OpenACC Based GPU Parallelization of Plane Sweep Algorithm for Geometric Intersection

Anmol Paudel

Satish Puri

Marquette University

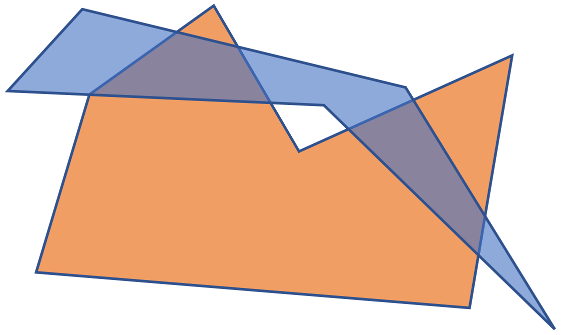
Milwaukee, WI

Introduction

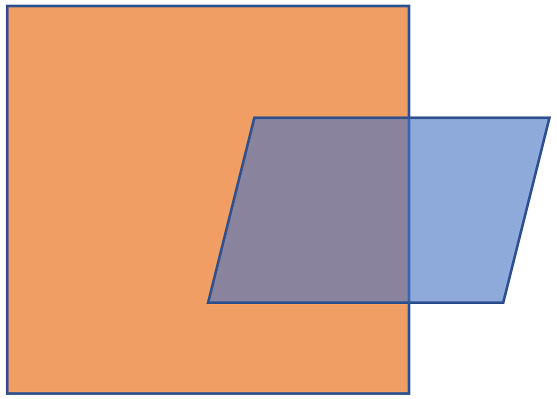
- Scalable spatial computation on high performance computing (HPC) environment has been a long-standing challenge in computational geometry.
- Harnessing the massive parallelism of graphics accelerators helps to satisfy the time-critical nature of applications involving spatial computation.
- Many computational 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

Introduction (contd.)

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

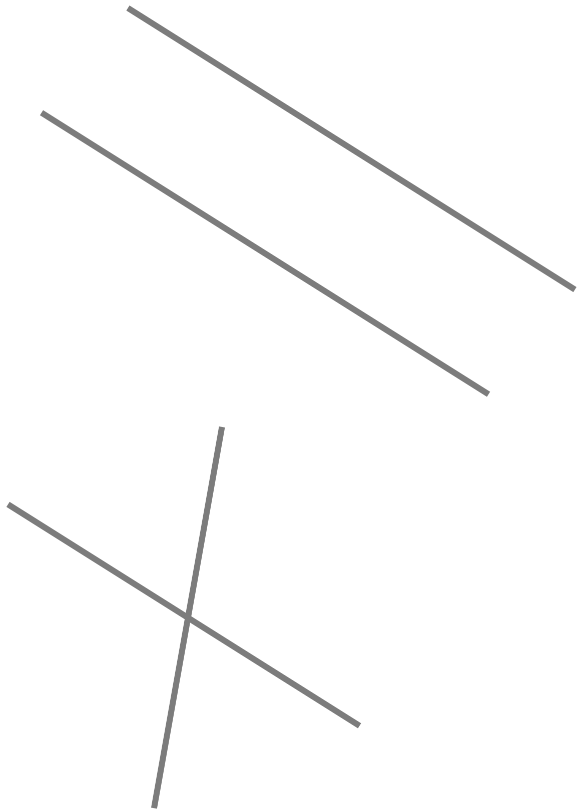


- Line segment intersection problem is one of the most basic problem in spatial computing and all other operations for bigger problems like polygon overlay or polygon clipping depends on results from it



Line Segment Intersection Problem

- The line segment intersection problem basically asks two questions –
  - Intersection detection problem
    - “are the line segments intersecting or not?”
- And if they are intersecting
  - intersection reporting problem
    - “what are the points of intersection?”
- We present an algorithmic solution for the second



Some Common Methods

- Simple brute force method
- Filter and refine method that uses a heuristic to avoid unnecessary intersection computations
- Plane Sweep

Contribution

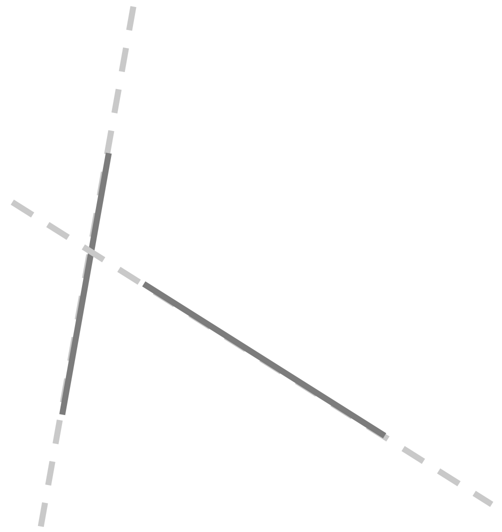
- To the best of our knowledge, this is the first work demonstrating an effective parallelization of plane sweep on GPUs
- A reduction based technique to find neighbors in the sweepline to reduce the added complexities of parallelization
- Completely directives-based implementation of all algorithms

Plane Sweep

- Technique to solve computational geometry problems by sweeping through the problem space
- Plane Sweep reduces $O(n^2)$ segment to segment pair-wise computation into
  - O(n*log*n) for identification
  - O(n + k*log*n) for reporting, good algorithm when k << n
- Works best if the dataset can fit in memory
- Parallelization difficult due to the in-order sequential processing of events stored in a binary tree and a priority queue data structure.
- Widely used in many other computational geometry problems like Voronoi diagram or Delaunay triangulation

In Computational Geometry

- Lines in computer application are usually finite lines with start and end points – not just y = mx + c
- Finding line intersection in computers might not be as simple as solving two mathematical equations.
- 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)



Directive-based Programming

- I would usually have a slide here discussing about Directive-based programming and why we choose that route
- But since this is WACCPD …
- Let’s just say directives are the future of accelerators and parallel programming

Algorithm 1 Naive Brute Force

Load all lines to L
for each line l_1 in L **do**
    for each line l_2 in L **do**
        Test for intersection between l_1 and l_2
        if intersections exists **then**
            calculate intersection point
            store it in results
        end if
    end for
end for

Algorithm 2 Plane Sweep

Load all lines to L

Initialize a priority queue (PQ) for sweeplines which retrieves items based on the y-position of the item

Insert all start and end points from L to PQ

Initialize a sweepline

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

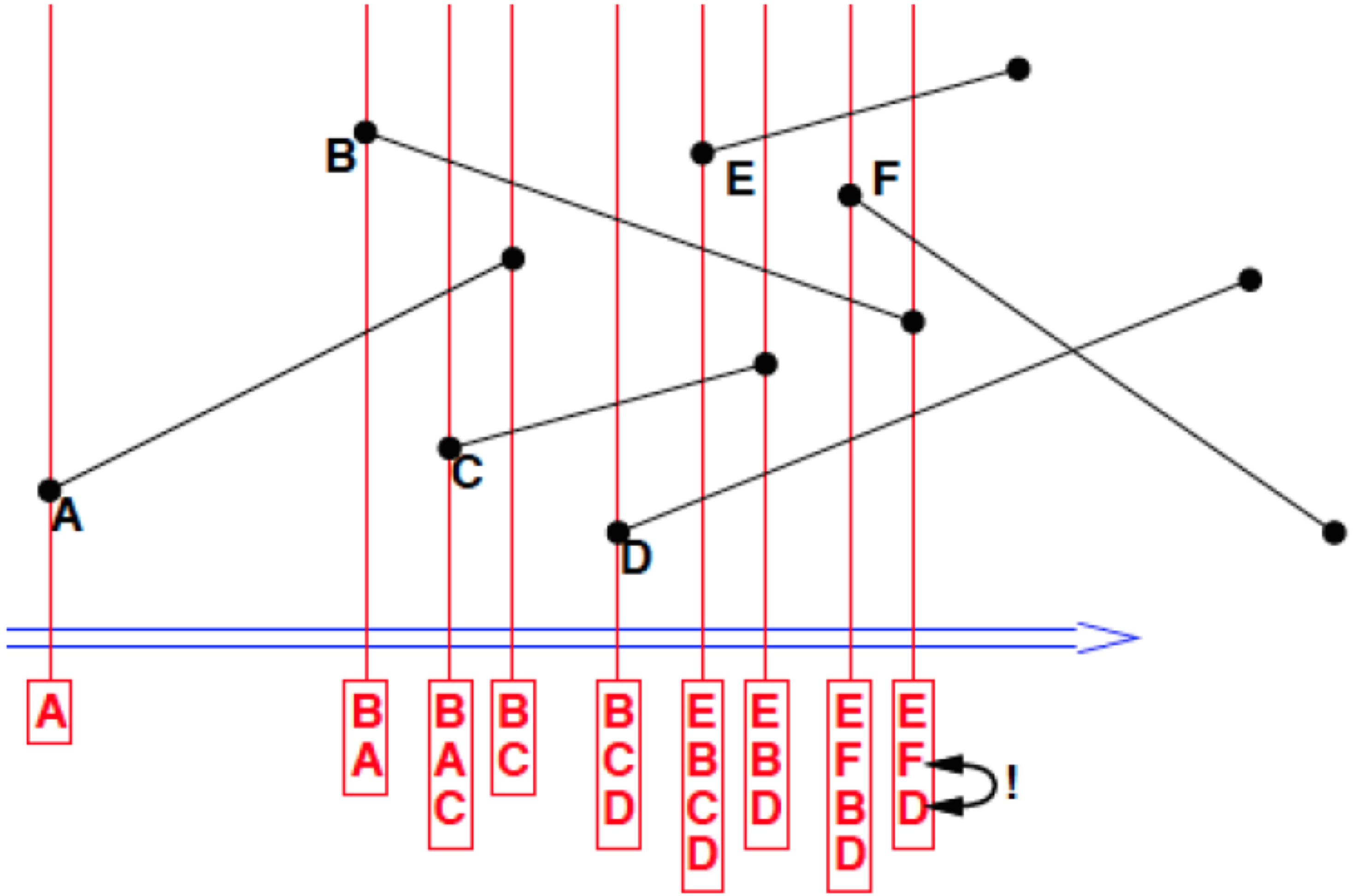
        HandleEndEvent(RemovedSegment)

    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



A
B
C
D
E
F
A
B A
B A C
B C
B C D
E B C D
E B D
E F B D
E F D
!

Algorithm 4 StartEvent Processing

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
end procedure

Algorithm 5 EndEvent Processing

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

Algorithm 6 IntersectionEvent Processing

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
end procedure

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

Start Event Sweeplines

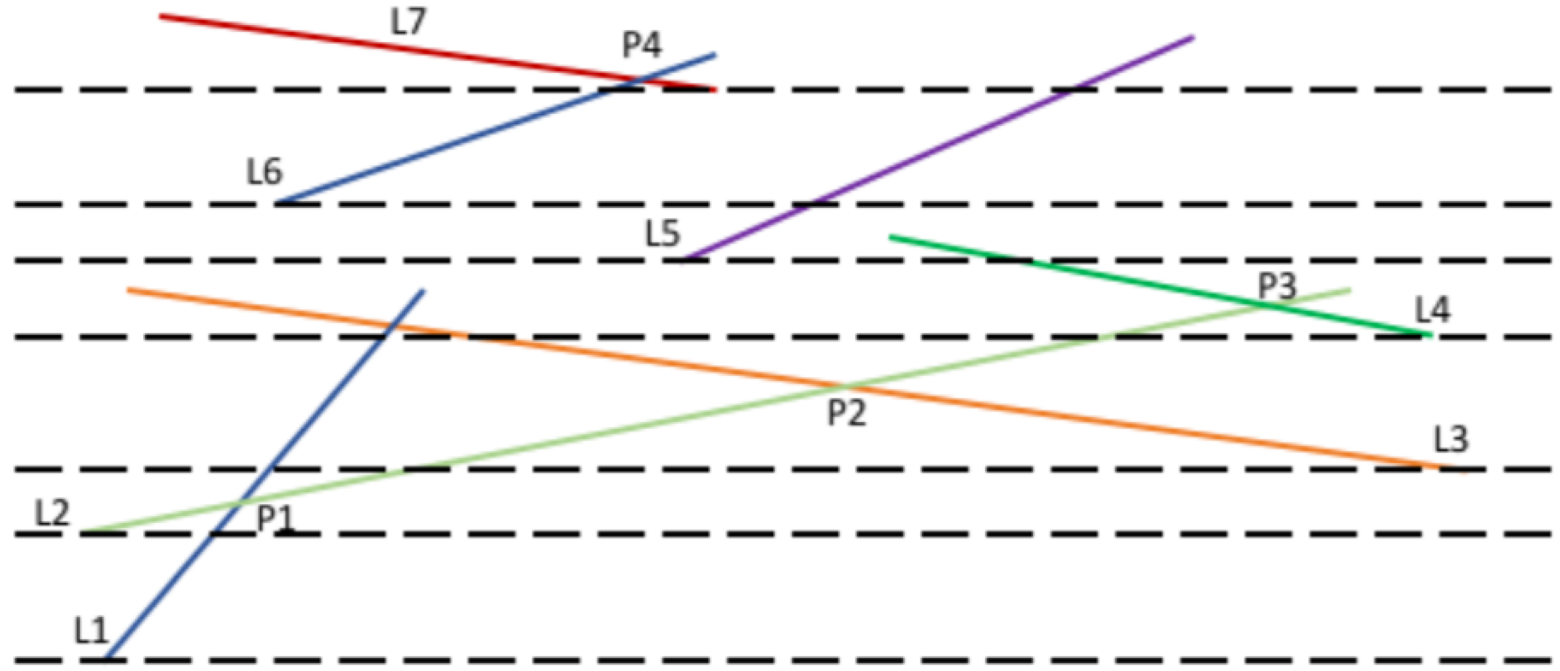


Fig. 2. Vertical Plane Sweep

Algorithmic Analysis

- Time Complexity
  - each of the N lines will have two sweeplines => $2N^2$ comparison steps
  - each of the K intersection event will also produce a sweepline => K*N steps
  - total is $2N^2 + K * N$ steps.
  - Assuming K << N, the time-complexity of this algorithm is $O(N^2)$
- Space Complexity
  - There will be 2N sweeplines for N lines
  - K sweeplines for K intersection events.
  - Total Memory requirement will be 2N +K
  - Assuming K << N, the space-complexity of the algorithm is O(N).

Algorithm 7 Reduction-based Neighbor Finding

```
Let SL be the sweepline
Let x be the x-coordinate in SL around which neighbors are needed
L ← all lines
prev ← MIN , nxt ← MAX
for each line l in L do-parallel reduction(maxloc:prev, minloc:nxt)
    if intersects(l,SL) then
        h ← intersectionPt(l,SL)
        if h < x then
            prev = h
        end if
        if h > x then
            nxt = h
        end if
    end if
end for
```

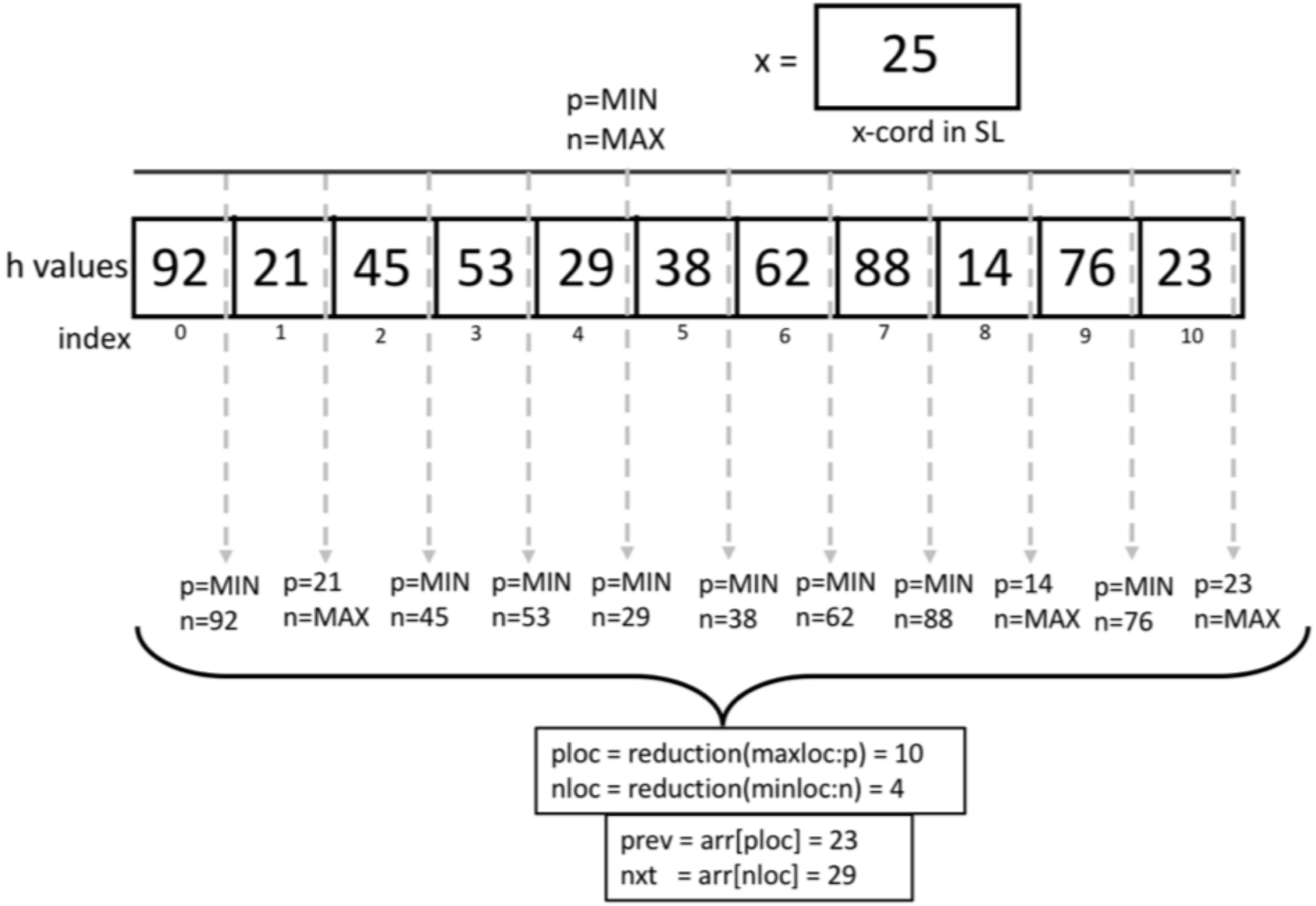


Fig. 3. Reduction-based Neighbor Finding

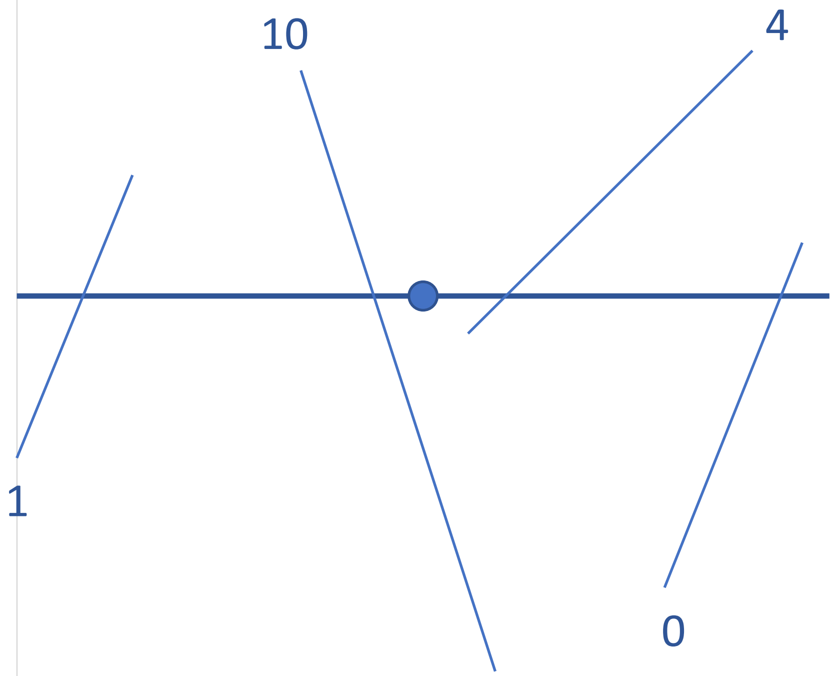


Table 2. Description of real-world datasets.

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

Segment Intersection Phases

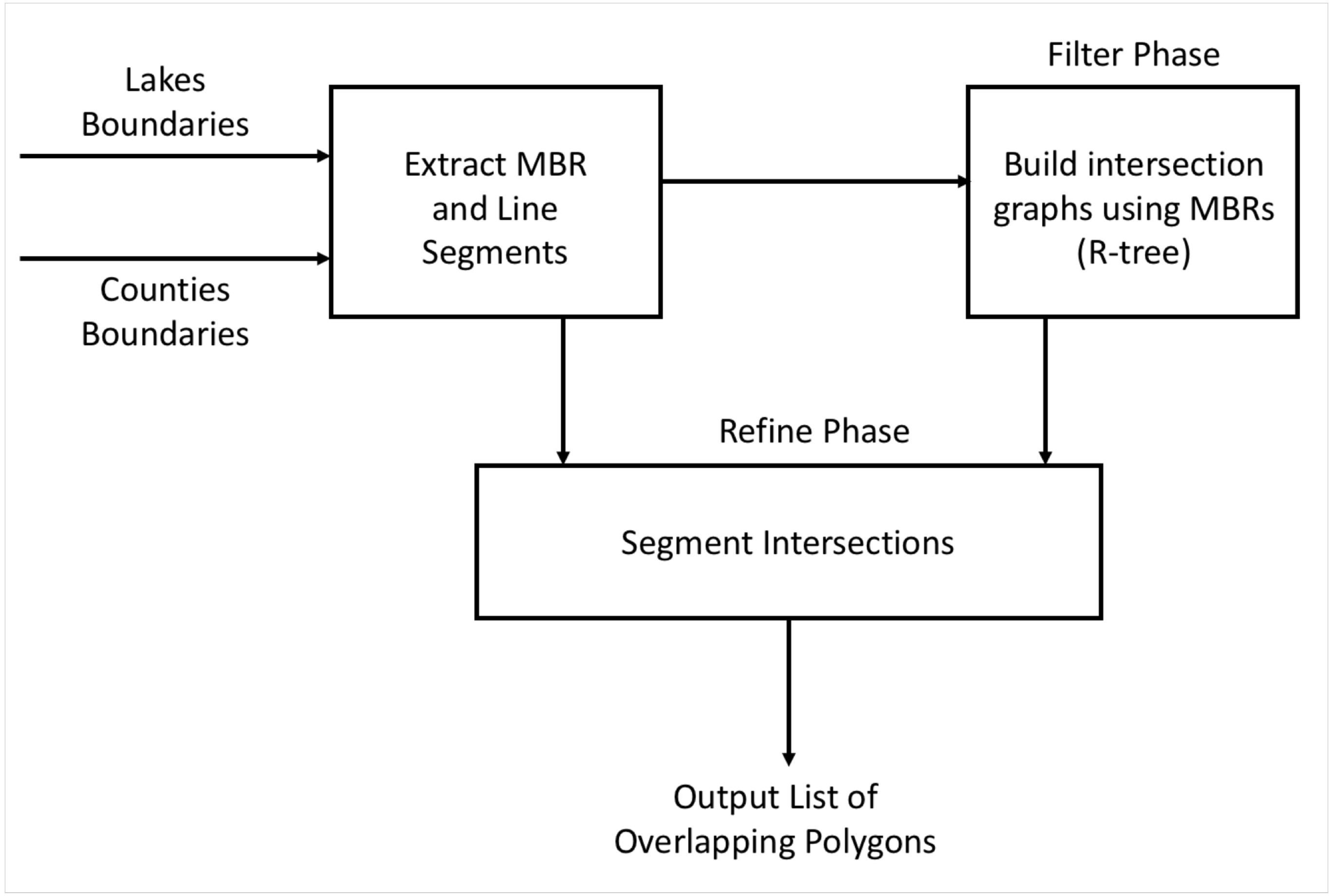


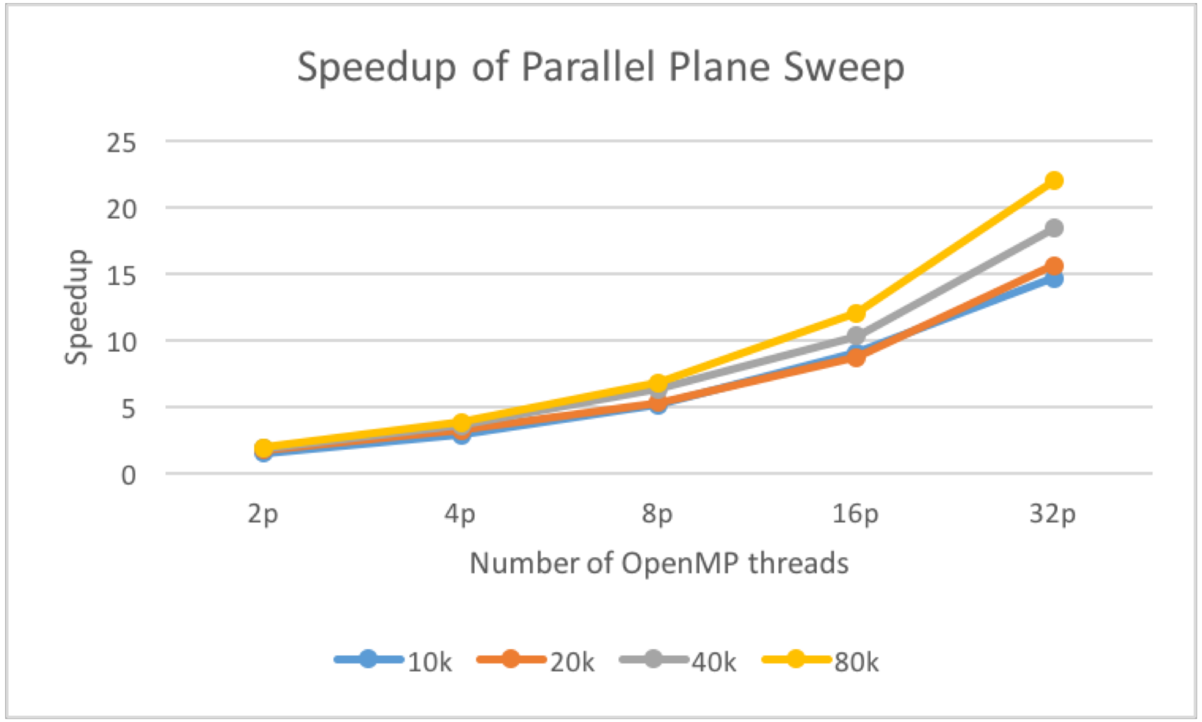
Table 4. Performance comparison of polygon intersection operation using sequential and parallel methods on real-world datasets.

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

Table 1. Dataset and corresponding number of intersections

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

Testing parallelizability with OpenMP



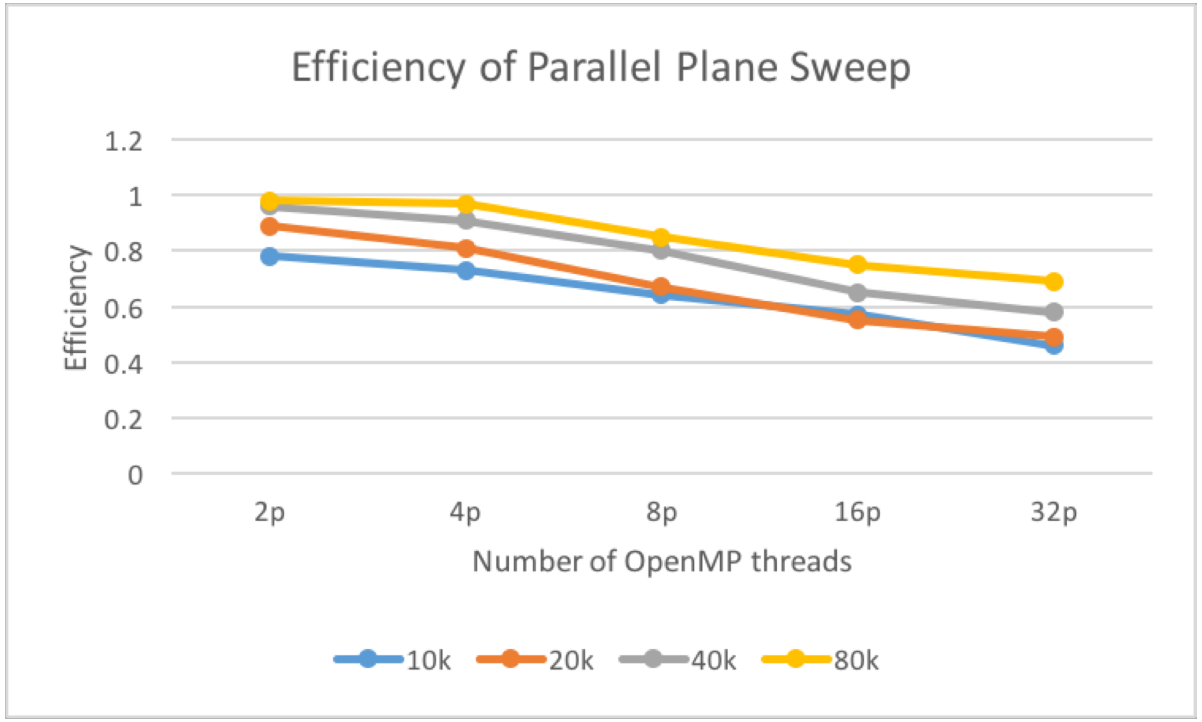
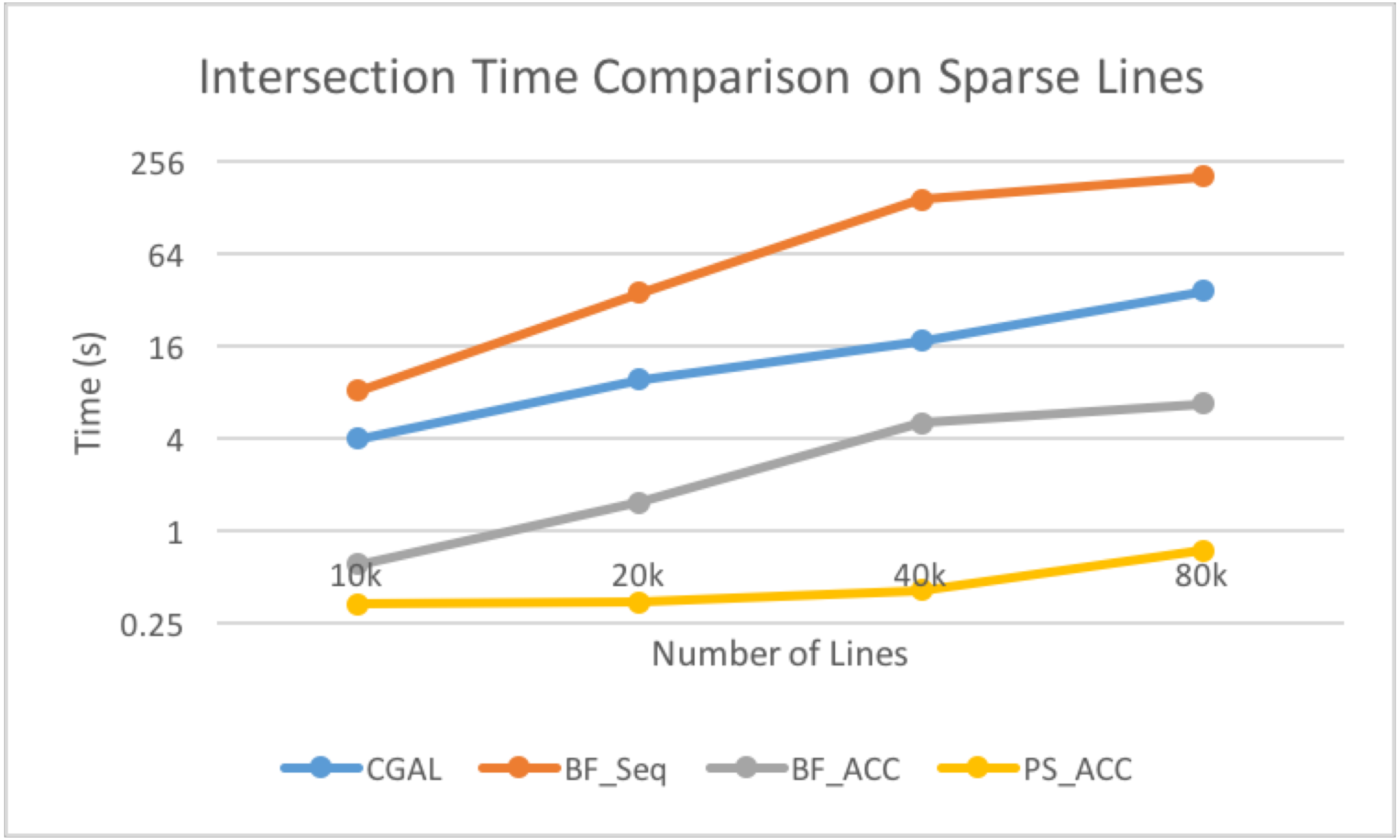


Table 3. CGAL, naive Sequential vs OpenACC on sparse lines

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

Table 8. Cuda vs OpenACC Parallel Plane Sweep on sparse lines

Lines	Cuda	PS-ACC
10k	0.23s	0.24s
20k	0.31s	0.25s
40k	0.65s	0.31s
80k	0.68s	0.65s


Intersection Time Comparison on Sparse Lines
256
64
16
4
1
0.25
Time (s)
10k
20k
40k
80k
Number of Lines
CGAL
BF_Seq
BF_ACC
PS_ACC

Machines Used

- Everest cluster at Marquette University
  - This machine was used to run the OpenMP codes and on the Intel Xeon E5 CPU v4 E5-2695
- Bridges cluster at the Pittsburgh Supercomputing Center
  - A single GPU node of this cluster was used which contained the NVIDIA Tesla P100
- NCSA ROGER Supercomputer
  - Sequential GEOS and OpenMP code was run on Intel Xeon E5-2660v3
  - GPU experiments using OpenACC on Nvidia Tesla P100

Conclusion

- 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

- Directives prove to be a promising avenue to explore in the future for parallelizing other spatial computations as well.

THANK YOU