OpenACC Based GPU Parallelization of Plane Sweep Algorithm for Geometric Intersection

Anmol Paudel(✉) and Satish Puri

Department of Mathematics, Statistics and Computer Science,
Marquette University, Milwaukee, WI 53233, USA
{anmol.paudel,satish.puri}@marquette.edu

Abstract. Line segment intersection is one of the elementary operations in computational geometry. Complex problems in Geographic Information Systems (GIS) like finding map overlays or spatial joins using polygonal data require solving segment intersections. Plane sweep paradigm is used for finding geometric intersection in an efficient manner. However, it is difficult to parallelize due to its in-order processing of spatial events. We present a new fine-grained parallel algorithm for geometric intersection and its CPU and GPU implementation using OpenMP and OpenACC. To the best of our knowledge, this is the first work demonstrating an effective parallelization of plane sweep on GPUs.

We chose compiler directive based approach for implementation because of its simplicity to parallelize sequential code. Using Nvidia Tesla P100 GPU, our implementation achieves around 40X speedup for line segment intersection problem on 40K and 80K data sets compared to sequential CGAL library.

Keywords: Plane sweep · Line segment intersection · Directive based programming · OpenMP · OpenACC

1 Introduction

Scalable spatial computation on high performance computing (HPC) environment has been a long-standing challenge in computational geometry. Spatial analysis using two shapefiles (4 GB) takes around ten minutes to complete using state-of-the art desktop ArcGIS software [15]. Harnessing the massive parallelism of graphics accelerators helps to satisfy the time-critical nature of applications involving spatial computation. Directives-based parallelization provides an easy-to-use mechanism to develop parallel code that can potentially reduce execution time. Many computational geometry algorithms exhibit irregular computation and memory access patterns. As such, parallel algorithms need to be carefully designed to effectively run on a GPU architecture.

Geometric intersection is a class of problems involving operations on shapes represented as line segments, rectangles (MBR), and polygons. The operations

S. Chandrasekaran et al. (Eds.): WACCPD 2018 Workshop, LNCS 11381, pp. 114–135, 2019.
https://doi.org/10.1007/978-3-030-12274-4_6

can be cross, overlap, contains, union, etc. Domains like Geographic Information Systems (GIS), VLSI CAD/CAM, spatial databases, etc. use geometric intersection as an elementary operation in their data analysis toolbox. Public and private sector agencies rely on spatial data analysis and spatial data mining to gain insights and produce an actionable plan [14]. We are experimenting with the line segment intersection problem because it is one of the most basic problems in spatial computing and all other operations for bigger problems like polygon overlay or polygon clipping depends on results from it. The line segment intersection problem basically asks two questions - "are the line segments intersecting or not?" and if they are intersecting "what are the points of intersection?" The first one is called intersection detection problem and the second one is called intersection reporting problem. In this paper, we present an algorithmic solution for the latter.

Plane sweep is a fundamental technique to reduce $O(n^2)$ segment to segment pair-wise computation into $O(nlogn)$ work, impacting a class of geometric problems akin to the effectiveness of FFT-based algorithms. Effective parallelization of the plane-sweep algorithm will lead to a breakthrough by enabling acceleration of computational geometry algorithms that rely on plane-sweep for efficient implementation. Examples include trapezoidal decomposition, construction of the Voronoi diagram, Delaunay triangulation, etc.

To the best of our knowledge, this is the first work on parallelizing plane sweep algorithm for geometric intersection problem on a GPU. The efficiency of plane sweep comes from its ability to restrict the search space to the immediate neighborhood of the sweepline. We have abstracted the neighbor finding algorithm using directive based reduction operations. In sequential implementations, neighbor finding algorithm is implemented using a self-balancing binary search tree which is not suitable for GPU architecture. Our multi-core and many-core implementation uses directives-based programming approach to leverage the device-specific hardware parallelism with the help of a compiler. As such the resulting code is easy to maintain and modify. With appropriate pragmas defined by OpenMP and OpenACC, the same source code will work for a CPU as well as a GPU.

In short, the paper presents the following research contributions

1. Fine-grained Parallel Algorithm for Plane Sweep based intersection problem.
2. Directives-based implementation with reduction-based approach to find neighbors in the sweeplines.
3. Performance results using OpenACC and OpenMP and comparison with sequential CGAL library. We report upto 27x speedup with OpenMP and 49x speedup with OpenACC for 80K line segments.

The rest of the paper is structured as follows. Section 2 presents a general technical background and related works to this paper. Section 3 describes our parallel algorithm. Section 4 provides details on OpenMP and OpenACC implementations. Section 5 provides experimental results. Conclusion and future work is presented in Sect. 6. Acknowledgements are in the last section.

2 Background and Related Work

There are different approaches for finding geometric intersections. In addition to the simple brute force method, there is a filter and refine method that uses a heuristic to avoid unnecessary intersection computations. For a larger dataset, filter and refine strategy is preferred over brute force. Plane sweep method works best if the dataset can fit in memory. However, the plane sweep algorithm is not amenable to parallelization due to the in-order sequential processing of events stored in a binary tree and a priority queue data structure. In the existing literature, the focus of parallel algorithms in theoretical computational geometry has been in improving the asymptotic time bounds. However, on the practical side, there has been only a few attempts to parallelize plane sweep on multi-cores. Moreover, those algorithms are not suitable to fine-grained SIMD parallelism in GPUs. This has led to the parallelization of brute force algorithms with $O(n^2)$ complexity and parallelization of techniques like grid partitioning on GPUs. The brute force algorithm that involves processing all segments against each other is obviously embarrassingly parallel and has been implemented on GPU, but its quadratic time complexity cannot compete even with the sequential plane sweep for large data sets. The uniform grid technique does not perform well for skewed data sets where segments span an arbitrary number of grid cells. Limitations in the existing work is our motivation behind this work.

In the remaining subsections, we have provided background information about segment intersection problem, different strategies used to solve the problem, existing work on the parallelization in this area and directive based programming.

2.1 Segment Intersection Problem

Finding line intersection in computers is not as simple as solving two mathematical equations. First of all, it has to do with how the lines are stored in the computer – not in the $y = mx + c$ format, but rather as two endpoints like (x1, y1, x2, y2). One reason for not storing lines in a equation format is because most of the lines in computer applications are finite in nature, and need to have a clear start and end points. Complex geometries like triangle, quadrilateral or any n-vertices polygon are further stored as a bunch of points. For example a quadrilateral would be stored like (x1, y1, x2, y2, x3, y3, x4, y4) and each sequential pair of points would form the vertices of that polygon. So, whenever we do geometric operations using computers, we need to be aware of the datatypes used to store the geometries, and use algorithms that can leverage them.

For non-finite lines, any two lines that are not parallel or collinear in 2D space would eventually intersect. This is however not the case here since all the lines we have are finite. So given two line segments we would first need to do a series of calculation to ascertain whether they intersect or not. Since they are finite lines, we can solve their mathematical equations to find the point of intersection only if they intersect.

In this way we can solve the segment intersection for two lines but what if we are given a collection of line segments and are asked to find out which of these segments intersect among themselves and what are the intersection vertices. Since most complex geometries are stored as a collection of vertices which results in a collection of line segments, segment intersection detection and reporting the list of vertices of intersection are some of the most commonly solved problems in many geometric operations. Geometric operations like finding the map overlays and geometric unions all rely at their core on the results from the segment intersection problem. Faster and more efficient approaches in segment intersection will enable us to solve a wide variety of geometric operations faster and in a more efficient manner.

2.2 Naive Brute Force Approach

Like with any computational problem, the easiest approach is foremost the brute force approach. Algorithm 1 describes the brute force approach to find segment intersection among multiple lines.

Algorithm 1. Naive Brute Force

```
1: Load all lines to L
2: for each line l1 in L do
3:     for each line l2 in L do
4:         Test for intersection between l1 and l2
5:         if intersections exists then
6:             calculate intersection point
7:             store it in results
8:         end if
9:     end for
10: end for
```

The brute force approach works very well compared to other algorithms for the worst case scenario where all segments intersect among themselves. For N line segments, its time complexity is $O(N^2)$. This is the reason we have parallelized this algorithm here. However, if the intersections are sparse, then there are heuristics and sophisticated algorithms available. The first method is to use filter and refine heuristic which we have employed for joining two polygon layers where the line segments are taken from polygons in a layer. The second method is to apply Plane Sweep algorithm.

Filter and Refine Approach: Let us consider a geospatial operation where we have to overlay a dataset consisting of N county boundaries (polygons) on top of another dataset consisting of M lakes from USA in a Geographic Information System (GIS) to produce a third dataset consisting of all the polygons from both datasets. This operation requires O(NM) pairs of polygon intersections in the worst case. However, not all county boundaries overlap with all lake boundaries.

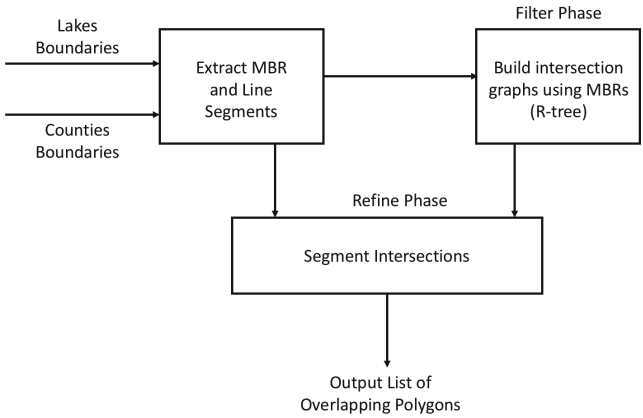

Fig. 1. Polygon intersection using filter and refine approach

This observation lends itself to filter and refine strategy where using spatial data structure like Rectangle tree (R-tree) built using bounding box approximation (MBR) of the actual boundaries, we prune the number of cross layer polygon intersections [1]. We have employed this approach while handling real spatial data. Figure 1 shows the workflow for joining two real-world datasets. The output consists of counties with lakes. The compute-intensive part here is the refine phase. Our directive based parallelization is used in the refine phase only.

2.3 Plane Sweep Algorithm

Plane sweep is an efficient algorithmic approach used in finding geometric intersections. Its time complexity is $O((N + K) \log N)$ where N is the number of line segments and K is the number of intersections found. In the worst case, K is $O(N^2)$, which makes it an $O(N^2 \log N)$ algorithm. Parallelization of plane sweep algorithm will impact many computational geometry algorithms that rely on plane-sweep for efficient implementation e.g. spatial join, polygon overlay, voronoi diagram, etc. The Bentley-Ottmann algorithm is a plane sweep algorithm, that given a collection of lines, can find out whether there are intersecting lines or not [5]. Computational geometry libraries typically use plane sweep algorithm in their implementations.

Algorithm 2 describes plane sweep using a vertical sweepline. The procedures for *HandleStartEvent*, *HandleEndEvent* and *HandleIntersectionEvent* used in Algorithm 2 are given in Algorithms 4, 5, 6 respectively. For simplicity in presentation, following assumptions are made in Algorithm 2:

1. No segment is parallel to the vertical sweeplines.
2. No intersection occurs at endpoints.
3. No more than two segments intersect in the same point.
4. No overlapping segments.

Algorithm 2. Plane Sweep

1: Load all lines to L
2: Initialize a priority queue (PQ) for sweeplines which retrieves items based on the y-position of the item
3: Insert all start and end points from L to PQ
4: Initialize a sweepline
5: While PQ is not empty:
    If the nextItem is startevent:
        The segment is added to the sweepline
        HandleStartEvent(AddedSegment)
    If the nextItem is endevent:
        The segment is removed from the sweepline
        HandleEndEvent(RemovedSegment)
    If the nextItem is intersection-event:
    [Note that there will be two contributing lines at intersection point. Let these two lines be l_1 *and* l_2*.]*
        HandleIntersectionEvent(l_1,l_2)
        Record the intersecting pairs

The segments that do not adhere to our assumptions in our dataset are called degenerate cases.

2.4 Existing Work on Parallelizing Segment Intersection Algorithms

Methods for finding intersections can be categorized into two classes: (i) algorithms which rely on a partitioning of the underlying space, and (ii) algorithms exploiting a spatial order defined on the segments. Plane sweep algorithm and theoretical algorithms developed around 80's and 90's fall under the second category [3,7,8]. These theoretical PRAM algorithms attain near-optimal poly-logarithmic time complexity [3,7,17]. These algorithms focus on improving the asymptotic time bounds and are not practical for implementation purposes. These parallel algorithms are harder to implement because of their usage of complex tree-based data structures like parallel segment tree and hierarchical plane-sweep tree (array of trees) [4]. Moreover, tree-based algorithms may not be suitable for memory coalescing and vectorization on a GPU.

Multi-core and many-core implementation work in literature fall under the first category where the input space is partitioned for spatial data locality. The basic idea is to process different cells in parallel among threads. Based on the data distribution, existing parallel implementations of geometric intersection algorithm use uniform or adaptive grid to do domain decomposition of the input space and data [2,4,6]. Ideal grid dimension for optimal run-time is hard to determine as it depends not only on spatial data distribution, but also on hardware characteristics of the target device. Moreover, the approach of dividing the underlying space has the unfortunate consequence of effectively increasing the size of the input dataset. For instance, if an input line segment spans multiple grid cells, then the segment is simply replicated in each cell. Hence, the problem

size increases considerably for finer grid resolutions. In addition to redundant computations for replicated data, in GPU with limited global memory, memory allocation for intermediate data structure to store replicated data is not space-efficient. Plane sweep does not suffer from this problem because it is an event-based algorithm.

The brute force algorithm that involves processing all line segments against each other is obviously embarrassingly parallel and has been implemented on GPU [11], but its quadratic time complexity cannot compete even with the sequential plane sweep for large data sets. Our current work is motivated by the limitations of the existing approaches which cannot guarantee efficient treatment of all possible input configurations.

Parallel algorithm developed by McKenney et al. and their OpenMP implementation is targeted towards multi-core CPUs and it is not fine-grained to exploit the SIMT parallelism in GPUs [9,10,12]. Contrary to the above-mentioned parallel algorithm, our algorithm is targeted to GPU and achieves higher speedup. In the context of massively parallel GPU platform, we have sacrificed algorithmic optimality by not using logarithmic data structures like priority queue, self-balancing binary tree and segment tree. Our approach is geared towards exploiting the concurrency available in the sequential plane sweep algorithm by adding a preprocessing step that removes the dependency among successive events.

2.5 OpenMP and OpenACC

When using compiler directives, we need to take care of data dependencies and race conditions among threads. OpenMP provides critical sections to avoid race conditions. Programmers need to remove any inter-thread dependencies from the program.

Parallelizing code for GPUs has significant differences because GPUs are separate physical devices with their numerous cores and their own separate physical memory. So, we need to first copy the spatial data from CPU to GPU to do any data processing on a GPU. Here, the CPU is regarded as the host and the GPU is regarded as the device. After processing on GPU is finished, we need to again copy back all the results from the GPU to the CPU. In GPU processing, this transfer of memory has overheads and these overheads can be large if we do multiple transfers or if the amount of memory moved is large. Also, each single GPU has its own physical memory limitations and if we have a very large dataset, then we might have to copy it to multiple GPUs or do data processing in chunks. Furthermore, the functions written for the host may not work in the GPUs and will require writing new routines. Any library modules loaded on the host device is also not available on a GPU device.

The way we achieve parallelization with OpenACC is by doing loop parallelization. In this approach each iteration of the loop can run in parallel. This can only be done if the loops have no inter-loop dependencies. Another approach we use is called vectorization. In the implementation process, we have to remove any inter-loop dependencies so that the loops can run in parallel without

any side-effects. Side-effects are encountered if the threads try to write-write or write-read at the same memory location resulting in race conditions.

3 Parallel Plane Sweep Algorithm

We have taken the vertical sweep version of the Bentley-Ottmann algorithm and modified it. Instead of handling event points strictly in the increasing y-order as they are encountered in bottom-up vertical sweep, we process all the startpoints first, then all the endpoints and at last we keep on processing until there aren't any unprocessed intersection points left. During processing of each intersection event, multiple new intersection events can be found. So, the last phase of processing intersection events is iterative. Hence, the sequence of event processing is different than sequential algorithm.

Algorithm 3 describes our modified version of plane sweep using a vertical sweepline. Figure 2 shows the startevent processing for a vertical bottom up sweep. Algorithm 3 also has the same simplifying assumptions like Algorithm 2. Step 2, step 3 and the for-loop in step 4 of Algorithm 3 can be parallelized using directives.

Algorithm 3. Modified Plane Sweep Algorithm

1: Load all lines to L
2: For each line l_1 in L:
    Create a start-sweepline (SSL) at the lower point of l_1
    For each line l_2 in L:
        If l_2 crosses SSL:
            update left and right neighbors
    HandleStartEvent(l_1)
3: For each line l_1 in L:
    Create an end-sweepline (ESL) at the upper point of l_1
    For each line l_2 in L:
        If l_2 crosses ESL:
            update left and right neighbors
    HandleEndEvent(l_1)
4: While intersection events is not empty, for each intersection event:
    Create an intersection-sweepline (ISL) at the intersection point
    For each line l in L:
        If l crosses ISL:
            update left and right neighbors
     // let l_1 and l_2 are the lines at intersection event
        HandleIntersectionEvent(l_1, l_2)
5: During intersection events, we record the intersecting pairs

Algorithm 3 describes a fine-grained approach where each event point can be independently processed. Existing work for plane sweep focuses on coarse-grained parallelization on multi-core CPUs only. Sequential Bentley-Ottmann algorithm

Algorithm 4. StartEvent Processing

1: **procedure** HANDLESTARTEVENT(l_1)
    Intersection is checked between
        l_1 and its left neighbor
        l_1 and its right neighbor
    If any intersection is found
        update intersection events
2: **end procedure**

Algorithm 5. EndEvent Processing

1: **procedure** HANDLEENDEVENT(l_1)
    Intersection is checked between
        the left and right neighbors of l_1
    If intersection is found
        update intersection events
2: **end procedure**

Algorithm 6. IntersectionEvent Processing

1: **procedure** HANDLEINTERSECTIONEVENT(l_1,l_2)
    Intersection is checked between
        the left neighbor of the intersection point and l_1
        the right neighbor of the intersection point and l_1
        the left neighbor of the intersection point and l_2
        the right neighbor of the intersection point and l_2
    if any intersection is found
        update intersection events
2: **end procedure**

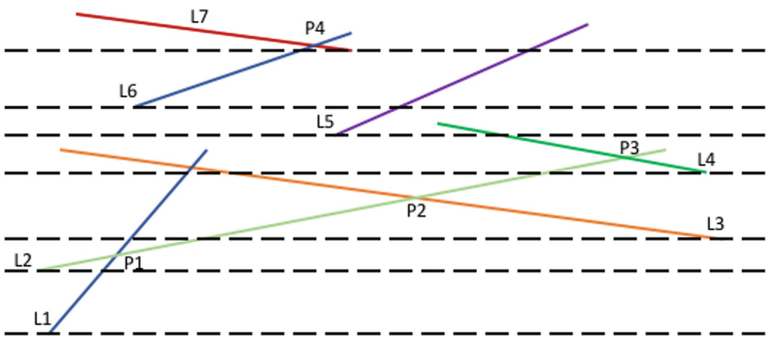

Fig. 2. Vertical plane sweep. Vertical plane sweep showing sweeplines (dotted lines) corresponding to starting event points only. P1 to P4 are the intersection vertices found by processing start event points only. L1, L2 and L3 are the active line segments on the third sweepline from the bottom. Event processing of starting point of L3 requires finding its immediate neighbor (L2) and checking doesIntersect (L2, L3) which results in finding P2 as an intersection vertex.

processes the event points as they are encountered while doing a vertical/horizontal sweep. Our parallel plane sweep relaxes the strict increasing order of event processing. Start and End point events can be processed in any order. As shown in step 4 of Algorithm 3, intersection event point processing happens after start and end point events are processed. An implementation of this algorithm either needs more memory to store line segments intersecting the sweepline or needs to compute them dynamically thereby performing more work. However, this is a necessary overhead required to eliminate the sequential dependency inherent in the original Bentley-Ottmann algorithm or its implementation. As we point out in the results section, our OpenMP and OpenACC implementations perform better than the existing work.

Degree of Concurrency: The amount of concurrency available to the algorithm is limited by Step 4 due to the fact that intersection events produce more intersection events dynamically. Hence, it results in a dependency graph where computation on each level generates a new level. The critical path length of the graph denoted by l is $0 < l < \binom{n}{2}$ where n is the input size. In general, l is less than the number of intersection points k. However, if l is comparable to k, then the Step 4 may not benefit from parallelization.

3.1 Algorithm Correctness

The novelty in this parallel algorithm is our observation that any order of concurrent events processing will produce the same results as done sequentially, provided that we schedule intersection event handling in the last phase. In a parallel implementation, this can be achieved at the expense of extra memory requirement to store the line segments per sweepline or extra computations to dynamically find out those line segments. This observation lends itself to directive based parallel programming because now we can add parallel for loop pragma in Steps 2, 3 and 4 so that we can leverage multi-core CPUs and many-core GPUs. The proof that any sweepline event needs to only consider its immediate neighbors for intersection detection is guaranteed to hold as shown by the original algorithm.

Bentley-Ottmann algorithm executes sequentially, processing each sweepline in an increasing priority order with an invariant that all the intersection points below the current sweepline has been found. However, since we process each sweepline in parallel, this will no longer be the case. The invariant in our parallel algorithm is that all line segments crossing a sweepline needs to be known a priori before doing neighborhood computation. As we can see, this is an embarrassingly parallel step.

Finally, we can show that Algorithm 3 terminates after finding all intersections. Whenever start-events are encountered they can add atmost two intersection events. End-events can add atmost one intersection event and intersection events can add atmost 4 intersection events. Because of the order in which the algorithm processes the sweeplines, all the intersection points below the current sweepline will have been found and processed. The number of iterations for Step 2 and Step 3 can be statically determined and it is linear in the number of inputs.

However, the number of iterations in Step 4 is dynamic and can be quadratic. Intersection events produce new intersection events. However, even in the worst case with $\binom{n}{2}$ intersection points generated in Step 4, the algorithm is bound to terminate.

3.2 Algorithmic Analysis

Time Complexity. For each of the N lines there will be two sweeplines, and each sweepline will have to iterate over all N lines to check if they intersect or not. So this results in $2N^2$ comparison steps, and then each intersection event will also produce a sweepline and if there are K intersection points this results in K * N steps so the total is $2N^2 + K * N$ steps. Assuming that $K \ll N$, the time-complexity of this algorithm is $O(N^2)$.

Space Complexity. Since there will be $2N$ sweeplines for N lines and for each K intersection events there will be K sweeplines. The extra memory requirement will be $O(N + K)$ and assuming $K \ll N$, the space-complexity of the algorithm is $O(N)$.

4 Directive-Based Implementation Details

Although steps 2, 3 and 4 of Algorithm 3 could run concurrently, we implemented it in such a way that each of the sweeplines within each step is processed in parallel. Also, in step 4 the intersection events are handled in batch for the ease of implementation. Furthermore, we had to make changes to the sequential code so that it could be parallelized with directives. In the sequential algorithm, the segments overlapping with a sweepline are usually stored in a data structure like BST. However, when each of the sweeplines are needed to be processed in parallel, using a data structure like the BST is not feasible so we need to apply different techniques to achieve this. In OpenMP, we can find neighbors by sorting lines in each sweepline and processing them on individual threads. Implementing the same sorting based approach is again not feasible in OpenACC because we cannot use the sorting libraries that are supported in OpenMP. So, we used a reduction-based approach supported by the reduction operators provided by OpenACC to achieve this without having to sort the lines in each sweepline.

Listing 1.1. Data structure for point

```
struct Point {
        var x,y;
        Point(var x, var y);
}
```

Listing 1.2. Data structure for line

```
struct Line {
        Point p1, p2;
        var m, c;

        Line(Point p1, Point p2) {
                m = ((p2.y - p1.y) / (p2.x - p1.x));
                c = (p1.y) - m*(p1.x);
        }
};
```

Listing 1.3. Routine for intersection point

```
#pragma acc routine
Point intersectionPoint(Line l1, Line l2) {
        var x = (l2.c - l1.c)/(l1.m - l2.m);
        var y = l1.m*x + l1.c;
        return Point(x,y);
}
```

Listing 1 shows the spatial data structures used in our implementations. The keyword *var* in the listing is meant to be a placeholder for any numeric datatype.

Finding neighboring line segments corresponding to each event efficiently is a key step in parallelizing plane sweep algorithm. In general, each sweepline has a small subset of the input line segments crossing it in an arbitrary order. The size of this subset varies across sweeplines. Finding neighbors per event would amount to sorting these subsets that are already present in global memory individually, which is not as efficient as global sorting of the overall input. Hence, we have devised an algorithm to solve this problem using directive based reduction operation. A reduction is necessary to avoid race conditions.

Algorithm 7 explains how neighbors are found using OpenACC. Each horizontal sweepline has a x-location around which the neighbors are to be found. If it is a sweepline corresponding to a startpoint or endpoint then the x-coordinate of that point will be the x-location. For a sweepline corresponding to an intersection point, the x-coordinate of the intersection point will be the x-location. To find the horizontal neighbors for the x-location, we need the x-coordinate of the intersection point between each of the input lines and the horizontal sweepline. Then a *maxloc* reduction is performed on all such intersection points that are to the left of the x-location and a *minloc* reduction is performed on all such intersection points that are to the right of the x-location to find the indices of previous and next neighbors respectively. A *maxloc* reduction finds the index of the maximum value and a *minloc* reduction finds the index of the minimum value. OpenACC doesn't directly support the *maxloc* and *minloc* operators so a workaround was implemented. The workaround includes casting the data and index combined to a larger numeric data structure for which max and min reductions are available and extracting the index from reduction results.

Figure 3 shows an example for finding two neighbors for an event with x-location as 25. The numbers shown in boxes are the x-coordinates of the intersection points of individual line segments with a sweepline (SL). We first find the index of the neighbors and then use the index to find the actual neighbors.

Algorithm 7. Reduction-based Neighbor Finding

```
1: Let SL be the sweepline
2: Let x be the x-coordinate in SL around which neighbors are needed
3: L ← all lines
4: prev ← MIN , nxt ← MAX
5: for each line l in L do-parallel reduction(maxloc:prev, minloc:nxt)
6:     if intersects(l,SL) then
7:         h ← intersectionPt(l,SL)
8:         if h < x then
9:             prev = h
10:        end if
11:        if h > x then
12:            nxt = h
13:        end if
14:    end if
15: end for
```

Polygon Intersection Using Filter and Refine Approach: As discussed earlier, joining two polygon layers to produce third layer as output requires a filter phase where we find pairs of overlapping polygons from the two input layers. The filter phase is data-intensive in nature and it is carried out in CPU. The next refine phase carries out pair-wise polygon intersection. Typically, on a dataset of a few gigabytes, there can be thousands to millions of such polygon pairs where a polygon intersection routine can be invoked to process an individual pair. First, we create a spatial index (R-tree) using minimum bounding rectangles (MBRs) of polygons of one layer and then perform R-tree queries using MBRs of another layer to find overlapping cross-layer polygons. We first tried a fine-grained parallelization scheme with a pair of overlapping polygons as an OpenMP task. But this approach did not perform well due to significantly large number of tasks. A coarse-grained approach where a task is a pair consisting of a polygon from one layer and a list of overlapping polygons from another layer performed better. These tasks are independent and processed in parallel by OpenMP due to typically large number of tasks to keep the multi-cores busy.

We used sequential Geometry Opensource (GEOS) library for R-tree construction, MBR querying and polygon intersection functions. Here, intersection function uses sequential plane-sweep algorithm to find segment intersections. We tried naive all-to-all segment intersection algorithm with OpenMP but it is slower than plane sweep based implementation. Our OpenMP implementation is based on thread-safe C API provided by GEOS. We have used the Prepared-

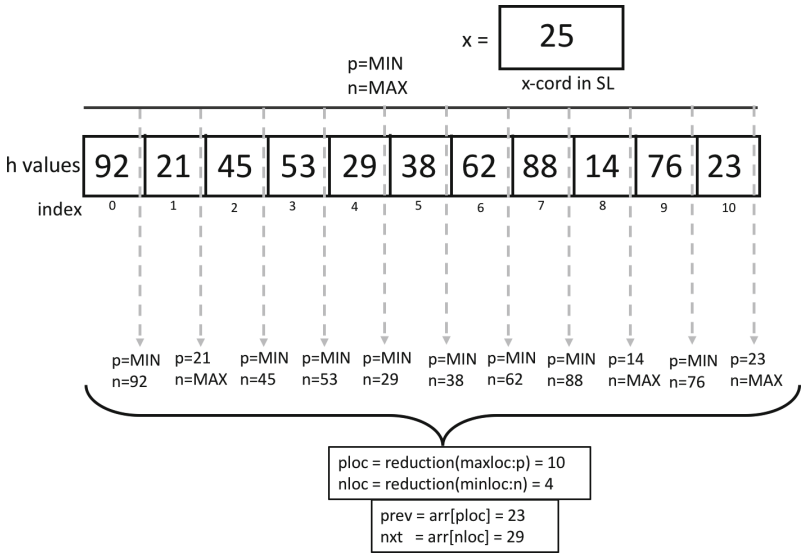

Fig. 3. Reduction-based neighbor finding. Here the dotted lines are the parallel threads and we find the left and right neighbor to the given x-cord (25) on the sweepline and their corresponding indices. p and n are thread local variables that are initialized as MIN and MAX respectively. As the threads execute concurrently their value gets independently updated based on Algorithm 7.

Geometry class which is an optimized version of Geometry class designed for filter-and-refine use cases.

Hybrid CPU-GPU Parallelization: Only the refine phase is suitable for GPU parallelization because it involves millions of segment intersections tests for large datasets. Creating intersection graph to identify overlapping polygons is carried out on CPU. The intersection graph is copied to the GPU using OpenACC data directives. The segment intersection algorithm used in OpenACC is the brute force algorithm. We cannot simply add pragmas to GEOS code. This is due to the fact that OpenACC is not designed to run sophisticated plane sweep algorithm efficiently. For efficiency, the code needs to be vectorized by the PGI compiler and allow Single Instruction Multiple Thread (SIMT) parallelization. Directive-based loop parallelism using *OpenACC parallel for* construct is used. The segment intersection computation for the tasks generated by filter phase are carried out in three nested loops. Outermost loop iterates over all the tasks. Two inner for loops carry out naive all-to-all edge intersection tests for a polygon pair.

5 Experimental Results

5.1 Experimental Setup

Our code was run on the following 3 machines:

- Everest cluster at Marquette university: This machine was used to run the OpenMP codes and contained the Intel Xeon E5 CPU v4 E5-2695 with 18 cores and 45 MB cache and base frequency of 2.10 GHz.
- Bridges cluster at the Pittsburgh Supercomputing Center: A single GPU node of this cluster was used which contained the NVIDIA Tesla P100 containing 3584 cuda cores and GPU memory of 12 GB.
- Our sequential GEOS and OpenMP code was run on 2.6 GHz Intel Xeon E5-2660v3 processor with 20 cores in the NCSA ROGER Supercomputer. We carried out the GPU experiments using OpenACC on Nvidia Tesla P100 GPU which has 16 GB of main memory and 3, 584 CUDA cores operating at 1480 MHz frequency. This GPU provides 5.3 TFLOPS of double precision floating point calculations. Version 3.4.2 of GEOS library was used[1].

Dataset Descriptions: We have used artificially generated and real spatial datasets for performance evaluation.

Generated Dataset: Random lines were generated for performance measurement and collecting timing information. Datasets vary in the number of lines generated. Sparsity of data was controlled during data set generation to have about only 10% of intersections. Table 1 shows the datasets we generated and used and the number of intersections in each dataset. The datasets are sparsely distributed and number of intersections are only about 10% of the number of lines in the dataset. Figure 4 depicts a randomly generated set of sparse lines.

Real-World Spatial Datasets: As real-world spatial data, we selected polygonal data from Geographic Information System (GIS) domain[2,3] [13]. The details of the datasets are provided in Table 2.

Table 1. Dataset and corresponding number of intersections

Lines	Intersections
10k	1095
20k	2068
40k	4078
80k	8062

[1] https://trac.osgeo.org/geos/.
[2] http://www.naturalearthdata.com.
[3] http://resources.arcgis.com.

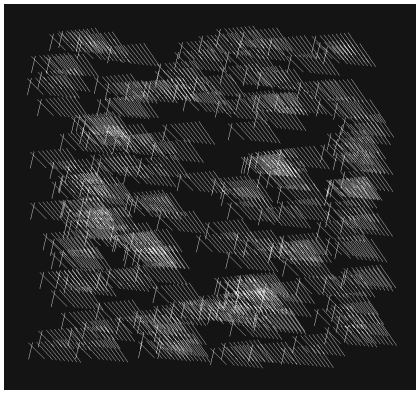

Fig. 4. Randomly generated sparse lines

Table 2. Description of real-world datasets.

	Dataset	Polygons	Edges	Size
1	Urban areas	11K	1,153K	20 MB
2	State provinces	4K	1,332K	50 MB
3	Sports areas	1,783K	20,692K	590 MB
4	Postal code areas	170K	65,269K	1.4 GB
5	Water bodies	463K	24,201K	520 MB
6	Block boundaries	219K	60,046K	1.3 GB

5.2 Performance of Brute Force Parallel Algorithm

Using Generated Dataset: Table 3 shows execution time comparison of CGAL, sequential brute-force (BF-Seq) and OpenACC augmented brute-force (BF-ACC) implementations.

Key takeaway from the Table 3 is that CGAL performs significantly better than our naive code for sparse set of lines in sequential and the increase in sequential time is not linear with the increase in data size. OpenACC however drastically beats the sequential performance especially for larger data sizes.

Table 3. Execution time by CGAL, naive Sequential vs OpenACC on sparse lines

Lines	CGAL	BF-Seq	BF-ACC
10k	3.96 s	8.19 s	0.6 s
20k	9.64 s	35.52 s	1.52 s
40k	17.23 s	143.94 s	5.02 s
80k	36.45 s	204.94 s	6.73 s

Using Real Polygonal Dataset: Here the line segments are taken from the polygons. The polygon intersection tests are distributed among CPU threads in static, dynamic and guided load-balancing modes supported by OpenMP. Table 4 shows the execution time for polygon intersection operation using three real-world shapefiles listed in Table 2. The performance of GEOS-OpenMP depends on number of threads, chunk size and thread scheduling. We varied these parameters to get the best performance for comparison with GEOS. For the largest data set, chunk size as 100 and dynamic loop scheduling yielded the best speedup for 20 threads. We see better performance using real datasets as well when compared to optimized opensource GIS library.

For polygonal data, OpenACC version is about two to five times faster than OpenMP version even though it is running brute force algorithm for the refine phase. The timing includes data transfer time. When compared to the sequential library, it is four to eight times faster.

Table 4. Performance comparison of polygon intersection operation using sequential and parallel methods on real-world datasets.

Dataset	Running time (s)		
	Sequential	Parallel	
	GEOS	OpenMP	OpenACC
Urban-States	5.77	2.63	1.21
USA-Blocks-Water	148.04	83.10	34.69
Sports-Postal-Areas	267.34	173.51	31.82

5.3 Performance of Parallel Plane Sweep Algorithm

Table 5 shows the scalability of parallel plane sweep algorithm using OpenMP on Intel Xeon E5. Table 6 is comparison of CGAL and parallel plane sweep (PS-ACC). Key takeaway from the Table 6 is that for the given size of datasets the parallel plane sweep in OpenACC drastically beats the sequential performance of CGAL or the other sequential method as shown in Table 3.

Table 5. Parallel plane sweep on sparse lines with OpenMP

Lines	1p	2p	4p	8p	16p	32p
10k	1.9 s	1.22 s	0.65 s	0.37 s	0.21 s	0.13 s
20k	5.76 s	3.24 s	1.78 s	1.08 s	0.66 s	0.37 s
40k	20.98 s	11.01 s	5.77 s	3.3 s	2.03 s	1.14 s
80k	82.96 s	42.3 s	21.44 s	12.18 s	6.91 s	3.78 s

Table 6. CGAL vs OpenACC parallel plane sweep on sparse lines

Lines	CGAL	PS-ACC
10k	3.96 s	0.33 s
20k	9.64 s	0.34 s
40k	17.23 s	0.41 s
80k	36.45 s	0.74 s

Table 7. Speedup with OpenACC when compared to CGAL for different datasets

	10K	20K	40K	80K
BF-ACC	6.6	6.34	3.43	5.42
PS-ACC	12	28.35	42.02	49.26

5.4 Speedup and Efficiency Comparisons

Table 7 shows the speedup gained when comparing CGAL with the OpenACC implementation of the brute force (BF-ACC) and plane sweep approaches (PS-ACC) on NVIDIA Tesla P100. Figure 5 shows the time taken for computing intersection on sparse lines in comparison to OpenACC based implementations with CGAL and sequential brute force. The results with directives are promising because even the brute force approach gives around a 5x speedup for 80K lines. Moreover, our parallel implementation of plane sweep gives a 49x speedup.

Figure 6 shows the speedup with varying number of threads and it validates the parallelization of the parallel plane sweep approach. The speedup is consistent with the increase in the number of threads. Figure 7 shows the efficiency (speedup/threads) for the previous speedup graph. As we can see in the figure,

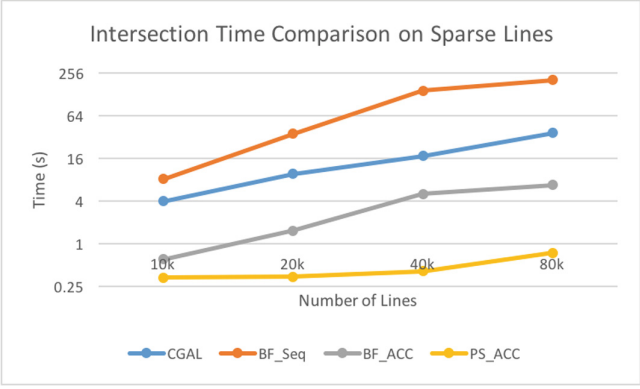

Fig. 5. Time comparison for CGAL, sequential brute-force, OpenACC augmented brute-force and plane sweep on sparse lines

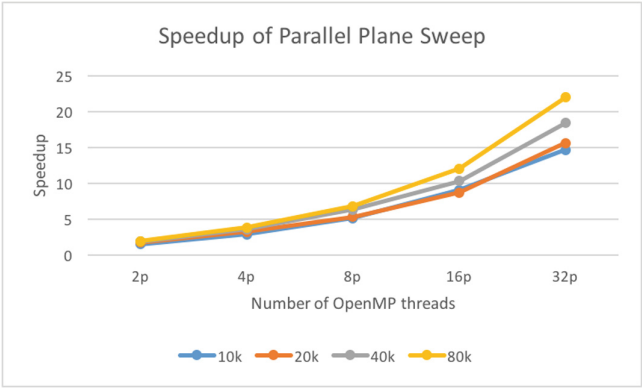

Fig. 6. Speedups for the parallel plane sweep with varying OpenMP threads on sparse lines

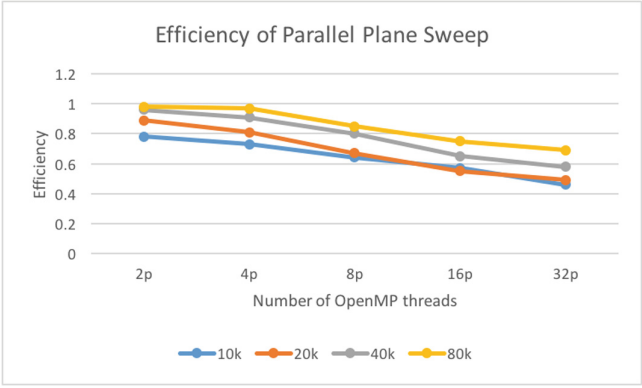

Fig. 7. Efficiency of the parallel plane sweep with varying OpenMP threads on sparse lines

the efficiency is higher for larger datasets. There is diminishing return as the number of threads increase due to the decrease in the amount of work available per thread.

Also, doing a phase-wise comparison of the OpenACC plane sweep code showed that most of the time was consumed in the start event processing (around 90% for datasets smaller than 80K and about 70% for the 80K dataset). Most of the remaining time was consumed by end event processing with negligible time spent on intersection events. The variation in time is due to the fact that the number of intersections found by different events is not the same. Moreover, start event processing has to do twice the amount of work in comparison to end event processing as mentioned in Algorithms 4 and 5. There are fewer intersection point events in comparison to the endpoint events.

6 Conclusion and Future Work

In this work, we presented a fine-grained parallel algorithm targeted to GPU architecture for a non-trivial computational geometry code. We also presented an efficient implementation using OpenACC directives that leverages GPU parallelism. This has resulted in an order of magnitude speedup compared to the sequential implementations. We have also shown our compiler directives based parallelization method using real polygonal data. We are planning to integrate the present work with our MPI-GIS software so that we can handle larger datasets and utilize multiple GPUs [16].

Compiler directives prove to be a promising avenue to explore in the future for parallelizing other spatial computations as well. Although in this paper we have not handled the degenerate cases for plane sweep algorithm, they can be dealt with the same way we would deal with degenerate cases in the sequential plane sweep approach. Degenerate cases arise due to the assumptions that we had made in the plansweep algorithm. However, it remains one of our future work to explore parallel and directive based methods to handle such cases.

Acknowledgements. This work is partly supported by the National Science Foundation CRII Grant No. 1756000. We gratefully acknowledge the support of NVIDIA Corporation with the donation of the Titan X Pascal GPU used for this research. We also acknowledge XSEDE for providing access to NVidia Tesla P100 available in PSC Bridges cluster.

Appendix A Artifact Description Appendix

A.1 Description

Check-List (Artifact Meta Information)

- **Algorithm:**
  All algorithms are mentioned and described in the paper itself and can be referred to in Algorithms 1 and 3.
- **Program:**
  The Computational Geometry Algorithms Library (CGAL) and Geometry Engine Open Source (GEOS) were external libraries that were used.
- **Compilation:**
  Compilations were done using the g++ compiler and pgc++ compilers.
  for OpenACC: pgc++ -acc -ta=tesla:cc60 -o prog prog.cpp
  for OpenMP: g++ -fopenmp -o prog prog.cpp
  for CGAL: g++ -lcgal -o prog prog.cpp
  for GEOS: g++ -lgeos -o prog prog.cpp
- **Hardware:**
  Description of the machines used to run code can be found in Sect. 5.1 for further information.
- **Publicly available:**
  CGAL, GEOS, OpenMP, OpenACC, gcc and pgcc are all publicly available.

How Software Can Be Obtained (if Available). All of the software and code we used to build up our experiments were freely and publicly available. However, our code implementation can be found in the website: https://www.mscs.mu.edu/~satish/mpiaccgis.html.

Hardware Dependencies. To be able to get the most out of OpenMP, a multicore CPU would be needed. And to be able to run OpenACC kernels a GPU would be needed.

Software Dependencies. CGAL, GEOS, OpenMP and OpenACC libraries must be installed. Compilers like gcc and pgcc are also needed.

Datasets. Real world spatial data were used and datasets containing random lines were generated. Please refer to Sect. 5.1 for more information. Generated datasets are also posted in the website: https://www.mscs.mu.edu/~satish/mpiaccgis.html, however they can be generated on your own.

A.2 Installation

1. Configure the multicore CPUs and GPU to run on your system
2. Install the necessary libraries
3. Download or generate the necessary datasets
4. Download the code
5. Check that the datasets are in the proper directory pointed by the code, if not then fix it
6. Compile the code
7. Execute the compiled executable

References

1. Agarwal, D., Puri, S., He, X., Prasad, S.K.: A system for GIS polygonal overlay computation on linux cluster - an experience and performance report. In: 26th IEEE International Parallel and Distributed Processing Symposium Workshops & PhD Forum, IPDPS 2012, Shanghai, China, 21–25 May 2012, pp. 1433–1439 (2012). https://doi.org/10.1109/IPDPSW.2012.180
2. Aghajarian, D., Prasad, S.K.: A spatial join algorithm based on a non-uniform grid technique over GPGPU. In: Proceedings of the 25th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems, p. 56. ACM (2017)
3. Atallah, M.J., Goodrich, M.T.: Efficient plane sweeping in parallel. In: Proceedings of the Second Annual Symposium on Computational Geometry, pp. 216–225. ACM (1986)
4. Audet, S., Albertsson, C., Murase, M., Asahara, A.: Robust and efficient polygon overlay on parallel stream processors. In: Proceedings of the 21st ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems, pp. 304–313. ACM (2013)

5. Bentley, J.L., Ottmann, T.A.: Algorithms for reporting and counting geometric intersections. IEEE Trans. Comput. **9**, 643–647 (1979)
6. Franklin, W.R., Narayanaswami, C., Kankanhalli, M., Sun, D., Zhou, M.C., Wu, P.Y.: Uniform grids: a technique for intersection detection on serial and parallel machines. In: Proceedings of Auto Carto, vol. 9, pp. 100–109. Citeseer (1989)
7. Goodrich, M.T.: Intersecting line segments in parallel with an output-sensitive number of processors. SIAM J. Comput. **20**(4), 737–755 (1991)
8. Goodrich, M.T., Ghouse, M.R., Bright, J.: Sweep methods for parallel computational geometry. Algorithmica **15**(2), 126–153 (1996)
9. Khlopotine, A.B., Jandhyala, V., Kirkpatrick, D.: A variant of parallel plane sweep algorithm for multicore systems. IEEE Trans. Comput.-Aided Des. Integr. Circ. Syst. **32**(6), 966–970 (2013)
10. McKenney, M., Frye, R., Dellamano, M., Anderson, K., Harris, J.: Multi-core parallelism for plane sweep algorithms as a foundation for GIS operations. GeoInformatica **21**(1), 151–174 (2017)
11. McKenney, M., De Luna, G., Hill, S., Lowell, L.: Geospatial overlay computation on the GPU. In: Proceedings of the 19th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems, pp. 473–476. ACM (2011)
12. McKenney, M., McGuire, T.: A parallel plane sweep algorithm for multi-core systems. In: Proceedings of the 17th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems, pp. 392–395. ACM (2009)
13. OSM: OpenStreet Map Data (2017). http://spatialhadoop.cs.umn.edu/datasets.html
14. Prasad, S., et al.: Parallel processing over spatial-temporal datasets from geo, bio, climate and social science communities: a research roadmap. In: 6th IEEE International Congress on Big Data, Hawaii (2017)
15. Puri, S., Prasad, S.K.: 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 (CCGrid)(CCGRID), pp. 576–585, May 2015. https://doi.org/10.1109/CCGrid.2015.43
16. Puri, S., Paudel, A., Prasad, S.K.: MPI-vector-IO: parallel I/O and partitioning for geospatial vector data. In: Proceedings of the 47th International Conference on Parallel Processing, ICPP 2018, pp. 13:1–13:11. ACM, New York (2018). https://doi.org/10.1145/3225058.3225105
17. Puri, S., Prasad, S.K.: Output-sensitive parallel algorithm for polygon clipping. In: 43rd International Conference on Parallel Processing, ICPP 2014, Minneapolis, MN, USA, 9–12 September 2014, pp. 241–250 (2014). https://doi.org/10.1109/ICPP.2014.33