
Univ. of Texas at San Antonio
San Antonio, USA
buddhiashan.mallikakankanamalage@utsa.edu

Satish Puri
Missouri Univ. of Science and Tech.
Rolla, USA
satish.puri@mst.edu

Sushil K. Prasad
Univ. of Texas at San Antonio
San Antonio, USA
sushil.prasad@utsa.edu

ABSTRACT

A segment tree is a versatile tree-based data structure over intervals or line segments efficiently supporting several computational operations such as stabbing query, segment arrangement, and planar point location, both theoretically and practically. Polygon clipping is a basic operation in domains such as Computer Graphics, Computer-aided Design, and Geographic Information Science (GIS). Given two polygons with n vertices, polygon clipping algorithms find the geometric intersection or union in $O(n^2)$ time using Foster's all-to-all edge intersection testing and $O((n+k)\log n)$ time using Vatti's sweep line-based method, where k is the number of intersections. No known segment tree implementation, including the CGAL library, supports intersection finding or polygon clipping. We extended the segment tree leveraging Chaselle's PRAM-model augmentation, parallelized the construction of our augmented segment tree, and employed it to find line segment intersections for polygon clipping while handling degenerate cases. Augmented segment tree eliminates 99% of non-intersecting edge pairs compared to 63% by the state-of-the-art filtering based on common minimum bounding rectangle method employed in Foster's GPU-based implementation. This, coupled with $\Omega(n \log n)$ work on a single CPU core, beats Foster's GPU performance with $O(n^2)$ work. Our OpenMP directive based multi-core implementation achieves up to 4X relative speedup for clipping two polygons with 182K vertices and 5X speedup for five polygons with 398K vertices. We also offloaded the parallel kernels to a GPU using OpenACC achieving performance competitive with Foster's GPU implementation. Our profiling indicates limitations of the compiler directives and potential for superior performance by employing pthread/cuda libraries.

CCS CONCEPTS

• **Computing methodologies** → **Shared memory algorithms**.

KEYWORDS

Segment tree, Polygon clipping, Degenerate cases for intersection, Foster's algorithm, Vatti's algorithm, OpenMP, OpenACC, computational geometry

ICPP '24, August 12–15, 2024, Gotland, Sweden

ACM ISBN 978-1-4503-XXXX-X/18/06
https://doi.org/XXXXXXX.XXXXXXX

ACM Reference Format:
Buddhi Ashan M. K., Satish Puri, and Sushil K. Prasad. 2024. Extending Segment Tree for Polygon Clipping and Parallelizing using OpenMP and OpenACC Compiler Directives. In *Gotland '24: 53rd International Conference on Parallel Processing, August 12–15, 2024, Gotland, Sweden.* ACM, New York, NY, USA, 11 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

A segment tree is a static binary tree data structure used to store intervals or segments efficiently [5, 19]. In literature, the segment tree is used as a data structure for many computational geometry algorithms such as stabbing query, line arrangement, and planar point location [14, 16, 23]. A stabbing query takes a point as an input and returns the set of intervals that contain the given point. Line arrangement is the subdivision of the plane formed by a collection of lines. In planar point location, given a partition of the space into disjoint regions, we have to determine the region where a query point lies [19]. A polygon is a collection of line segments and in this work, these line segments of the polygons are used to build a segment tree. In Fig. 1a, two example polygons are shown. A segment tree constructed from the segments of these two polygons is shown in Fig. 1b. For n line segments, a segment tree can be built in $O(n \log n)$ time and space [8].

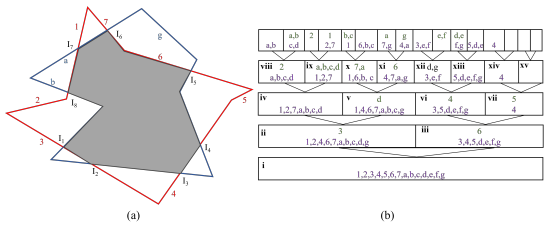

Figure 1: (a) The base polygon is denoted in blue and the clipping polygon is denoted in red. There are seven edges in both polygons. I_1, I_2, I_3, I_4, I_5, I_6, and I_7 are the line segment intersection points. The grey region is the result of the polygon clipping. (b) The corresponding segment tree constructed using the base and clipping polygons.

Polygons are used to represent boundaries of regions (shapes) in Geographic Information Science (GIS) and Computer Graphics domains. Geometric set operations such as intersection, union, and set difference on very large polygonal datasets are common and important in these domains. Polygon clipping is the calculation of $B \cap C$ between two polygons B and C producing the common region(s) between the input polygons (Fig. 1a). General polygon clipping algorithms can handle all types of polygons. They can be

easily modified to compute other geometric set operations such as union and set difference [11].

Greiner-Hormann (GH) algorithm and Vatti's algorithm are two well-known algorithms for arbitrary polygon clipping [15, 31]. Both algorithms first discover line segment intersections, then label them, and finally trace the results with the help of the labels. The dominating computational phase in these algorithms is the line segment intersection discovery. GH algorithm uses a brute force approach to report the intersections in $O(n^2)$ time, where n is the total number of vertices in the input polygons. Vatti's algorithm uses scan beams to decompose the line segments and finds intersections in $O((n+k)\log n)$ time, where k is the number of line segment intersections. Foster's polygon clipping algorithm is an extension of the GH algorithm with proper degenerate case handling which is limited in the GH algorithm [3, 11].

The objective of this work is to extend the segment trees to handle geometric intersection of polylines and polygons and explore portable parallelization on multi-core CPUs via OpenMP compiler directives. With our practical extension, now segment trees can be used to compute geometric intersections while handling degenerate cases in addition to stabbing queries, window queries, etc [6, 23].

Parallel pair-wise line segment intersection is a core component in the polygon clipping algorithms. Goodrich presented an output-sensitive CREW PRAM algorithm leveraging an extension of the segment tree to discover line segment intersections in $O(\log n)$ time using $O(n \log n + k)$ number of processors where k is the number of intersections [14]. In the worst-case scenario $k = \Theta(n^2)$. It leverages Chaselle's rule for finding line segment intersections in the segment tree [8]. Goodrich's PRAM algorithm is not suitable for an efficient multi-core implementation for real-world polygon clipping. The real-world polygons involved in clipping have at most a few hundred thousand edges, therefore segment tree nodes are comparatively small in terms of the number of line segments contained in them and parallel overheads dominate.

In this work, we leverage Goodrich's theoretical work in a practical setting utilizing it to discover intersections in polygon clipping efficiently. First, we build an augmented segment tree by inserting the segments of the input polygons into the tree nodes, with special handling for vertical segments. The nodes of the tree maintain *cover lists* as well as *end lists* containing suitable subsets of the line segments of the input polygons. Next, the output line segment intersection points are computed using the *cover-list* and the *end-list* of the nodes of the tree using Chaselle's rule. It should be noted that the *cover lists* are part of the standard segment tree and *end lists* are additionally needed to find the line segment intersections for polygon clipping. Finally, Foster's labeling and result tracing steps are employed to calculate the resulting intersecting polygon(s).

The main contributions of this work are as follows.

- The first work to extend segment tree data structure for polygon clipping based on Chaselle's rule and evaluate the augmented segment trees experimentally.
- Parallelized augmented segment tree construction and line segment intersection finding on multicore CPUs to handle polygon clipping using OpenMP directives.
- Effective edge pair filtering employing the segment tree resulting in the elimination of 99% of the non-intersecting edge pairs on an average as compared to 63% with common minimum bounding rectangle (CMBR) based filtering employed in the state-of-the-art Foster GPU implementation [3]. This yields a single core CPU performance of segment-tree based polygon clipping with $\Omega(n \log n)$ work beating the state-of-the-art many-core GPU performance of Foster's clipping with $O(n^2)$ work.
- C++ OpenMP based multi-core implementation yields up to 4X speedup over real-world datasets, processing two polygons with a total of 182K vertices on a Xeon Silver 4210R CPU, compared to the sequential segment tree-based polygon clipping algorithm running on a single CPU core.
- One-to-many polygon clipping utilizing a single segment tree to discover intersections among multiple polygon pairs enables 5X relative speedup.

The rest of the paper is organized as follows. Section II introduces the background on polygon clipping algorithms, segment tree and its augmentation, and segment tree-based line segment intersection. Section III presents our practical polygon clipping algorithm based on a parallel augmented segment tree and implementation details. Section IV contains an experimental evaluation of the parallel polygon clipping algorithm. Section V presents our conclusions and future work.

2 LITERATURE AND CONCEPTS

2.1 Polygon Clipping

A polygon is a collection of at least three points in the 2-dimensional space connecting to construct a closed region. The points are referred to as vertices. Each adjacent pair of vertices connects using a straight line segment referred to as an edge.

There are different types of polygons. 1) *simple* polygon where its edges do not self-intersect, 2) $self\text{-}intersecting$ polygon where at least one edge intersects another edge, 3) *convex* polygon where all interior angles are no more than 180°, and 4) *concave* polygon where some interior angles are greater than 180°.

Polygon clipping involves the computation of line segment intersections. An intersection point is generated when a segment/edge from a polygon overlaps or crosses a segment from another polygon. There are well-known sequential and parallel polygon clipping algorithms in Computer Graphics and GIS domains. Maillot's algorithm only clips using a rectangle, but not against polygons [18]. Sutherland-Hodgman, Weiler-Atherton, Liang-Barsky, Vatti's, and GH algorithms can clip a concave polygon against another concave polygon [15, 17, 30–32]. Vatti's and GH algorithms can also clip arbitrary polygons [15, 31].

In addition to Vatti's algorithm, plane-sweep based sequential polygon clipping algorithms are also discussed in [20, 22, 28]. The GH algorithm has a simpler way to represent polygons than Vatti's and its time complexity is not output sensitive. Foster et al. present an extension to the GH algorithm with the ability to handle degenerate cases properly [11].

There are parallel clipping algorithms based on [17, 30] implemented on classic parallel architectures [26]. Naïve $O(n^2)$ algorithms and grid partitioning have been used in practical GPU overlay algorithms discussed in [4, 12]. There is a multi-core Vatti's algorithm implementation presented in [24]. Parallel many-core

and multicore implementations of the GH algorithm are presented in [25]. Both GPU clipping algorithms discussed in [4, 25] are unable to handle degenerate cases (see Fig 6). Parallel many-core implementation of Foster's algorithm is presented in [3] uses common minimum bounding rectangle (CMBR) based filtering and line segment minimum bounding rectangle (MBR) based filtering. CMBR filter only focuses on the line segments that are within or intersect the common intersection area between the two polygonal MBRs [3] (see Fig. 2a). Hence, those line segments that are not part of the common MBR are filtered out from further processing. In the refinement phase, before performing the line segment intersection test for a pair of line segments, those pairs are further weeded out whose MBRs do not overlap [3] (see Fig. 2b and 2c). This is referred to as the line segment MBR (LSMBR) filter in [3].

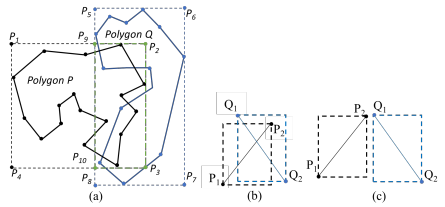

Figure 2: (a) Common minimum bounding rectangle (CMBR) filter: MBR^P=[P_1, P_2, P_3, P_4], MBR^Q=[P_5, P_6, P_7, P_8]. $CMBR$=[P_9, P_2, P_3, P_{10}]. Only those edges of P and Q that intersect CMBR are used for further processing, LSMBR filter: (b) Possible intersection, (c) no intersection possible [2, 3].

2.2 Segment Tree

A segment tree is a static full and complete binary tree data structure used to store intervals or segments efficiently [5, 19]. Segment tree construction runs in $O(n \log n)$ time using $O(n \log n)$ storage, where n is the number of intervals. Given a collection of line segments, a segment tree is built in two stages. In the first stage, the skeleton of the tree is built by sorting the x-coordinates of the endpoints of the line segments in order to generate the leaf nodes where each node represents an elementary interval. The interior nodes are then built using a bottom-up approach by the union of the interval of their child nodes. In the second stage, all the input line segments are inserted into the tree.

Assume a set E of n intervals. Let $p_1, p_2, ..., p_n$ be the x-coordinates of the segments' endpoints sorted in non-decreasing order. These endpoints define the ***elementary intervals*** corresponding to leaf nodes of a segment tree. The *elementary intervals* for this set of x-coordinates are as follows: $(-\infty, p_1)$, $[p_1, p_1]$, (p_1, p_2), $[p_2, p_2]$, (p_2, p_3), ..., (p_{n-1}, p_n), $[p_n, p_n]$, $(p_n, +\infty)$ [19]. Let T denote a segment tree and let v denote a node in T. Let $\mathbf{\Pi_v}$ denote the interval of node v. For an internal node $v \in T$, Π_v denotes the union of all the intervals of its child nodes. For a node interval representing a vertical segment with starting and ending points with the same x-coordinates, $\Pi_v = [p_i, p_i]$, where $1 \leq i < n$. These point intervals are always at the leaf level of the segment tree and do not become internal/parent nodes.

Parallel and distributed segment tree algorithms are presented in [7, 9, 13, 16, 23, 27, 29, 33]. Dehne and Chaplin presented a segment tree implementation on a hypercube [9]. Su presented theoretical and practical segment tree construction algorithms that run in polylogarithmic time [29]. Chan et al. presented a distributed implementation of the segment tree [7]. Gerbessiotis presented a bulk-synchronous parallel (BSP) segment tree construction algorithm [13]. Zheng et al. presented a distributed segment tree for efficient query processing [33]. Shen et al. presented a distributed segment tree supporting both range query and cover query [27]. Jaja discusses a PRAM segment tree construction algorithm [16].

2.3 Chaselle's Observation

Chaselle observed that a segment tree can be used for line segment intersection by augmenting its nodes with an *end-list* at each node [8]. This work is further improved by Goodrich [14].

Chaselle's observation involves a *cover-list* and an *end-list* at each segment tree node. A segment e_i with endpoints (x_i, y_i) and (x_i', y_i') ***spans*** the interval Π_v of node v if the interval (x_i, x_i') is a superset of the interval Π_v. A segment e_i ***covers*** a node $v \in T$ if e_i spans Π_v but not $\Pi_{parent(v)}$, that is, v is the highest such node that e_i spans [14]. All such segments are in the *cover-list* of node v.

$$\textbf{\textit{cover-list}}(v) = \{e \in E \mid e \text{ covers } v\} \tag{1}$$

An edge can be assigned to no more than 2 nodes at any level of the tree per this definition of cover list [19].

A segment e_i belong to the *end-list* of a node $v \in T$ if e_i does not span Π_v but Π_v contains an endpoint x_i or x_i' of e_i [8], that is one of the end points of e_i lies within Π_v.

$$\textbf{\textit{end-list}}(v) = \{e \in E \mid e_i \textit{ does not span } \Pi_v \textit{ but } \Pi_v \textit{ contains an endpoint of } e_i\} \tag{2}$$

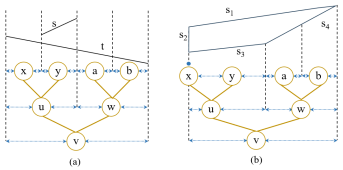

Figure 3: (a) Line segments s, t, and their related segment tree. (b) A polygon with edges s_1, s_2, s_3, and s_4 and its segment tree. Node x represents vertical edge s_2, which is a point interval. Nodes a and b are empty nodes added to complete the tree.

Fig. 3a depicts segment s and a segment tree. We illustrate related concepts for clarity. The segment tree is only sensitive to a single dimension of values. But a line segment's endpoints have both x and y coordinates. Hence, the segment tree uses only the x coordinates of the endpoints to define the aforementioned *cover-lists* and *end-lists*. The tree node intervals are represented in dotted lines. Nodes a and b are empty nodes added to complete the tree. Segment s does not span nodes x, u, w, or v, but their intervals contain endpoints of s. Segment s *covers* node y. Segment s gets added to End(x), End(u), End(w), End(v), and Cover(y).

Fig. 3b depicts a polygon with edges s_1, s_2, s_3, and s_4 and the corresponding segment tree. Node x has a point interval since it represents vertical segment s_2. Segment s_1 *covers* root node v since

it does not have any parents. Since this segment *covers* root node, it does not belong to any *end-list*. Segment s_2 *covers* node x since it cannot span node u interval (parent(x)). It is in the *end-list* of node y since its endpoints are at the border of that node interval and it does not span node y. Segment s_3 *covers* node u since it does not cover the parent node of u. Segment s_3 is in the *end-list* of w since it is on the border of that node's interval and does not span the interval. Segment s_4 *covers* node w and it is in the node y's *end-list*. The *end-list* of point interval nodes are always empty since a point is sufficient to cover its interval and by doing so, it violates the *end-list* definition (see equation 2).

In our implementation of the augmented segment tree to utilize Chaselle's observation described next, we used closed intervals at the leaf level nodes of the tree avoiding point interval nodes unless they represent a vertical segment. Hence our version of *elementary intervals* is as follows: $[p_1, p_2], [p_2, p_3], ..., [p_{n-1}, p_n]$, resulting in some duplicate edges at neighboring *end lists*, but not missing any. This approach helps to handle vertical edges properly and, in post-processing phase, duplicates are eliminated if there are any.

Chaselle observed that there is a unique node $v \in T$ that discovers any segment pair e_1 and e_2 intersection, making one of the following rules true, where the intersection point is within Π_v.

- $e_1, e_2 \in cover\text{-}list(v)$
- $e_1 \in end\text{-}list(v)$ *and* $e_2 \in cover\text{-}list(v)$
- $e_2 \in end\text{-}list(v)$ *and* $e_1 \in cover\text{-}list(v)$

Thus, finding intersections can be limited to pairwise searches within the *cover-list* and cross-pairs searches between the *cover-list* and the *end-list*, and these can be independently performed for each node.

3 METHODOLOGY

A general polygon clipping algorithm consists of line segment intersection discovery, intersection labeling, and result tracing, where the dominating step is line segment intersection discovery [3, 11, 15, 25]. Our alternative approach has three major steps: Segment tree construction, intersection discovery, and labeling and tracing results. We employ our augmented segment tree to decompose the polygon edges to find the intersections faster, replacing the trivial brute-force all-to-all quadratic-time edge test. We introduce additional post-processing to save the intersections in the correct edges of the original polygons in the order they appear. Afterward, Foster's labeling and result tracing steps are employed to calculate the resulting intersecting polygon(s), handling degenerate inputs.

3.1 Augmented Segment Tree Construction

Terminology: We use the following terminology to explain polygon clipping. Let the input base polygon B and the clipping polygon C consist of edge sets E^B and $E,^C$word respectively. Let the vertex sets of the input polygons be V^B and V^C, respectively. Let the resultant polygon after clipping be P. Let the set of x coordinates in sorted order are $X = \{x_1, x_2, ..., x_i, ..., x_l\}$, where $X = V^B \cup V^C$ and $l = |E^B| + |E^C|$. These define the set of intervals of the input polygons with $interval_i = [x_i, x_{i+1}]$. The segment tree is a full and complete binary tree represented in a heap-like array of $2l-1$ nodes. Logically, each node in this tree has a structure as depicted in Fig. 4.

```
struct{
    real intervalStart
    real intervalEnd
    int coverList[]
    int endList[]
}Node
```

```
struct{
    Node node[]
}segmentTree
```

Figure 4: Segment tree data structure.

However, for implementation efficiency, we use three 2-D arrays: *ST-NodeIntvl*, *Cover-list*, and *End-list* to save node interval, *cover-list*, and *end-list* of each tree node, respectively. Each row of these arrays represents a node and its columns store the relevant node intervals, edge IDs of the *cover-list*, and edge IDs of the *end-list*, respectively.

Algorithm 1 - Parallel Augmented Segment Tree Construction

Input: Edge lists from input polygons B and C ($E,^B$ E^C)
Output: *ST-NodeIntvl*, *Cover-list*, *End-list*

1: Read *intervals* from the x-coordinates of E^B and E^C.
2: Construct *elementaryIntervals* and the leaf level *ST-NodeIntvl*.
   a. Sort *intervals*.
   b. Remove duplicates from *intervals*. Add both start and endpoints when an *interval* is a point.
3: Construct the *ST-NodeIntvl* (internal node intervals) by union of children yielding parent's interval, starting with leaf level intervals.
4: Construct *Cover-list* by finding covered nodes for each edge traversing from the root downward.
5: Construct *End-list* by populating the *end lists* of leaves and merging level by level weeding out duplicates.

Algorithm 1 outlines the steps involved in the parallel segment tree construction. Step 1 identifies *intervals* from the input polygons. An interval is computed using the x-coordinates of two consecutive vertices. Step 2 computes the *elementary intervals* using the *intervals* calculated before. The *elementary intervals* is the list of unique endpoints of the *intervals*. However, when a particular interval is a point (when start == end), both start and end points are inserted in this list. First, the *intervals* are sorted (Step 2a). Next, the duplicates are removed unless the points belong to a point interval (Step 2b). The elementary intervals are used to construct the leaf level intervals of the segment tree. **Algorithm 2** sketches the bottom-up approach used to build the intervals of the segment tree. The levels are visited from leaf to root level. In Step 3, the intervals of the internal nodes of the segment tree are constructed using the leaf-level node intervals. Steps 1 - 3 are also referred to as segment tree skeleton construction in our run time evaluations. In **Algorithm 2**, at a given level, the node intervals can be calculated in parallel. However, parent-level nodes depend on their child-level nodes in order to calculate their own intervals. Hence, we only parallelized the inner for loop and at the end of that loop, we used a *barrier* directive to synchronize all threads.

Algorithm 2 - Build Tree Intervals

Input: *ST-NodeIntvl*: Indexed array of leaf level intervals
Output: *ST-NodeIntvl*: Node intervals for internal nodes of the tree

```
1: for each level in Tree do
2:     for each node in level do in parallel
3:         rightChild = 2 * node
4:         leftChild = 2 * node + 1
5:         interval.start = rightChild.start
6:         interval.end = leftChild.end
7:         segmentTreeNodeIntervals[node] = interval
```

Steps 4 and 5 of **Algorithm 1** construct a *cover-list* and an *end-list* at each node respectively. These data structures are independent of each other and can be built simultaneously. **Algorithm 3** lays out the steps in the parallel *Cover-list* construction using input polygon edges (E) and tree node intervals (*ST-NodeIntvl*). *Cover-list* is a 2-D array used to store the *cover-list* of each node. For each edge insertion, a recursive search is used to find the grandparent(s). The race condition when storing an edge is handled using section locks. We employ a *lock* per node to maximize concurrent insertions.

This recursive approach is unsuitable for the GPU. Instead, we built *cover lists* at each node scanning a candidate edge set. For any *left child*, we only scan the edges whose max points lie between the node and its parent's max interval boundaries. For any *right child*, we only scan the edges whose min points lie between the node and its parent's min interval boundaries. This enabled simultaneous construction of all nodes, albeit with more overheads, for an OpenACC implementation.

Algorithm 3 - Build Cover-lists

Input: $E = \{E^B \cup E^C\}$, *ST-NodeIntvl*
Output: All *Cover-list*

```
1: for each e in E do in parallel start from root node v
2:     if interval of node v is contained in e then
3:         store e at v atomically
4:         exit
5:     else
6:         if e ∩ Π_Left−Child(v) then
7:             insert e in that subtree recursively
8:         if e ∩ Π_Right−Child(v) then
9:             insert e in that subtree recursively
```

Step 5 of **Algorithm 1** constructs an *end-list* at each node unless the node is a point interval. We use 2-D array *End-list* to save the *end-lists* of the nodes of the entire segment tree. The *end-lists* at point interval nodes are always empty since the *end-list* definition does not allow any segment that spans a node interval. **Algorithm 4** sketches the bottom-up approach used to construct *End-list* array. For each *elementary point*, their left and right neighboring node IDs are saved in arrays. In leaf level *end-list* construction, these arrays are used to access the left and right neighbors of a given interval point in $O(1)$ time. All polygon edges (E) are inserted in the right node of the start vertex of an edge and in the left node of the end vertex unless the nodes are point intervals. The *end-list* of a parent

Algorithm 4 - Build End-lists

Input: $E = \{E^B \cup E^C\}$, *ST-NodeIntvl*
Output: *End-list*

```
1: for each ep in elementaryPoints do in parallel
2:     rightPointMap ← right node of ep
3:     leftPointMap ← left node of ep
4: for each e in E do in parallel
5:     insert e in End-list at rightPointMap[e.end]
6:     insert e in End-list at leftPointMap[e.start]
7: for each level in tree (leaf to root) do
8:     for each v in level do in parallel
9:         p = index of v
10:        leftChild = 2p
11:        rightChild = 2p + 1
12:
            End-list[p] = End-list[leftChild] ∩
                          End-list[rightChild]
```

node is calculated by performing a union operation on the child *end-lists*. Hence, the total *end-list* size at each level is no more than the total *end-list* size at the leaf level of the tree. This allows us to allocate enough memory in the *End-list* array to save *end-lists* of the internal nodes avoiding the need for memory copying of leaf level *end-lists*.

The union operation is performed in parallel at each level of the tree. But each level needs to finish its *end-list* construction in order to move to the next level due to the dependency between the parent and child nodes. The point interval nodes are always assigned at the leaf level of the tree, thus this operation does not need to handle them explicitly.

Our experiments showed that the nested loop at Step 9 of **Algorithm 4** degrades performance when offloaded in the GPU due to the complex computation including branching and loops. Hence we performed this computation over the multi-core environment employing OpenMP.

Fig. 5 illustrates an example of a segment tree built for a pair of polygons. The segment tree has 13 elementary intervals and 3 empty intervals are added to complete the tree. Each node has a *cover-list* and an *end-list* that are shown in green and purple respectively. The intersections are I_1, I_2, I_3, I_4, I_5, I_6, I_7, and I_8 (see Fig. 5). They are displayed pointing at the nodes where each is uniquely found. Edges 3 and d intersect at node 2 producing I_2 which is also found at node 12. But I_2 is only contained at node 2. Therefore, node 12 does not report it as an intersection to avoid duplicates. Similarly, intersections I_1 - I_8 are reported at unique nodes.

3.2 Intersection Finding

Algorithm 5 sketches the steps in intersection calculation. Step 1 discovers intersections utilizing Chaselle's observation using local *cover-list* and *end-list* data, performing intersection tests between the edges from the *cover-list* itself and against the edges from the *end-list*. For efficient intersection finding in parallel, we employed *dynamic scheduling* in the Step 1 outer loop with a chunk size of 100. We also leverage the LSMBR filtering presented in [3] to

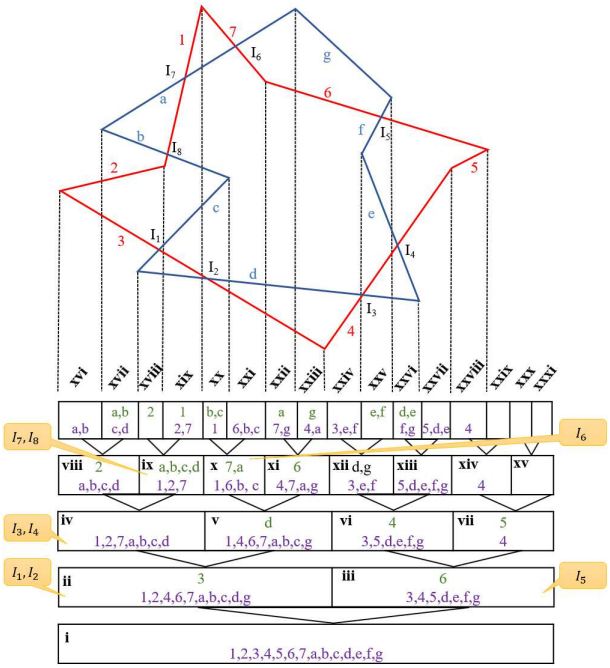

Figure 5: Polygon clipping using an augmented segment tree. The base polygon is denoted in blue and the clipping polygon is denoted in red. There are 7 edges, each in the base and clipping polygons. Leaf level node intervals are labeled xvi - xxxi and the internal node intervals are labeled i - xv. The *cover-list* elements at each node are in green letters and the *end-list* elements are in purple letters. Intersections are shown pointing at the nodes where they are found. I_1, I_2, I_3, I_4, I_5, I_6, I_7, and I_8 are intersections.

Algorithm 5 - Parallel Intersection Finding

Input: *ST-NodeIntvl*, *Cover-list*, *End-list*
Output: $E,'^B$ $E,'^C$ $\alpha,^B$ $\beta,^C$ $Neighbors,^B$ $Neighbors^C$ arrays

1: **for each** node v in Tree **do in parallel**
2: **for each** edge pair within *cover-list* **do**
3: Find intersections.
4: Save intersection types and edge IDs of the input polygons.
5: **for each** edge pair across *cover-list* and *end-list* **do**
6: Find intersections.
7: Save intersection types and edge IDs of the input polygons.
8: Sort intersecting edge IDs of the base polygon.
9: Sort intersecting edge IDs of the clipping polygon.
10: Save intersections and their α, β values.
11: Sort intersections by α, β values separately.
12: Copy the non-degenerate intersections in the base and clip polygons at the correct locations (E'^B and E'^C). Mark degenerate intersections in the source vertex list.
13: Copy α, β values to $\alpha,^B$ β^C. Save clipping polygon edge id in $Neighbors^B$ and base polygon edge id in $Neighbors^C$

identify the edge pairs whose MBRs intersect with each other before employing much more expensive intersection finding. In this step, the intersection type, base polygon edge id, and clipping polygon edge id are stored in three arrays.

Step 11 constructs an index array in the sorted order of the intersections by the base polygon edge IDs. Step 12 produces a similar index array, but sorted by the clipping polygon edge IDs. The sorted index arrays help to insert the intersections in the source polygons' edges efficiently. Step 13 saves the intersection coordinates, α, and β values in three arrays. The ***α* value** is the distance ratio from the parent vertex of the contributing edge of the base polygon to an intersection. ***β* value** is defined similarly, but uses the clipping polygon's contributing edge [11].

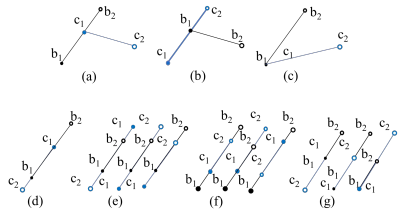

Figure 6: Degenerate intersection types [3, 11]. (a) and (b) T-intersections. (c) V-intersection. Degenerate overlap types [3, 11]. (d) X-overlap. (e) and (f) T-overlap. (g) V-overlap.

Step 14 produces two sorted index arrays by α and β values. Using these sorted index arrays, Step 15 copies the intersections, α, and β values in $E,'^B$ E'^C arrays marking degenerate cases. Fig 6 shows a summary of all degenerate cases. Step 16 copies the non-degenerate intersections into source polygons along with α and β arrays. The degenerate intersections update the corresponding vertices in the input polygons [11]. This step also builds $Neighbors^B$ and $Neighbors^C$ arrays. These are used to find the contributing edge id of an intersection from the other polygon.

3.3 Intersection Labeling and Tracing the Results

Algorithm 6 - Labeling and Tracing Results

Input: $E,'^B$ $E,'^C$ $\alpha,^B$ $\beta,^C$ $Neighbors,^B$ $Neighbors,^C$ arrays
Output: contour of output polygon(s)

1: Initial labeling in parallel.
2: Copy initial labels and intersection arrays into linked lists. Perform other labeling on the linked lists.
3: Trace results.

Algorithm 6 outlines the steps in intersection labeling and tracing to construct the contour of the resulting polygon(s) using the intersections, their α, β, and neighbor values that were calculated in **Algorithm 5**. The initial labels are calculated in parallel since the α, β, and neighbor values are locally available at each intersection vertex. The rest of the intersection labeling uses serial Foster's algorithm labeling and result tracing steps.

Foster's algorithm uses *entry*/*exit* label based on the inside/outside status of polygonal edges with respect to another polygon. There are multiple stages of labeling. In the initial labeling, the non-overlapping intersections are labeled *Crossing* or *Bouncing* considering the relative location of the contributing edge from the clipping polygon. Data for initial labeling are locally available at each edge and can be done in parallel.

The second stage labels the intersection chains starting with a turn label. Turn labels handle chains of degenerate intersections properly since they all have the same *entry*/*exit* label. In the third stage, final *entry*/*exit* labels are given using second stage labels. Step 2 then copies the labels in the source polygons and traverses them to construct the resulting polygon (s). These steps consume a very small percentage of the total run time thus parallelization is not practical.

3.4 One-to-Many Polygon Clipping

There are two observations of the one-to-one polygon clipping approach: (1) segment tree construction which consists of tree skeleton, *cover-list*, and *end-list* constructions is the most expensive phase (see Fig. 11). (2) this method is inefficient when using smaller inputs. To mitigate these issues, we investigated one-to-many (1-to-M) polygon clipping leveraging a segment tree. 1-to-M clipping is defined as follows. Assume a base polygon B and a set of clipping polygons $CL = \{C_1, C_2, ..., C_l\}$ where $l > 1$. 1-to-M clipping calculates $B \cap CL$. **Algorithm 7** sketches the steps in 1-to-M clipping leveraging a segment tree to handle multiple polygon pairs.

Algorithm 7 - One-to-Many Polygon Clipping

Input: Edge lists from input polygons B and CL
Output: contour of output polygons

1: Construct a segment tree S using segments from B and CL.
2: **for each** C in CL
3: Find intersection of $B \cap C$ using *Cover-list* and *End-list* of S **in parallel**.
4: Create $E,'^B$ $E,'^C$ $\alpha,^B$ $\beta,^C$ $Neighbors,^B$ $Neighbors^C$ arrays.
5: Label and trace results using $E,'^B$ $E,'^C$ $\alpha,^B$ $\beta,^C$ $Neighbors,^B$ $Neighbors^C$ arrays.
6: Create output polygons.
7: **end for**

In Step 1, a segment tree is constructed in parallel leveraging **Algorithm 1** using all edges of the input polygons. The base polygon is common for all clipping calculations. Instead of constructing multiple segment trees for each clipping polygon pair, this approach uses a single segment tree, where the edges of the base polygon are shared among multiple clipping polygons to calculate the clipping results, amortizing the cost of segment tree construction. In Step 2, each clipping layer polygon is clipped against the base polygon utilizing **Algorithms 5** and 6. The resulting clipping polygons of each pair are reported as the 1-to-M output.

4 EXPERIMENTAL RESULTS

4.1 Testbed

Our workstation is equipped with an Intel Xeon Silver 4210R CPU running on 2.40GHz with 10 cores with 64 GB of memory, and an Nvidia Quadro RTX 5000 GPU card with 16 GB of VRAM, 48 SMs, and 3072 CUDA cores. We used OpenMP and OpenACC compiler directives and C++ for our implementation. We used C++ CUDA implementation of Foster's GPU polygon clipping algorithm, downloadable from the Github repository at https://github.com/buddhi1/GH-CUDA.

4.2 Datasets

Table 1: Real-world datasets and polygon clipping characteristics [1, 10, 21].

#	Datasets	|Base|	|Clipping|	|Result|	Intersect count
1	lakes_174690, parks_321571	102,721	79,686	21,501	228
2	lakes_174690, parks_169840	102,721	81,897	54,214	320
3	lakes_174690, parks_140315	102,721	79,602	44,004	106
4	lakes_174690, parks_34622	102,721	54,992	27,401	212
5	Classic S, C	101,242	72,997	50,312	47
6	ne_10m_ocean(0), continents(521)	100,612	16,205	37,608	10,082
7	ne_10m_ocean(0), continents(1661)	100,612	12,613	16,895	1,427

We used two real-world polygon datasets for performance evaluation. The dataset consists of selected large polygons extracted from real-world geospatial datasets: lakes, parks [10], Classic polygon pair [21], ocean (ne_10m_ocean), and Continents [1]. (see Table 1).

Table 1 reports the polygon IDs, their sizes, resulting clipped polygon sizes, and the intersection count for each dataset. Datasets 1-4 consist of a large polygon from the lakes dataset and 4 large polygons from the parks dataset. The four polygons from the parks dataset were translated in the 2-D space to discover a reasonable number of segment intersections against the large polygon from the lakes dataset. On average, the segment tree size is 2.4 times compared to the number of input edges. On average, 70% of the segment tree nodes contribute to intersection finding and the rest 30% are there to make the tree binary complete.

Table 2: One-to-many datasets and Polygon Clipping Characteristics [1, 10].

#	Datasets	|Base|	|Clipping|	|Result|	Intersect count
8	lp1	102,721	296,177	147,120	866
9	lp2	196,726	296,791	46,984	290
10	lp3	206,429	236,095	28,641	114
11	oc1	100,612	28,818	54,503	11,509

The second real-world dataset is used to evaluate the performance of the one-to-many approach (see Table 2). The base polygons of Datasets 8 to 10 are chosen from the lakes dataset. Multiple

clipping polygons are selected from the parks dataset. The polygons from the parks dataset were translated to discover an ample amount of segment intersections against the base polygon from the lakes dataset in 2-D space. Datasets 1-4 in Table 1 are merged into dataset 8 in Table 2. Datasets 6 and 7 in Table 1 are merged into dataset 11 in Table 2. All datasets have multiple clipping polygons to clip against the base polygon. These datasets will be publicly shared for reproducibility.

4.3 Augmented Segment Tree Edge Filtering

Table 3: The segment tree edge pair filtering performance against CMBR filter of Foster's GPU clipping algorithm.

#	Total candidate edge pairs	CMBR filter	Using segment tree	% eliminated with CMBR filter	% eliminated with segment tree
1	8,185M	3,072M	21,177K	62.47%	99.74%
2	8,413M	1,078M	18,584K	87.19%	99.78%
3	8,177M	788M	18,079K	90.36%	99.78%
4	5,649M	1,679M	17,533K	70.28%	99.69%
5	7,390M	6,074M	4,118K	17.82%	99.94%
6	1,630M	1,142M	6,026K	29.97%	99.63%
7	1,269M	186M	5,539K	85.34%	99.56%

For a given real-world pair of polygons, the intersecting edge pair percentage is small [3]. Eliminating non-intersecting edge pairs using fast filters helps to optimize the overall clipping efficiency. The GPU-based parallelization of Foster's algorithm uses a CMBR filter to eliminate non-intersecting edge pairs by eliminating the edges that do not fall inside the CMBR of the input polygons. In the segment tree based polygon clipping, the candidate edge set for intersection finding is limited to the edge pairs that satisfy Chaselle's rules. To further optimize, we employed the LSMBR filter from Foster's GPU implementation to prune non-intersecting edge pairs. This filter is an inexpensive operation based on MBR intersection to eliminate non-intersecting edge pairs further. In this section, we compare the performance of the CMBR filter vs the segment tree based edge filter. We do not consider the LSMBR filter since both algorithms use it to further refine the filtered results.

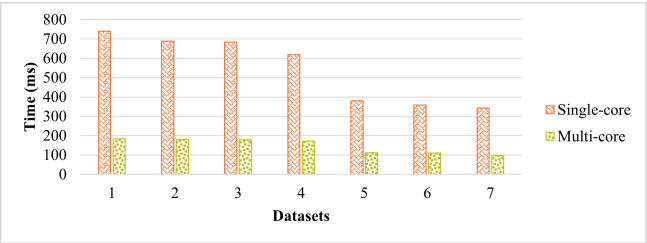

Figure 7: Performance of the multi-core algorithm over real-world dataset (with 10 OpenMP threads, excluding I/O times).

The segment tree-based clipping eliminates most of the non-intersecting edge pairs at the tree nodes. Table 3 reveals the actual edge comparisons using the segment tree compared to the trivial brute force approach. We observe that the segment tree can eliminate 99% of the non-intersecting edge pairs. On average, CMBR filtering in Foster's GPU implementation eliminates 63% of the edge pairs ranging from 18% to 90%. The superior filtering of the segment tree achieves a sequential execution time on par with or better compared to Foster's GPU polygon clipping (see Fig 7 and 13). For example, for Dataset 1, sequential and Foster's execution times are around 700 ms and for Dataset 5, sequential time is below 400 ms while Foster's is more than 800ms.

4.4 Multi-core/OpenMP Performance

We compare the execution times of our C++ sequential segment tree-based clipping against the C++ OpenMP multi-core directive based implementation over the real-world dataset using 10 threads as shown in Fig 7 (excluding I/O times). The multi-core execution attains an average of 3.6X speedup over all cases compared to the sequential version over the real-world dataset with a maximum speedup of up to 4X. These indicate that the overhead of building a segment tree can be practically offset by its filtering performance and OpenMP based parallelization.

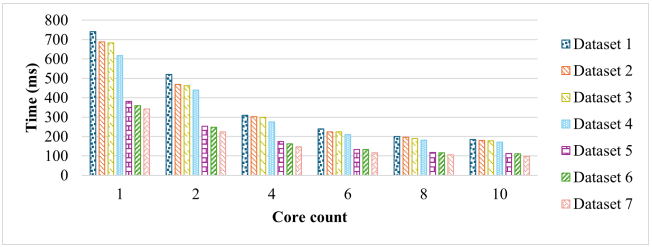

Figure 8: Execution times for different numbers of cores.

Fig. 8 shows run times of our multi-core algorithm using OpenMP for increasing numbers of cores over real-world datasets. However, our multi-core implementation achieves an average 3.7X speedup against the state-of-the-art Foster's GPU algorithm over the real-world dataset with a maximum speedup of 7.8X on Dataset 5. This shows the superior non-intersecting edge pair filtering capability of the segment tree approach practically, beating a GPU algorithm by a CPU algorithm only using compiler directives (see Fig 13).

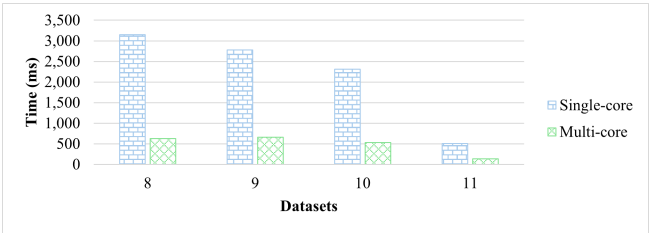

Figure 9: Multi-core performance of 1-to-M polygon clipping (excluding I/O times) employing 10 threads.

In our experiments, we do not compare our approach against the Clipper library which is designed solely for polygon clipping with little overhead and performs better for clipping polygons as opposed to our implementation that extends general segment tree to handle polygon clipping in addition to its other existing use cases.

4.5 Multi-core One-to-Many Polygon Clipping

We compare the serial 1-to-M polygon clipping approach against the multi-core 1-to-M polygon clipping approach as shown in Fig 9. Since our 1-to-M approach does not build segment trees for each polygon pair, the total run time in this dataset is reduced by 25%. The multi-core 1-to-M approach achieves up to 5X speedup against its serial counterpart. Fig 10 depicts run times of our multi-core algorithm using OpenMP directives for increasing numbers of core counts over the one-to-many dataset.

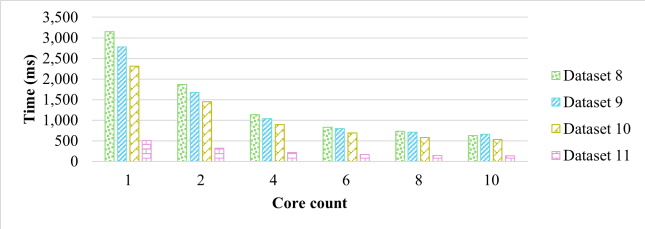

Figure 10: 1-to-M clipping times over for different number of cores.

4.6 Multi-core Run Time Profile

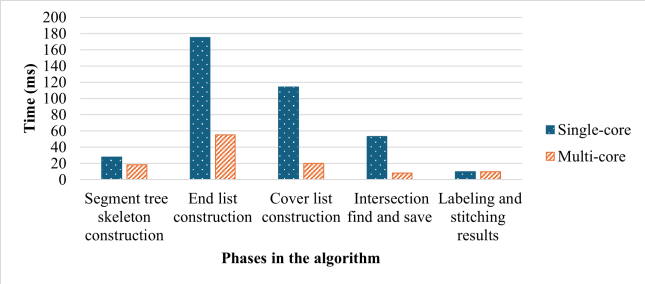

Figure 11: Execution time breakdown for Dataset 5 using 10 OpenMP threads.

We investigated the limits of OpenMP directives for parallelizing this tree-based workload as our experiments did not achieve more than 4-fold relative speedup. Our algorithm has five major phases: Segment tree construction comprising the construction of the tree skeleton, *cover list*, and *end list*, line segment intersection discovery, and result tracing. Fig. 11 displays the run time breakdown of the serial and multi-core versions over Dataset 5. The serial algorithm spends 73% - 83% of its run time in segment tree construction constituting major overhead. On average, 6% of the run time is spent on tree skeleton construction, 31% on *cover-list* construction, and 38% on *end-list* building. Another 23% of the run time is spent at intersection finding and saving. We focused our parallelization effort on the dominating tree skeleton construction, *cover-list* and *end-list* constructions, and the output sensitive intersection finding phases. On average, it is 98% of the total run time.

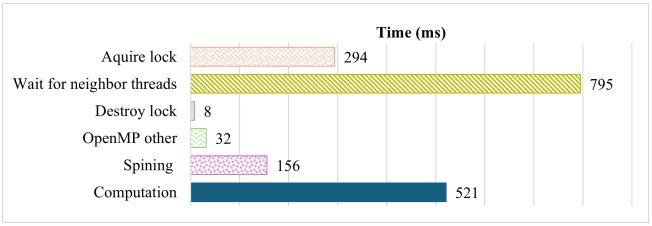

Figure 12: Multi-core execution time profile for Dataset 5 employing ten threads.

Fig 12 is a snapshot of the runtime profiling over Dataset 5 using Intel VTune profiler. The algorithm spends 16% of its run time acquiring and destroying locks used in the *cover list* construction phase. Another 44% of the run time is spent in thread synchronization indicating the imbalanced nature of the tree traversal and augmented data structure construction. This also leads the algorithm to spend 9% of its run time spinning. The algorithm only spends 29% of its run time on the computations. This shows why the relative speedup of the algorithm is limited to 3X-4X. The augmented data structures help to eliminate a massive amount of non-intersecting edge pairs, but constructing them practically over a parallel platform employing only OpenMP compiler directives involves a lot of synchronization limiting its performance. This also points to potential gain by using rigorous parallelization using pthread like low-level library.

4.7 Hybrid CPU-GPU Performance

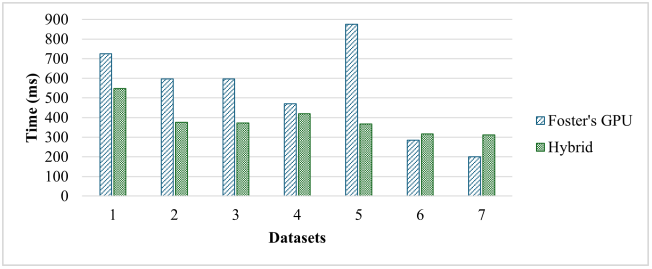

Figure 13: Performance of the hybrid CPU-GPU algorithm over against Foster's GPU polygon clipping algorithm (excluding I/O times).

To investigate the limits of OpenACC compiler directives for this tree-based algorithm, we offloaded the parallel kernels of the segment tree based polygon clipping algorithm in a CPU-GPU hybrid environment to compare against the state-of-the-art CUDA based Foster's GPU implementation. Fig 13 depicts hybrid run times vs Foster's GPU implementation. The hybrid performance is limited but on par with or better than Foster's. Implementing this method

in parallel environments is challenging since a tree is involved. In the next section, we discuss the limitations of our hybrid algorithm using a profiling study.

4.8 Hybrid CPU-GPU Algorithm Run Time Profile

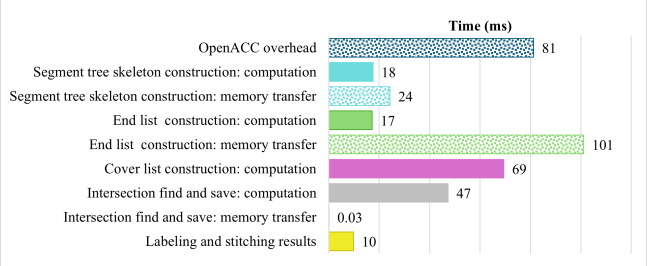

Figure 14: Hybrid execution time breakdown for Dataset 5.

Fig 14 depicts a snapshot of the runtime profiling over Dataset 5 in the hybrid environment. 22% of the total parallel time is spent initializing the OpenACC run time environment. Tree skeleton construction computation spends 5% and another 7% in memory transfers to copy input data. The *end list* construction spends 5% and 27% of the total run time in memory transfers. The *end list* building from the leaf level to the root level only using a GPU degraded the performance and we moved that computation to the CPU. This leads to an increase in the memory transfer time since the intermediate results need to be copied between the CPU and the GPU. However, this approach is 2X faster overall compared to the GPU only approach. 19% of the total run time is spent constructing the *cover list* and 13% of the run time is spent discovering the intersections. The OpenACC overhead and memory transfers at tree skeleton and *end list* constructions thus severely limit the overall performance of the hybrid algorithm.

5 CONCLUSION

In this work, we demonstrate a practical segment tree based polygon clipping. With our augmentation, segment trees can perform efficient polygon clipping in addition to the standard operations like stabbing query. We presented a multi-core segment tree-based polygon clipping algorithm employing OpenMP directives that utilizes Chaselle's observation and Foster's polygon clipping labeling. In our experiments using real-world datasets, the segment tree eliminates 99% of non-intersecting edge pairs on average compared to all-to-all edge pair tests resulting in up to 7.8 fold speedup advancing the state-of-the-art Foster's CUDA based implementation. To achieve this performance, we parallelized intersection discovery, tree skeleton, *cover-list*, and *end-list* constructions.

Evaluations using real-world datasets indicate that our algorithm performs reasonably well with a large number of input edges with OpenMP based directives and shows potential for better acceleration on multi-core and many-core platforms employing rigorous low-level libraries such as PThread and CUDA. Our approach lays a foundation for implementing an efficient practical sweep-line-based parallel polygon clipping algorithm in the future.

ACKNOWLEDGMENTS

This work is partly funded by NSF grants #2313039 and #2344585.

REFERENCES

[1] 2022. https://www.naturalearthdata.com/
[2] Danial Aghajarian, Satish Puri, and Sushil Prasad. 2016. GCMF: an efficient end-to-end spatial join system over large polygonal datasets on GPGPU platform. In *Proceedings of the 24th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*. 1–10.
[3] M K Buddhi Ashan, Satish Puri, and Sushil K Prasad. 2023. Efficient PRAM and Practical GPU Algorithms for Large Polygon Clipping with Degenerate Cases. In *2023 IEEE/ACM 23rd International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*. IEEE, 579–591.
[4] Samuel Audet, Cecilia Albertsson, Masana Murase, and Akihiro Asahara. 2013. Robust and efficient polygon overlay on parallel stream processors. In *Proceedings of the 21st ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*. 304–313.
[5] Jon Louis Bentley. 1977. Solutions to Klee's rectangle problems. *Unpublished manuscript* (1977), 282–300.
[6] CGAL Editorial Board. 2022. CGAL: Computational Geometry Algorithms Library. https://www.cgal.org/
[7] Albert Chan, Frank Dehne, and Andrew Rau-Chaplin. 1999. Coarse-grained parallel geometric search. *J. Parallel and Distrib. Comput.* 57, 2 (1999), 224–235.
[8] Bernard Chaselle. 1984. Intersecting is easier than sorting. In *Proceedings of the sixteenth annual ACM symposium on Theory of computing*. 125–134.
[9] Frank Dehne and Andrew Rau-Chaplin. 1990. Implementing data structures on a hypercube multiprocessor, and applications in parallel computational geometry. In *Graph-Theoretic Concepts in Computer Science: 15th International Workshop WG'89 Castle Rolduc, The Netherlands, June 14–16, 1989 Proceedings 15*. Springer, 316–329.
[10] Ahmed Eldawy and Mohamed F. Mokbel. 2015. SpatialHadoop: A MapReduce framework for spatial data. In *31st IEEE International Conference on Data Engineering, ICDE 2015, Seoul, South Korea, April 13-17, 2015*, Johannes Gehrke, Wolfgang Lehner, Kyuseok Shim, Sang Kyun Cha, and Guy M. Lohman (Eds.). IEEE Computer Society, 1352–1363. https://doi.org/10.1109/ICDE.2015.7113382
[11] Erich L Foster, Kai Hormann, and Romeo Traian Popa. 2019. Clipping simple polygons with degenerate intersections. *Computers & Graphics: X* 2 (2019), 100007.
[12] Chao Gao, Furqan Baig, Hoang Vo, Yangyang Zhu, and Fusheng Wang. 2018. Accelerating cross-matching operation of geospatial datasets using a CPU-GPU hybrid platform. In *2018 IEEE International Conference on Big Data (Big Data)*. IEEE, 3402–3411.
[13] Alexandros V Gerbessiotis. 2006. An architecture independent study of parallel segment trees. *Journal of Discrete Algorithms* 4, 1 (2006), 1–24.
[14] Michael T Goodrich. 1989. Intersecting line segments in parallel with an output-sensitive number of processors. In *Proceedings of the first annual ACM symposium on Parallel algorithms and architectures*. 127–137.
[15] Günther Greiner and Kai Hormann. 1998. Efficient clipping of arbitrary polygons. *ACM Transactions on Graphics (TOG)* 17, 2 (1998), 71–83.
[16] Joseph JáJá. 1992. *An introduction to parallel algorithms*. Addison Wesley Longman Publishing Co., Inc.
[17] You-Dong Liang and Brian A Barsky. 1983. An analysis and algorithm for polygon clipping. *Commun. ACM* 26, 11 (1983), 868–877.
[18] Patrick Gilles Maillot. 1992. A new, fast method for 2D polygon clipping: analysis and software implementation. *ACM Transactions on Graphics (TOG)* 11, 3 (1992), 276–290.
[19] de Berg Mark, Cheong Otfried, van Kreveld Marc, and Overmars Mark. 2008. *Computational geometry algorithms and applications*. Spinger.
[20] Francisco Martinez, Antonio Jesus Rueda, and Francisco Ramon Feito. 2009. A new algorithm for computing Boolean operations on polygons. *Computers & Geosciences* 35, 6 (2009), 1177–1185.
[21] Rogue Modron. 2011. Polygon Clipping: a Wrapper, a Benchmark. https://rogue-modron.blogspot.com/2011/04/polygon-clipping-wrapper-benchmark.html
[22] Jürg Nievergelt and Franco P Preparata. 1982. Plane-sweep algorithms for intersecting geometric figures. *Commun. ACM* 25, 10 (1982), 739–747.
[23] Anmol Paudel. 2022. *Acceleration of Computational Geometry Algorithms for High Performance Computing Based Geo-Spatial Big Data Analysis*. Ph. D. Dissertation. Marquette University.
[24] Satish Puri and Sushil K Prasad. 2014. Output-sensitive parallel algorithm for polygon clipping. In *2014 43rd International Conference on Parallel Processing*. IEEE, 241–250.
[25] Satish Puri and Sushil K Prasad. 2015. A parallel algorithm for clipping polygons with improved bounds and a distributed overlay processing system using MPI. In *2015 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*. IEEE, 576–585.

[26] B-O Schneider and Jim van Welzen. 1998. Efficient polygon clipping for an SIMD graphics pipeline. *IEEE Transactions on Visualization and Computer Graphics* 4, 3 (1998), 272–285.
[27] Guobin Shen, Changxi Zheng, Wei Pu, and Shipeng Li. 2007. *Distributed segment tree: A unified architecture to support range query and cover query.* Technical Report. Technical Report, Microsoft Research Asia.
[28] Lucanus J Simonson. 2010. Industrial strength polygon clipping: A novel algorithm with applications in VLSI CAD. *Computer-Aided Design* 42, 12 (2010), 1189–1196.
[29] Peter Su and Scot Drysdale. 1992. *Building segment trees in parallel.* Technical Report. Dartmouth College.
[30] Ivan E Sutherland and Gary W Hodgman. 1974. Reentrant polygon clipping. *Commun. ACM* 17, 1 (1974), 32–42.
[31] Bala R Vatti. 1992. A generic solution to polygon clipping. *Commun. ACM* 35, 7 (1992), 56–63.
[32] Kevin Weiler and Peter Atherton. 1977. Hidden surface removal using polygon area sorting. *ACM SIGGRAPH computer graphics* 11, 2 (1977), 214–222.
[33] Changxi Zheng, Guobin Shen, Shipeng Li, and Scott Shenker. 2006. Distributed Segment Tree: Support of Range Query and Cover Query over DHT.. In *IPTPS*.

Received 29 April 2024; revised 12 March 2009; accepted 10 June 2009