Acceleration of Computational Geometry Algorithms for High Performance Computing Based Geo-Spatial Big Data Analysis

Anmol Paudel
Marquette University

ACCELERATION OF COMPUTATIONAL GEOMETRY ALGORITHMS FOR HIGH PERFORMANCE COMPUTING BASED GEO-SPATIAL BIG DATA ANALYSIS

by

Anmol Paudel, B.E., M.S.

A Dissertation submitted to the Faculty of the Graduate School,
Marquette University,
in Partial Fulfillment of the Requirements for
the Degree of Doctor of Philosophy

Milwaukee, Wisconsin

May 2022

ABSTRACT
ACCELERATION OF COMPUTATIONAL GEOMETRY ALGORITHMS
FOR HIGH PERFORMANCE COMPUTING BASED
GEO-SPATIAL BIG DATA ANALYSIS

Anmol Paudel, B.E., M.S.

Marquette University, 2022

Geo-Spatial computing and data analysis is the branch of computer science that deals with real world location-based data. Computational geometry algorithms are algorithms that process geometry/shapes and is one of the pillars of geo-spatial computing. Real world map and location-based data can be huge in size and the data structures used to process them extremely big leading to huge computational costs. Furthermore, Geo-Spatial datasets are growing on all V's (Volume, Variety, Value, etc.) and are becoming larger and more complex to process in-turn demanding more computational resources. High Performance Computing is a way to breakdown the problem in ways that it can run in parallel on big computers with massive processing power and hence reduce the computing time delivering the same results but much faster.

This dissertation explores different techniques to accelerate the processing of computational geometry algorithms and geo-spatial computing like using Many-core Graphics Processing Units (GPU), Multi-core Central Processing Units (CPU), Multi-node setup with Message Passing Interface (MPI), Cache optimizations, Memory and Communication optimizations, load balancing, Algorithmic Modifications, Directive based parallelization with OpenMP or OpenACC and Vectorization with compiler intrinsic (AVX). This dissertation has applied at least one of the mentioned techniques to the following problems. Novel method to parallelize plane sweep based geometric intersection for GPU with directives is presented. Parallelization of plane sweep based Voronoi construction, parallelization of Segment tree construction, Segment tree queries and Segment tree-based operations has been presented. Spatial autocorrelation, computation of getis-ord hotspots are also presented. Acceleration performance and speedup results are presented in each corresponding chapter.

ACKNOWLEDGMENTS

Anmol Paudel, B.E., M.S.

First and foremost, I would like to acknowledge and express my utmost gratitude to my Ph.D. advisor Dr. Satish Puri for his continuous mentorship and invaluable guidance throughout my graduate studies. I would then like to thank the members of my committee, Dr. Sheikh Iqbal Ahamed and Dr. Praveen Madiraju for their time, comments, feedback and encouragement. I would also like to extend my gratitude to Dr. Stephen J. Merrill, Dr. Gary Krenz, Dr. Daniel Rowe and Dr. Thomas J. Schwarz for their continuous support during my graduate studies.

My parents deserve all the credit for what I am able to accomplish today. I would like to thank my Mom Sangita, Dad Bishnu and Brother Aniket for always supporting, encouraging and loving me.

I would also like to acknowledge and express gratitude to my cousin big-brother Lokesh and his wife Barsha and their kids for their love and support.

I would not have made it without my awesome group of friends here in Milwaukee who became my second family. I would like thank all of them for their friendship and support.

I would also like to thank all my mentors during my various research internships and the collaborators I worked with.

Furthermore, I would like to thank all my teachers, mentors, friends, relatives and extended family back home who always wished me best.

The research carried out in this dissertation is partly supported by the National Science Foundation (NSF) CRII Grant No.1756000. Computing resources were partly made available via the XSEDE system and the high-performance computing facility "Raj" funded by the NSF award CNS-1828649 and Marquette University.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ALGORITHMS

LIST OF CODE LISTINGS

CHAPTER 1: INTRODUCTION AND BACKGROUND

There are different ways to accelerate computation. First and foremost would be to write better code, use better libraries or use better algorithms. [4] Next would be the use of compiler level optimizations. Then the computations could be executed on faster hardware but there are physical limits to how fast a single processor core can be manufactured. With multicore processors, parallelization techniques could be implemented to speedup computation. This can usually be done with concurrency and threads. Again there are physical limits to the number of cores that can put into a single processor. So, multiple processor and something like the Message Passing Interface (MPI) to manage communication among different processors could be the next step. Furthermore, accelerator hardware like manycores GPU can be used to offload some of the computation. GPUs are extremely useful and efficient in doing Single Instructions Multiple Data (SIMD) computations but have some data transfer overheads. GPU coding also requires rewriting the kernels in CUDA so they have coding effort overheads too. In the CPUs, we can further use vectorization using intrinsics to vectorize the code so that the operation run concurrently. Also, focus on the memory hierarchy and cache organisation can lead to developing algorithms that are either cache aware or oblivious and reduce the memory movement overheads. Also, in a distributed setup, communication reducing or communication avoiding can reduce the communication costs which are usually a big part of distributed computing. Proper load balancing among the nodes can also lead to more efficient computation. Targeting memory IO patterns based on the existing hardware or file systems and avoiding writes to disk as much as possible because writes are far more costlier than reads, can also lead to better total computation time on a distributed system.

1.1 Motivation

GeoSpatial computing is the computing related to location and geographic data. It includes datasets huge size like of maps of a territory and all the features in it (like roads, buildings, lakes, rivers) or high definition satellite imagery, etc. With the explosion of personal and mobile devices with location sensor like smart phones, smart cars, peripherals and gadgets and IOT devices like smart home appliances, smart industrial equipments, remote sensing infrastructures etc., the data collected with location information is enormous. Efficient parallel computational geometry algorithms are needed to process such huge datasets swiftly. Computational Geometry Algorithms are used in a variety of areas like GIS, image processing, data analytics, etc. Specially in the field of GeoSpatial computing, dataset can be extremely large and would require computationally efficient algorithms that are parallelizable and memory efficient.

Plane Sweep is an algorithmic technique in computational geometry where a sweep line scans through the search space to keep track of computations. We have successfully parallelized plane sweep for geometric intersections using directives.

Voronoi diagrams are a partition technique in where the space is partitioned into boundaries from a given set of points and a constraint that each point inside the partition area must be closest to the given input point. We have used directives to parallelize the Fortune's algorithm which is a plane sweep based voronoi computation algorithm.

Segment Trees are static data structure used to store line segments or intervals. They can then be used to query points and windows and then the queries can be used in other complex geometric operations. We have been able to parallelize building and querying on Segment Trees on both CPUs and GPUs.

Spatial autocorrelation are a way to calculate the statistical relationship between two points/boundaries/geometry in space based on some weight metric and their relative position. One of the most common spatial autocorrelation computation is the calculation of hotspots of spatial data.

1.2 Broader Impact

Accelerated algorithms in GeoSpatial computing can lead to faster data processing. It will reduce the time for scientific discovery allowing scientist and researchers to do more impactful work in the same duration of time. Moreover, in time critical situation like planning a rescue mission in an evolving disaster zone, being able to process as much of the data in as short amount of time as possible can directly correlate success of the rescue mission and number of lives saved. In climate related disaster zones, the conditions changes rapidly, so being able to incorporate new data which are usually global and enormous in proportion and recompute the latest scenario to reformulate plans is extremely essential.

Also for businesses that use location data to provide services like ride-sharing or deliveries, being able to process all the continuous big data in a faster and more efficient manner can lead to better and quicker customer satisfaction which could translate to better business. Also, with the increasing trend of such companies trying to reduce service time to as short as possible, faster and more efficient algorithms could be a key to achieving that.

Furthermore, faster algorithms free up computing resources and personnel time allowing exploration and experimentation into different frontiers that could push the bounds of human discovery.

1.3 Background (Acceleration Techniques)

1.3.1 Directive Based Parallelization

Portions of serial code written in programming languages like C/C++ can be made parallel by the use of compiler directives. Compiler directives are addendum to the existing code that provides hints to the compiler on how to parallelize the code. Directives can be extremely useful in parallelizing code if a sequential codebase already exists. We can identify areas in the code that can be parallelized using the existing directives and with some minor modification to the code, we can get a parallel version. Even though a sequential codebase maynot exist, it is still useful because many of the available algorithms are inherently sequential. Compiler directives can also be used to write parallel code from the start. Directives based parallelization also reduce the overhead of learning different parallelization techniques because it can be the one shot solution to both writing parallel code or modifying serial code to parallel. The two most common directive based parallelization libraries are OpenMP and OpenACC. OpenMP is most commonly used to parallelize code for multicore processing using threads and OpenACC is most commonly used to parallelize code for manycore processing, especially NVIDIA GPUs. OpenMP could be an alternative to using the programming language's threads library. OpenACC is an alternative to using CUDA.

1.3.2 GPU Parallelization

GPUs are collection of manycores processors. GPUs were original used to offload and accelerate graphics processing off the CPU but due to their extremely efficient ability to be able accelerate SIMD computation they have been used in General Purpose GPU (GPGPU) programming and applied to scientific computing. CUDA is a programming language used to write GPGPU code for NVIDIA based

GPUs. Beyond just offloading computation, GPUs have a lot of opportunity of optimization using warp level parallelization, warp level load balancing or memory access optimizations. An extreme scale of acceleration would GPU based parallelization with large number of GPUs in distributed setup with numerous nodes having multiple GPUs in each node. Some message passing like protocol is necessary to manage the computation distribution and communication among the nodes.

1.3.3 Memory Movement Optimization

Computation is performed at the core of the processor. However, data can be at one of the many levels of the memory hierarchy. Memory hierarchy refers to different levels of memory further away from on the core, each level away may have greater size but slower access to the processing core. Different levels of caches are some of the closest and fastest accessible memory. This field of research started with a class of algorithms known as "External Memory Algorithms". Accessing each level away in the memory hierarchy is costlier, so algorithms that can load memory through the hierarchy in an efficient way would perform better. For parallel and distributed computing, this becomes an even bigger challenge because there can be shared memory cores and distinct processor nodes on a network. Given the technology trends of faster processors not having caught up to memory devices or memory transfer devices at the same rate, this movement of data among the memory during computation becomes the bigger bottleneck.

Memory inefficiencies in cache, communication or network tend to be the bigger bottleneck in extreme scale computing when compared to the costs of computation. Even in the area of memory, writes (writing to memory) usually are a factor of times costlier in time and energy than reads (reading from memory).

Moving data between levels of a memory hierarchy or between processors over a network, is comparatively much more expensive than computation. A lot of

work has been done in minimize communication and attain lower bounds but most of the work focuses on the total number of reads plus writes and does not distinguish between the two. Writes, however, can be much more expensive than reads in many storage devices such as nonvolatile memories. Technological trends are increasing the gap in costs between computation and communication. Memory devices like NVM are being used in many scientific applications and extreme scale computer clusters. Phase Change Memory is a type of NVM where a write is 15 times slower than a read both in terms of latency and bandwidth and consumes 10 times as much energy. Another technology called CBRAM uses significantly more energy for writes (1pJ) than reads (50fJ). Writes to NVM can also be less reliable than reads, require multiple attempts for success, and can cause device wear out. [5]

1.3.4 Cache Based Optimizations

Cache-Oblivious Algorithms have a property that to get good performance, tuning of variables dependent on hardware parameters, such as cache size and cache-line length are not necessary. Nevertheless, these algorithms do optimal amount of work and move data optimally among multiple levels of cache. It has been shown that algorithms designed for 2 levels of cache generalizes to multiple levels of cache and are portable. Optimal cache-oblivious algorithms are known for the matrix multiplication. Further machine-specific tuning may be required to obtain nearly optimal performance in an absolute sense. The goal of cache-oblivious algorithms is to reduce the amount of such tuning that is required. Typically, a cache-oblivious algorithm works by a recursive divide and conquer algorithm, where the problem is divided into smaller and smaller subproblems. Eventually, one reaches a subproblem size that fits into cache, regardless of the cache size. For example, an optimal cache-oblivious matrix multiplication is obtained by recursively dividing each matrix into four sub-matrices to be multiplied, multiplying the

submatrices in a depth-first fashion. In tuning for a specific machine, one may use a hybrid algorithm which uses blocking tuned for the specific cache sizes at the bottom level, but otherwise uses the cache-oblivious algorithm.

A cache-oblivious algorithm is designed to perform well, without modification, on multiple machines with different cache sizes, or for a memory hierarchy with different levels of cache having different sizes. For matrix multiplication, cache-oblivious algorithms are competitive with cache-aware algorithms. The benefit with CO algorithms is that explicit tuning for three levels of memory hierarchy is not needed which is the case with cache-aware algorithms. We will evaluate these differences.

Cache-aware algorithms use hardware parameters such as level of each cache, size of each cache, cache-line length, cache latency and bandwidth and other hardware level details as inputs to the program optimize performance on a specific hardware. Given the parameters of the specific hardware, it maximizes the efficiency by moving data in the most optimal way. These parameters for hardware can be inferred by studying the architecture, by profiling, and by hardware counters.

1.3.5 Communication Avoiding

Communication avoiding (CA) algorithms are parallel algorithms that trade fast memory space for reducing inter-processor communication. Communication is the limiting factor in exploiting large scale parallelization. Consider the following running-time model:

Time taken per FLOP is γ

Time taken to move a word from slow to fast memory is β.

So, total running time is equal to:

γ * (no. of FLOPs) + β * (no. of words moved).

In computations where the right side expression is significantly greater than

the left side expression as measured in time, communication cost would dominate computation cost.

CA algorithms have been shown to be very effective in HPC applications involving linear algebra, n-body simulation, etc. In CA literature [6, 7], it has been shown that the parallel efficiency(speedup/processors) decreases dramatically as the number of processors are scaled up with no replication. With replication of the data among processors, communication time decreases, and better efficiency is realized in practice. In matrix multiplication upto 12x speedup (on thousands of processors) has been reported in literature. 2.5D algorithms interpolate between a 2D and 3D processor topology. The novelty in 2.5D algorithm is that it can use the extra memory available in a processor in a systematic manner to optimize communication pattern.

1.3.6 Intrinsics and Vectorization

Each core in modern processor can do streaming processing of contiguous memory. Advanced Vector Extensions (AVX) are extensions which allow the compiler to use vector registers available in the processor hardware for faster vector operations. Depending on the precision size of the data structure used, different number of vector operations can be performed at once. An added benefit to using vector extension is also that the memory access to contiguous memory is inherently more cache efficient. Object code can be generated to inspect the vectorization achieved with intrinsics. Vectorization can be compounded with thread level parallelization to get the most out of every core in the processor.

1.4 Background (Computational Geometry)

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. 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 [8].

Plane sweep algorithm and theoretical algorithms developed around 80's and 90's fall under the second category [9, 10, 11]. These theoretical PRAM algorithms attain near-optimal poly-logarithmic time complexity [9, 10, 12]. 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) [13].

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 [13, 14, 15]. 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. 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 [16, 17, 18].

Significantly speeding up the sequential Voronoi computation has remained a long standing challenge since Fortune came up with the planesweep approach that reduced the complexity of Voronoi computation to $O(n\log n)$. Delaunay triangulation has a dual relationship with Voronoi diagrams. The Delaunay triangulation for a set of discrete points is the connected graph of all the points in such way that no points lie inside the triangles formed by joining the points. A Voronoi diagram is basically the Delaunay triangulation of the vertices of the resultant Voronoi graph.

Biniaz and Dastghaibyfard compares different sweep line approaches like the Fortune's sweep-line algorithm, Zalik's sweep-line algorithm, and a sweep-circle algorithm proposed by Adam, Kauffmann, Schmitt and Spehner [19]. Wong and Muller presents an even more efficient implementation of the Fortune's algorithm [20]. Effort has been spent on tuning the code and paying attention to hotspots that slow down the implementation. Work done by Bollig explores Voronoi computations in the GPU using a flooding algorithm along with Lloyd's method [21]. Majdandzic et al. claims to presents a parallel algorithm and its GPU-based implementation to calculate a discrete approximation to the Voronoi diagram [22].

Yuan et al. explores the problem of using the GPU to compute the generalized Voronoi diagram for higher order sites, such as line segments and curves using the jump flooding algorithm and their improvements upon it [23]. The work done by Rong et al. explores a GPU-assisted computation of centroidal Voronoi tessellation using Lloyd's algorithm [24]. Tsidaev explores the use of Green-Sibson Voronoi tessellation method in the parallelization technique for Natural Neighbor interpolation algorithm [25]. Nievergelt and Preparata presents two plane sweep methods for merging geometric figures [26]. Theoretical parallel algorithms for Voronoi diagram construction have been designed on mesh, hypercube, PRAM

models of computation [27].

Segment tree data structure was introduced by John Louis Bentley in 1977 [28]. One of the important operations on a segment tree is a stabbing query, which takes a query point p as an input to report all the line segments overlapping with an imaginary vertical ray passing through x coordinate of p. For n line segments as input, the time complexity of building the tree is $O(n \log n)$ [28]. The space complexity of building the tree is $O(n \log n)$. The time complexity of the stabbing query is $O(\log n + k)$ where k is the number of line segments in the output of the query [28]. An important application of segment tree is in parallelizing plane sweep algorithms for computing line segment intersections using PRAM model [9, 10, 29]. External memory segment tree algorithms have been presented in [30, 31]. Parallel and distributed algorithms for segment tree data structure are presented in [32, 33, 34, 35, 36, 37]. Segment tree construction on a hypercube architecture to solve next element search problem (also known as first hit) is presented in [32]. A parallel algorithm using PRAM (Parallel Random Access Machine) model of computation and implementation on connection machine was presented in [34]. Bulk Synchronous Parallel (BSP) model has also been used to develop and analyze segment tree algorithms [35]. Segment tree was used in parallel PRAM algorithm for polygon clipping in [12]. The influence of caches on the performance of heap data structure was presented earlier by LaMarca and Ladner [38]. A variant of binary heap optimized for virtual memory environments was presented as B-heap [39]. B-heap keeps subtrees in a single page of virtual memory and performs well for large heaps.

In external memory algorithms, a segment tree variant has been designed to minimize data movement by increasing the fanout of the node and recursively splitting the node among its children [30, 40]. Compared to the earlier theoretical

work [10, 30, 40], we present practical algorithms that allow parallelism in computational geometry applications as well as minimize data movement between fast memory and slow memory.

A cache-oblivious method known as Van Emde Boas (vEB) layout has been presented earlier to store static binary search trees recursively in a cache-efficient manner to minimizes data movement in a query operation [41, 42, 43]. Conceptually, vEB layout transforms a static binary tree by recursively splitting the tree at the middle level of edges so that the tree nodes get grouped together to minimize data movement in search operations [42]. GPU has been used for implementing data structures like Btree [44] and computational geometry data structures like KD-tree [45], R-tree [46] and range tree [47].

The notion of spatial autocorrelation is related to first law of geography: Everything is related to everything else, but nearby things are more related than distant things [48]. The value of attributes at a given location tend to vary gradually over space. Events in a given area are influenced by the events at neighboring areas. In spatial statistics, this property is called spatial autocorrelation [49]. With the volume of data increasing due to its spatio-temporal nature, parallelization of existing algorithms have been done [50, 51, 52, 53]. Existing approaches use spatial partitioning methods like quadtree for parallelization [50]. A Matlab-based shared memory parallelization has been described in [53]. Hadoop MapReduce has been used to parallelize Getis-Ord based Hotspots detection problem using quadtree-based decomposition of spatial data [50]. Apache Spark framework has also been used to parallelize spatial hotspot computation [51, 52]. Spark MapReduce papers are short papers from GIS Cup competition organized with SIGSPATIAL conference [51, 52]. Hadoop and Spark based projects make good use of thread-level and coarse-grained parallelism but do not take full advantage of HPC

resources (e.g., SIMD, GPUs) thus leaving performance on the table [50, 51, 52].

CHAPTER 2: ACCELERATION OF PLANE SWEEP ALGORITHM FOR GEOMETRIC INTERSECTION

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.

2.1 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 [93]. 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 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 [94]. 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 chapter, 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 chapter 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 chapter is structured as follows. Section 2.2 presents a general technical background and related works to this chapter. Section 2.3 describes our parallel algorithm. Section 2.5 provides details on OpenMP and OpenACC implementations. Section 5 provides experimental results. Conclusion and future work is presented in Section 2.7. Acknowledgements are in the last section.

2.2 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.

2.2.1 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.

2.2.2 Naive Brute Force Approach

Like with any computational problem, the easiest approach is foremost the brute force approach. Algorithm 2.1 describes the brute force approach to find segment intersection among multiple lines.

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

Algorithm 2.1 Naive Brute Force

```
1: Load all lines to L
2: for each line l_1 in L do
3:     for each line l_2 in L do
4:         Test for intersection between l_1 and l_2
5:         if intersections exists then
6:             calculate intersection point
7:             store it in results
8:         end if
9:     end for
10: end for
```

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.

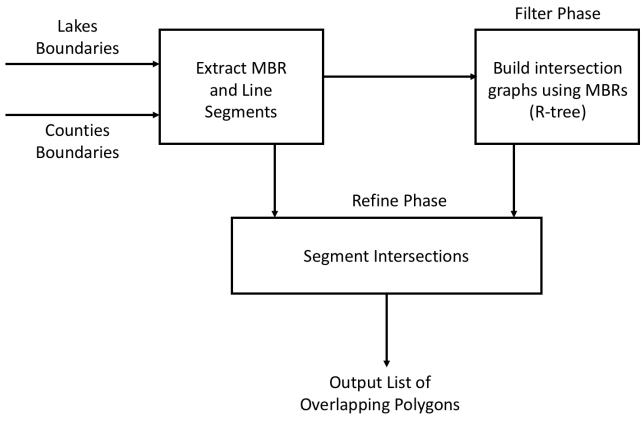

Figure 2.1: Polygon intersection using Filter and Refine approach

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. 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 [95]. We have employed this approach while handling real spatial data. Figure 2.1 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.

2.2.3 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 [8]. Computational geometry libraries typically use plane sweep algorithm in their implementations.

Algorithm 2.2 describes plane sweep using a vertical sweepline. The procedures for *HandleStartEvent*, *HandleEndEvent* and *HandleIntersectionEvent* used in Algorithm 2.2 are given in Algorithms 2.4, 2.5, 2.6 respectively. For simplicity in presentation, following assumptions are made in Algorithm 2.2:

1. No segment is parallel to the vertical sweeplines.
2. No intersection occurs at endpoints.

Algorithm 2.2 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:
6:     If the nextItem is startevent:
7:         The segment is added to the sweepline
8:          HandleStartEvent(AddedSegment)
9:     If the nextItem is endevent:
10:        The segment is removed from the sweepline
11:         HandleEndEvent(RemovedSegment)
12:    If the nextItem is intersection-event:
       [Note that there will be two contributing lines at intersection point.
13:    Let these two lines be l_1 and l_2.]
14:        HandleIntersectionEvent(l_1,l_2)
15:        Record the intersecting pairs
```

3. No more than two segments intersect in the same point.

4. No overlapping segments.

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

2.2.4 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 [9, 10, 11]. These theoretical PRAM algorithms attain near-optimal poly-logarithmic time complexity [9, 10, 12]. 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) [13]. 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 [13, 14, 15]. 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 [96], 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 [16, 17, 18]. 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.

2.2.5 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.

2.3 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 2.3 describes our modified version of plane sweep using a vertical sweepline. Figure 2.2 shows the startevent processing for a vertical bottom up sweep. Algorithm 2.3 also has the same simplifying assumptions like Algorithm 2.2. Step 2, step 3 and the for-loop in step 4 of Algorithm 2.3 can be parallelized using directives.

Algorithm 2.3 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 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 2.3, intersection event point processing

Algorithm 2.3 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 2.4 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 2.5 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 2.6 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**

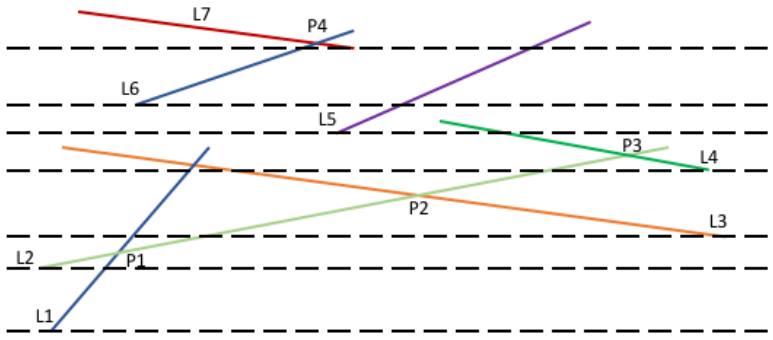

Figure 2.2: Vertical Plane Sweep
Vertical Plane Sweep showing sweeplines (dotted lines) corresponding to starting event points only. P1 to P4 are the intersection vertices found by processing start event points only. L1, L2 and L3 are the active line segments on the third sweepline from the bottom. Event 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.

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.

2.3.1 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 2.3 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.

2.4 Algorithmic Analysis

2.4.0.1 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 $K << N$, the time-complexity of this algorithm is $O(N^2)$.

2.4.0.2 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 $K << N$, the space-complexity of the algorithm is $O(N)$.

2.5 Directive-based Implementation Details

Although steps 2, 3 and 4 of Algorithm 2.3 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.

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

Listing 2.1: Data Structure for Point

```
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 2.2: Data Structure for Line

```
#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 2.3: Routine for Intersection Point

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 2.7 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 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 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 2.3 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 2.7 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

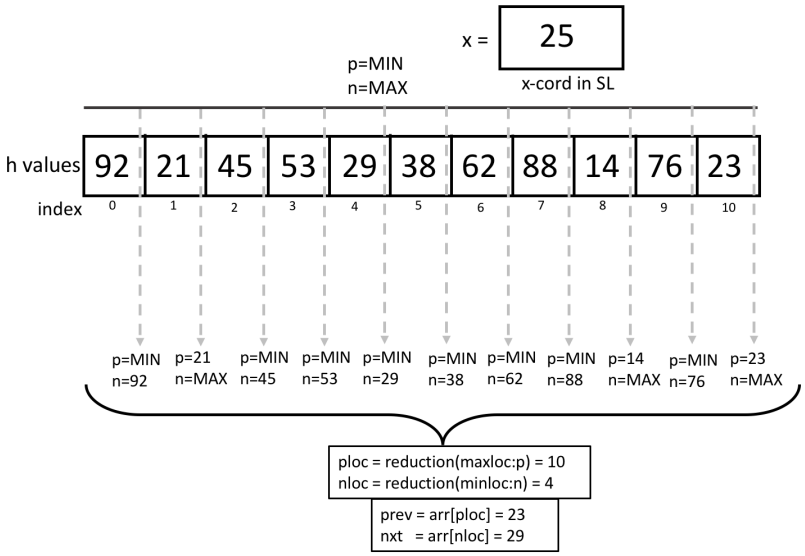

Figure 2.3: Reduction-based Neighbor Finding
Here the dotted lines are the parallel threads and we find the left and right neighbor to the given x-cord (25) on the sweepline and their corresponding indices. p and n are thread local variables that are initialized as MIN and MAX respectively. As the threads execute concurrently their value gets independently updated based on Algorithm 2.7.

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 PreparedGeometry 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.

2.6 Experimental Results

2.6.1 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 45MB cache and base frequency of 2.10GHz.

- 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 12GB.
- 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 2.1 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 2.4 depicts a randomly generated set of sparse lines.

Table 2.1: Dataset and corresponding number of intersections

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

Real-world Spatial Datasets: As real-world spatial data, we selected

[1] `https://trac.osgeo.org/geos/`

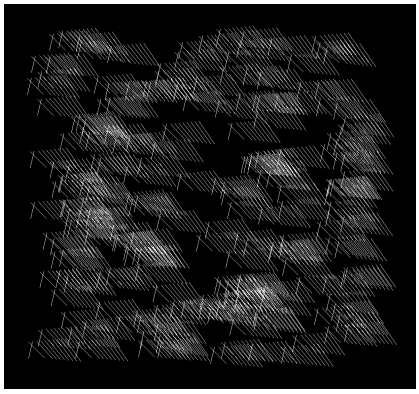

Figure 2.4: Randomly generated sparse lines

polygonal data from Geographic Information System (GIS) domain [2], [3] [97]. The details of the datasets are provided in Table 2.2.

Table 2.2: 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	$219K$	$60,046K$	$1.3GB$

2.6.2 Performance of Brute Force Parallel Algorithm

2.6.3 Using Generated Dataset:

Table 2.3 shows execution time comparison of CGAL, sequential brute-force (BF-Seq) and OpenACC augmented brute-force (BF-ACC) implementations.

Key takeaway from the Table 2.3 is that CGAL performs significantly better

[2]http://www.naturalearthdata.com
[3]http://resources.arcgis.com

Table 2.3: Execution time by 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

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.

2.6.3.1 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 2.4 shows the execution time for polygon intersection operation using three real-world shapefiles listed in Table 2.2. 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.

2.6.4 Performance of Parallel Plane Sweep Algorithm

Table 2.5 shows the scalability of parallel plane sweep algorithm using OpenMP on Intel Xeon E5. Table 2.6 is comparison of CGAL and parallel plane

Table 2.4: 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

sweep (PS-ACC). Key takeaway from the Table 2.6 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 2.3.

Table 2.5: Parallel plane sweep on sparse lines with OpenMP

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

Table 2.6: CGAL vs OpenACC Parallel Plane Sweep on sparse lines

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

2.6.5 Speedup and Efficiency comparisons

Table 2.7: 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

Table 2.7 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 2.5 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 2.6 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 2.7 shows the efficiency (speedup/threads) for the previous speedup graph. As we can see in the figure, 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.

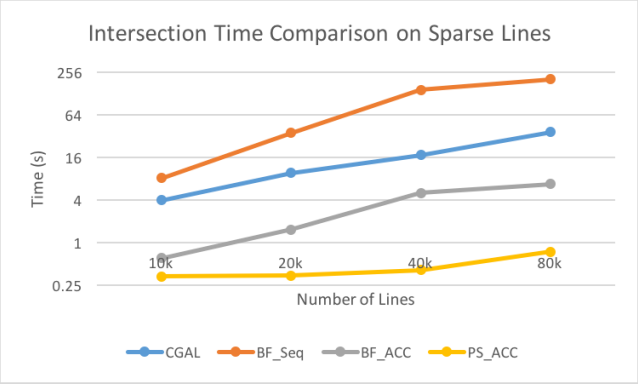

Figure 2.5: Time comparison for CGAL, sequential brute-force, OpenACC augmented brute-force and plane sweep on sparse lines

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

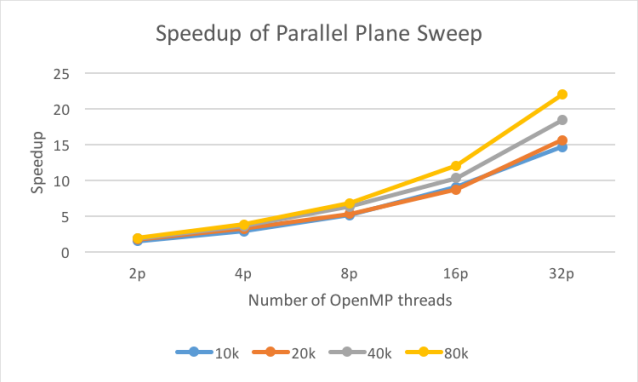

Figure 2.6: Speedups for the parallel plane sweep with varying OpenMP threads on sparse lines

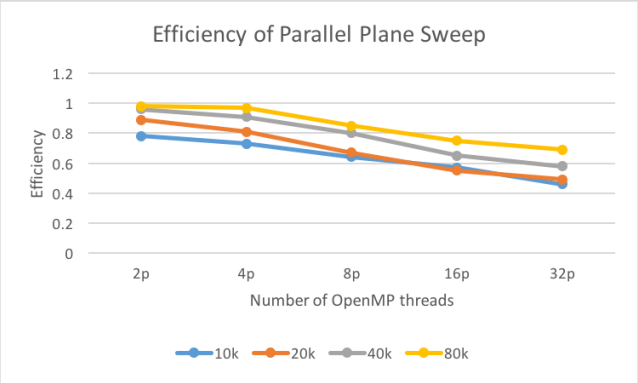

Figure 2.7: Efficiency of the parallel plane sweep with varying OpenMP threads on sparse lines

processing as mentioned in Algorithms 2.4 and 2.5. There are fewer intersection point events in comparison to the endpoint events.

2.7 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 [98].

Compiler directives prove to be a promising avenue to explore in the future for parallelizing other spatial computations as well. Although in this chapter 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.

CHAPTER 3: ACCELERATION OF PLANE SWEEP BASED VORONOI COMPUTATION

Voronoi diagram computation is a common and fundamental problem in computational geometry and spatial computing. Numerous algorithms and their corresponding implementations already exist along with multiple approaches to parallelize Voronoi computation. This chapter attempts the parallelization of Voronoi diagram construction by augmenting an existing sequential implementation of Fortune's planesweep algorithm with compiler directives. In doing so, it explores the possibilities and challenges of implementing directives-based parallelization of existing computational geometry implementations.

To the best of our knowledge, this is the first work that explores the possibility of exploiting concurrency available at each event of the planesweep algorithm. We have found and experimentally demonstrated that the maintenance of data structures associated with planesweep has enough computational steps to leverage shared memory parallelism on a multi-core CPU. On the Intel Xeon E5 CPU, our shared-memory parallelization with OpenMP achieves around 2x speedup compared to the sequential implementation using datasets containing 2k-128k sites. Finally, observations and potential ideas for exploiting more parallelism with directives are proposed.

3.1 Introduction

Partitioning an area into different regions is an important and well-studied problem. An area in a cartesian plane can be partitioned differently depending on whether it is for sales, marketing, voting, schooling, policing, etc. The method used for creating these partitions varies according to the application. For example, if an area had multiple emergency response centers, and we wanted to partition the area

into different regions based on distance from the centers in such a way that any point in the partitioned region would be closest to a center, then the resulting subdivisions for such a partitioning would be a Voronoi diagram. The input points are also known as sites in a Voronoi diagram and the edges between such partitioned regions are called Voronoi edges or segments.

Voronoi diagrams are extensively used in computational geometry to partition a plane into multiple regions where each region corresponds to and contain a site, and that site will be the closest site to all points in that region. Figure 3.1 shows a Voronoi diagram with a unique region for each site. Here is a mathematical definition of a Voronoi region:

Definition 1. *The Voronoi region R_k, associated with the site P_k is the set of all points in target region X whose distance to P_k is not greater than their distance to the other sites P_j where j is any index different from k. In other words, if $d(x, A) = infimum\{d(x, a) \mid a \in A\}$ denotes the distance between the point x and the subset A, then $R_k = \{x \in X \mid d(x, P_k) \leq d(x, P_j)$ for all $j \neq k\}$*

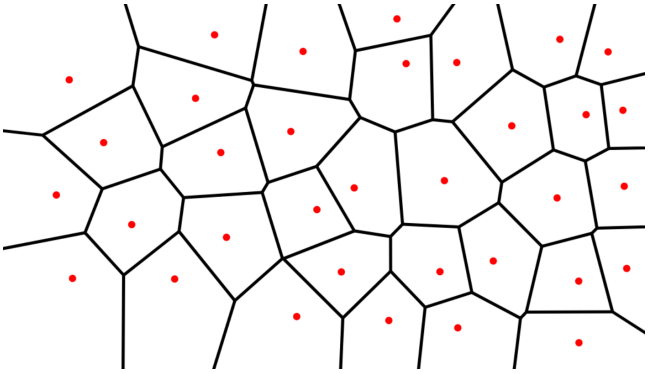

Figure 3.1: Voronoi Diagram
[The dots in the figure are the sites and the lines are the edges of the a partitioned region given that particular set of sites. It can be observed that for any arbitrary point in the whole space, the closest site is the one inside the same region as it is. Also, the number of regions are exactly equal to the number of sites and have a one-to-one correspondence.]

The strategy to partition a target area depends on the particular purpose of

partitioning; such as companies partition areas into regions for sales or for warehousing inventory. The rationale for partitioning dictates the strategy used and the regions and boundaries that we get can vary significantly for the same area partitioned with different strategy. An example of a strategy could be when we are given numerous sites inside an area and we might want to partition the area with regions which are of roughly same area but each region corresponds to just one site.

There are different algorithms to construct Voronoi diagram with n sites as input. A brute-force algorithm constructs one region at a time. Since each region is the intersection of n-1 half planes, it takes $O(n\log n)$ time per region, thereby resulting in an $O(n^2\log n)$ time algorithm. An optimal algorithm has $O(n\log n)$ lower bound [104]. The planesweep algorithm that we consider here for parallelization is an optimal algorithm.

We are exploiting parallelism in the planesweep algorithm on a per event basis, however, the order of event processing is still sequential. This is because there is interdependence between the static and dynamic events generated by concurrent event processing. We have discovered that there is enough computation in an event itself to warrant performance improvement in a shared memory environment. These computations include intersection of neighboring arcs (w.r.t. an event) that is required to generate new events. This is the first work to identify and report the performance enhancement possible while maintaining the spatial data structures (beachline) on a per-event basis concurrently.

OpenMP and OpenACC are application programming interfaces which enables us to parallelize existing C, C++ or Fortan code by just adding compiler directives (#pragma) to it. The compiler tries to take the directives as hints for potential ways to inject parallelism in the sequential code. Directives based parallelization can be targeted at multicore CPUs, GPUs or a combination of both.

The latest versions of both OpenMP and OpenACC has features and directives that enable both types of parallelization. Directives based programming is promising because it is possible to parallelize a sequential code with minimal refactoring effort compared to directly using threads. However, it may be necessary to re-organize the code to ensure performance improvement in a concurrent code. Adding directives should not affect the correctness of the results produced, although the order in which results are produced might vary due to concurrency.

Criteria for a successful parallelization with directives are as follows:

1. Embedding directives in the code results in speedup on parallel hardware.
2. The added overheads due to parallelization is comparatively low.
3. No substantial changes in the original algorithm and its implementation.

This chapter is a part of our series of work focused on parallelizing existing spatial and computational geometry code using directives. Our prior work was successful in the parallelization of the planesweep version of the segment intersection [105] and polygon intersection [12]. Planesweep algorithms work by processing events in a loop. Unlike Voronoi diagram construction, segment intersection and polygon intersection algorithms require few computational steps to maintain the associated data structures. Among a class of problems that can be solved by planesweep paradigm, we found that sequential Voronoi diagram construction algorithm is unique in the sense that it can be improved by parallelizing the computations involved in each event (point). In this respect, our work is complimentary to the existing research that focuses on data parallel approaches for the construction algorithm.

In particular, this chapter presents the following contributions:

1. Identification of the inherent difficulty in the parallelization of Voronoi diagrams,

especially Fortune's algorithm.

2. Exploration of potential places to inject parallelism via directives in an implementation of Fortune's algorithm. The computations involved in generating new events dynamically has been effectively parallelized leading to around two-fold speedup for a variety of datasets.

3. OpenMP based parallelization of C++ version of the implementation. The skeleton for the sequential C++ code used was inspired by the work of Matt Brubeck [106].

This chapter explores the concurrency available in processing each event in Voronoi diagram construction and uses directives to make an existing implementation of Voronoi computation faster with minimal efforts using compiler directives. This chapter also explores the performance in different scenarios and tries to provide a framework for future work in similar problems. In the rest of the sections below, we discuss the related work and our new algorithm with OpenMP directives. Finally, we provide quantitative results to validate our algorithmic improvement and conclude with a discussion on our future work.

3.2 Related Work

To the best of our knowledge, this chapter is the first in exploring a completely directives enabled parallelization for Voronoi computation. However, significantly speeding up the sequential Voronoi computation has remained a long standing challenge since Fortune came up with the planesweep approach that reduced the complexity of Voronoi computation to $O(n\log n)$. Delaunay triangulation has a dual relationship with Voronoi diagrams. The Delaunay triangulation for a set of discrete points is the connected graph of all the points in such way that no points lie inside the triangles formed by joining the points. A

Voronoi diagram is basically the Delaunay triangulation of the vertices of the resultant Voronoi graph.

Biniaz and Dastghaibyfard compares different sweep line approaches like the Fortune's sweep-line algorithm, Zalik's sweep-line algorithm, and a sweep-circle algorithm proposed by Adam, Kauffmann, Schmitt and Spehner [19]. It tests the implementations of these algorithms on a number of uniformly and none-uniformly distributed sites. Their paper successfully shows that a well written implementation of the sweep-circle method can provide significant reduction in runtime. They have shown significant time reduction with the Zalik's sweep-line algorithm and the sweep-circle method even beats that although not by much. They have used heuristics like number of tests per site to reduce computation time and experimented with different choices of data structures like hashed table or linked list and not been able to show that any one is better than the other.

Wong and Muller presents an even more efficient implementation of the Fortune's algorithm [20]. Effort has been spent on tuning the code and paying attention to hotspots that slow down the implementation. Since no particular routine that dominates the running time was identified, pre-allocating or reusing data structures and maintaining free lists are used to improve storage management. The performance of Fortune's scheme is sensitive to the bucket size. Their paper argues that the efficiency of these data structures is comparable to Fortune's scheme. Abstract data types that are independent of the algorithm is used and the added benefit of allowing performance improvements to the algorithm by simply changing the underlying data structure is stated.

Work done by Bollig explores Voronoi computations in the GPU using a flooding algorithm along with Lloyd's method [21]. A custom tile-based algorithm to evaluate regional mass centroids was further explored. But with increasing

number of seeds, the approaches tried in this work starts to loose efficiency. While it does introduce a new implementation by combining some existing concepts, it doesn't conclude with a scalable technique for the GPU.

Majdandzic et al. claims to presents a parallel algorithm and its GPU-based implementation to calculate a discrete approximation to the Voronoi diagram [22]. This work however focuses on computing Voronoi in the raster where it matches each point on the surface to a particular color-coded seed. When compared to the sequential run of this approach, the GPU-based implementation does give us good speedup but the sequential version of this code is not the best possible sequential method to do this kind of computation. So, for a raster system, this work does look promising but results from this type of implementation can not be further utilized in doing more complex non-raster, computational geometry analyses.

Yuan et al. explores the problem of using the GPU to compute the generalized Voronoi diagram for higher order sites, such as line segments and curves using the jump flooding algorithm and their improvements upon it [23]. This work also explores the raster methods for computing Voronoi diagram. However, rather than just parallelizing the brute force method usually used by raster approaches, they use the idea of separation of site identifier and textures. This way the work focuses on increasing accuracy and reducing memory consumption. However the efficiency of the flooding stage is affected by the extra texture fetching operations needed for accessing the information of a site from its identifier. Their paper claims that its experimental results show their method is much more accurate than that of the original jump flooding algorithm on a GPU and reduces the memory requirement by around 83%.

The work done by Rong et al. explores a GPU-assisted computation of centroidal Voronoi tessellation using Lloyd's algorithm [24]. Centroidal Voronoi

tessellation are special types of Voronoi diagram where each Voronoi cell is also its mean. Their paper proposes a new technique for computing Voronoi diagrams on surfaces and a novel way of using vertex programs to perform the regional reduction over Voronoi cells. The idea is then extended to computing centroidal Voronoi tessellation on surfaces with GPUs. The work is only GPU-assisted because the computation of centroidal Voronoi tessellar energy values and gradients, which is the most time-consuming part, is performed on the GPU and then these values are read back to the CPU for computing the new sites which down the overall computation. Pending complete migration to the GPU, the maximum possible speedup still suffers. Furthermore, the work is still a discrete approximation and still has some inaccuracies.

Tsidaev explores the use of Green-Sibson Voronoi tessellation method in the parallelization technique for Natural Neighbor interpolation algorithm [25]. The idea behind Natural Neighbor interpolation algorithm idea is to calculate Voronoi diagram for all initial points, and then to add each interpolated point into the tessellation with sequential diagram recalculation i.e. natural neighbor algorithm is basically the algorithm to insert an additional point into existing Voronoi diagram. The test example shown in their paper claims that even complex algorithms that cannot be vectorized well can still be efficiently parallelized and furthermore GPU computations could be used as a much cheaper alternative.

Nievergelt and Preparata presents two plane sweep methods for merging geometric figures [26]. This type of work can be used in combining two Voronoi diagrams, especially if the target region was divided to calculate Voronoi diagrams separately and needs merging later. The two algorithms are for cases when the merging figures are convex or non-convex respectively. Voronoi cells are convex in nature but the boundary of collection of Voronoi cells can also be non-convex.

Theoretical parallel algorithms for Voronoi diagram construction have been designed on mesh, hypercube, PRAM models of computation [27]. Parallelizing Voronoi diagram may need experimentation with other approaches from computational geometry that have not been tried yet and if there are parallel versions of these approaches, then we will be able to easily use them in our parallelization.

3.3 Plane Sweep

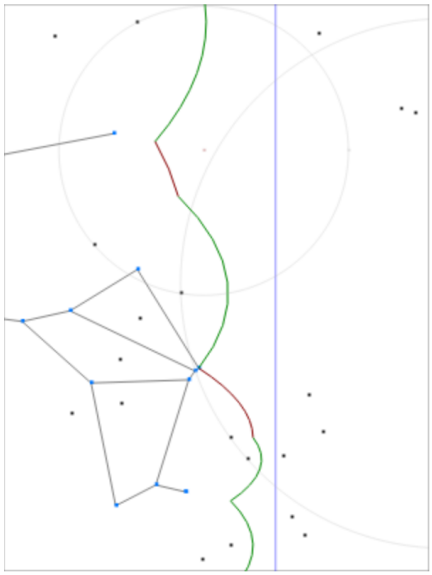

Figure 3.2: Plane Sweep Voronoi Calculation
[The figure show an in-progress computation of the Voronoi diagram. The dots in the figure are the sites. The arcs are collectively the currently active beachline line at the sweepline position. The sweepline is the vertical line across the figure in the middle.]

Plane sweep is a common approach in computational geometry where the target geometric space is swept by an imaginary line and as the sweepline progresses computational geometric solutions at and below the sweepline are calculated. Plane sweep is extremely efficient in solving varieties of problems in computational geometry especially by reducing the complexity of the problem down to O(nlogn) scale. Plane sweep is used in basic operations like calculating intersections of lines

and polygons to complex operations like computing Voronoi diagrams. There are different sweep approaches for solving Voronoi diagrams [19] but the most widely used and simple to implement is Fortune's Algorithm [107]. Fortune's algorithm however is more complicated than the usual plane sweep approaches because the solutions below the sweepline can still be influenced by points above the sweepline. Fortune's algorithm tackles this problem by the ingenious use of beachline. This property that solution could be affected by future events however adds more complexity to the algorithm which makes parallelization of this algorithm even more difficult.

Definition 2. *For a horizontal left-right sweep, the beachline B at a particular sweepline position is the trace of the maximum bounds (max_x) of all the active arcs at that sweepline position.*

3.4 Fortune's Algorithm

Fortune's algorithm is a planesweep algorithm for computing Voronoi Diagram in $O(nlogn)$ time with $O(n)$ space [107]. Fortune presented a transformation that could be used to compute Voronoi diagrams with a sweepline technique. According to Fortune, "Rather than compute the Voronoi diagram, we compute a geometric transformation of it. The transformed Voronoi diagram has the property that the lowest point of the transformed Voronoi region of a site appears at the site itself. Thus the sweepline algorithm need consider the Voronoi region of a site only when the site has been intersected by the sweepline" [107]. Algorithm 3.1 shows the overall structure of the algorithm proposed by Fortune as explained by Berg et al. in their book [28].

In Algorithm 3.1, in HandleSiteEvent at line 6 new arcs get created and in line 8, HandleCircleEvents arcs gets removed from the beachline. For each event, the algorithm loooks at three concecutive arcs for convergence, new events may get

Algorithm 3.1 VoronoiDiagram(P)

```
1: P ← {p_1, p_2, ..., p_n} sites as points
2: Initialize the event queue E with all site events, initialize an beachline B and an
   empty edge list O.
3: while E is not empty do
4:     Remove the event with largest x-coordinate from E
5:     if the event is a site event, occurring at site p_i then
6:         HandleSiteEvent(p_i)
7:     else
8:         HandleCircleEvent(r)
           where r is the arc from the beachline that will
           disappear
9:     end if
10: end while
11: The arcs still present in B correspond to the half-infinite edges of the Voronoi
    diagram. Compute a bounding box that contains all vertices of the Voronoi
    diagram in its interior, and attach the half-infinite edges to the bounding box by
    updating the edge list appropriately.
```

created by the events processing method. Output of Algorithm 3.1 will be O, the list of all the edges of the computed Voronoi diagram. We have identified the computations involved for parallelizaion in HandleCircleEvents as described in subsection 3.4.1.

Figure 3.3, adopted from [108], shows the construction of Voronoi diagram as the sweepline progresses. This figure shows a top-down vertical sweep rather than the horizontal left-right sweep mentioned primarily in this chapter and the sweep direction is inter-changeable. In the figure, the grey points are the unprocessed site points, red points are the processed site points, green points are circle events and blue ponints are the Voronoi vertices. The purple horizontal line is the sweepline, the light grey curves are the arc for each processed site points and the green trace of light grey curves shows the beachline at the active sweepline position. At each of the sweepline position, the beachline is updated and Voronoi vertices are identified via circle events check. Then before moving on to the next sweepline position, Voronoi edges upto that point are constructed. Figure 3.3(e) is an example where a

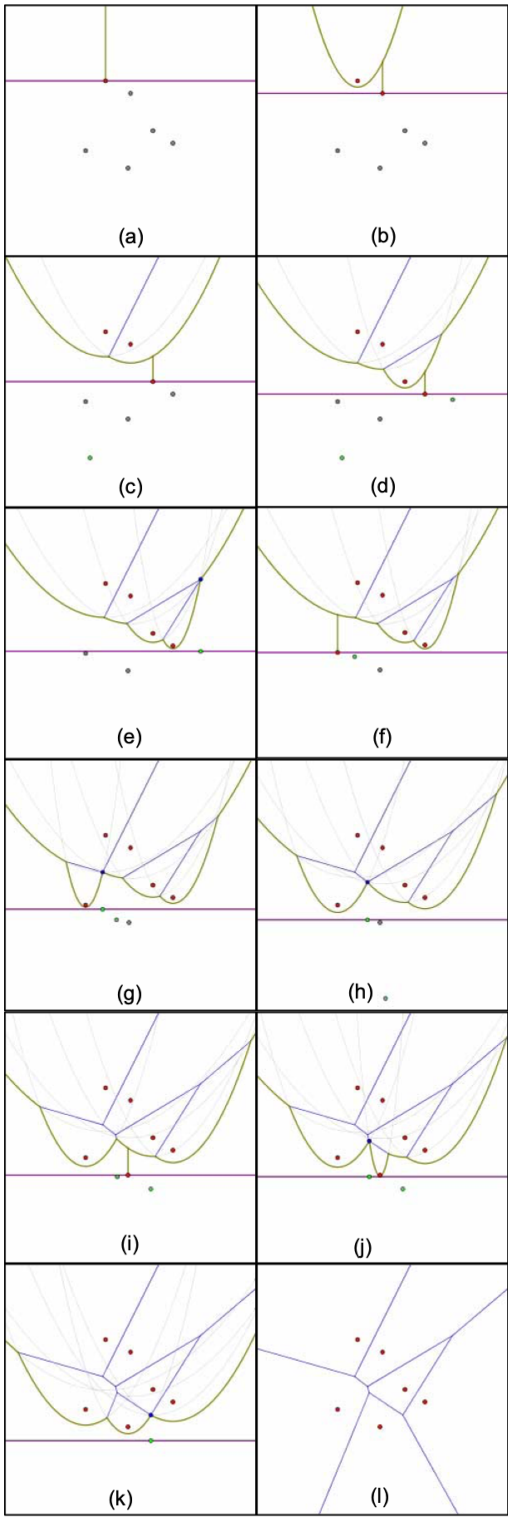

Figure 3.3: Fortune's Algorithm Progression
This figure is best viewed in color as each color corresponds to a different phase in the progression.

circle event check results in a Voronoi vertex. Figure 3.4, also from [108], further illustrates this. At the sweepline, correspoding to the lowest red site, we need to find the corresponding arc vertically above it. The computations involved in doing so requires searching the active arcs in the beachline. This search process per event is one of the target of parallelization that we have described in subsection 3.4.1.

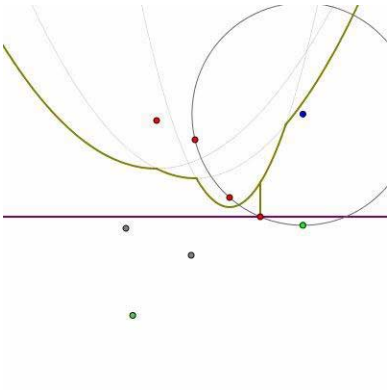

Figure 3.4: Circle Event during Fortune's progression
Expansion of Figure 3.3(e) for better viewing

Algorithm 3.2 is a simplified algorithmic description of the implementation of Fortune's Algorithm. The algorithm and data structures mentioned follows their C-type implementation style loosely. The focus of the description here is to show the flow of the algorithm so that the possibilities and limitations to a directive based approach can be explored. This algorithmic description here is necessary to understand the flow of execution and interdependencies among the variables that are key to any directive-based parallelization.

Following are the data structures used in Fortune's algorithm. Here var is a placeholder for any numeric data type like int, float, double, etc.

```
struct point {
    var x,y;
}
```

Listing 3.1: Data Structure for Point

Algorithm 3.2 Fortune's Algorithm (Horizontal Sweep)

```
1: P ← load all points
2: Initialize a bounding box with offset
3: Initialize beachline B
       // B is of type arc
4: Initialize output O
       // O is a collection of segments
5: Initialize events priority queue
       // event with minimum x-coordinate is at the top
6: Sort P in ascending order by x-coordinate
7: for each p in P do
8:     while (events.top.x ¡= p.x) do
9:         ProcessEvent(events.deque())
10:    end while
11:    ProcessPoint(p)
12: end for
13: ProcessRemainingEvents()
14: FinishEdges()
```

```
struct event {
    var x;
    point p;
    arc *a;
}
[Here x is the maximum x-location a circle event can affect
   and it introduces a event processing there.
So, x = p.x + radiusOfTheCircle]
```

Listing 3.2: Data Structure for Event

```
struct arc {
    point p;
    arc *prev, *next;
    event *e;
    seg *s0, *s1;
```

```
}
```

Listing 3.3: Data Structure for Arc

```
struct seg {
    point start, end;
    bool done;
}
```

Listing 3.4: Data Structure for Segment

Algorithm 3.3 ProcessEvent(event e)

```
1: Input event e
2: if (e.valid) then
3:     Begin a new Segment s at e.x
4:     Remove e.a from beachline B
5:     Complete segments e.a.s0 and e.a.s1
6:     // Check circle events
           CheckCircleEvent(e.a.prev, e.x)
           CheckCircleEvent(e.a.next, e.x)
7: end if
```

Algorithm 3.4 ProcessPoint(point p)

```
1: Input point p
2: for arc i in beachline B do
3:     if intersects(p,i) then
4:         Add new arc at p.x to beachline B
5:         Connect new arc to prev and next segments of i
6:         // Check circle events
               CheckCircleEvent(i, p.x)
               CheckCircleEvent(i.prev, p.x)
               CheckCircleEvent(i.next, p.x)
7:         return
8:     end if
9: end for
10: arc i ← last arc in B
11: Insert segment between p and i
```

Algorithm 3.5 CheckCircleEvent(arc i, var x)

```
1: Input arc i and var x
2: Check new circle event for i at x
3: if Circle Event Found then
4:     Initialize a new event
5: end if
```

Algorithm 3.6 ProcessRemainingEvents()

```
1: while (events not empty) do
2:     ProcessEvent(events.top)
3: end while
```

3.4.1 Unpacking Fortune's Algorithm

We start by trying to find opportunities in the algorithm where compiler directives can be inserted for parallelization. The most obvious choice would be to parallelize the loops. Loop parallelization using directives is the easiest way to parallelize and usually has very less overheads. Furthermore, internal loops inside nested loops can also be parallelized giving us further speedup.

In Algorithm 3.2, the for-loop in line 7 can not be parallelized because there is a event processing inside there, and event processing changes the list of active events and also removes an arc from the beachline. Also, event processing checks if there are circle events or not, and if there are circle events then again new events are added. So we would like to see if at least the while-loop in line 8 can be parallelized. This is again not possible because of the same reasons as above. Algorithm 3.4 and Algorithm 3.3 can not be rum concurrently.

In Fortune's algorithm, the site events and the circle events have

Algorithm 3.7 FinishEdges()

```
1: for (each arc i in beachline B) do
2:     Complete any incomplete segments
3: end for
```

interdependence, as such these two class of events can not be separately parallelized. In the Algorithm 3.2, we can not do a loop fission by removing the processing of points outside the for-loop to run it later separately after all the events are processed. If this separation were possible then we could try the parallelization of the separated loop. However, since all the events processing before the points processing are dependent on the list of active events and the beachline and points processing updates them, points processing has to be done at that point. Since points processing is dependent on the list of active events and the beachline; neither moving it to a separate loop or processing it in parallel is viable. Due to the sorted nature of points, each x-coordinate of a point corresponds to the location of a sweepline. Since processing points in parallel turns out to be not viable, this would mean that processing the sweeplines in parallel is not viable. In some cases, the order of processing of certain parts of the code can be altered but it most certainly is not the case here. However, we may still be able to parallelize processing within a sweepline.

Next we look at Algorithm 3.3. Since entirety of its execution is based on a conditional, we need to determine the possibility of parallelizing this portion if it gets executed. Here, line 4 is dependent on line 3 because we need the segment s to remove $e.a$ from beachline B. However, excluding this, the two operations in line 5 and the two operations in line 6 can be parallelized to run concurrently. Completing the two segments in line 5 does not affect any other operations that could happen here concurrently. However the two circle events check in line 6 can lead to new events being added, but since these events are just added and not used elsewhere, we can put adding events part of the code inside critical sections and still parallelize line 6. So, in overall we can have five sections that run in parallel here - one section would comprise of lines 3 and 4, another two sections would comprise of completing each segment in line 5 and the other two sections would comprise of the two circle

events checks in line 6.

Next we look at Algorithm 3.4. This portion is even more complicated to parallelize because it has loops, conditionals inside loop and early exits inside those conditionals. An event is rendered invalid if the arc associated with that event is no longer in the beachline.

The outermost loop is searching for an arc corresponding to the new event. This is done by performing an exhaustive search looking for a single instance for which the search criterion is fulfilled. Then a series of operations is performed on the resultant instance if it was found. If a resultant instance was found then not only the loop is returned but the whole procedure is exited. We can start by separating the search and the execution of the result of the search. So, we parallelize the loop in step 2 to find an arc i which satisfies the if-condition and remove the execution part below. One problem here is that if such an arc is found by sequential iteration early on, parallelizing it might just give us unessential overhead. To remedy this, we will convert this search loop into a chunked iterative exhaustive search loop by providing hints to the compiler that there might be a loop cancellation before each chunked iteration. This transformation makes it suitable for utilizing OpenMP parallel loop cancellation feature as shown line 5 and line 6 of Algorithm 3.8.

Concurrent Processing of Circle Events:

Another problem with Algorithm 3.4 is that, a sequential search would have terminated after finding the first instance for which the search criteria would have been satisfied but during a concurrent chunked iteration, there might be multiple instances for which the search criteria has been satisfied. For correctness with regards to the sequential code, we can use a minimum reduction to make certain that the first instance is reported. At this point we will either have an arc i that satisfies the conditional or not and the loop will be exited but the procedure will not

have been terminated. We can put this conditional of whether an arc i has been found in an if-statement with its else-part as lines 10-11. If an arc i has been found then we can execute the lines 4-6 with i and if not then we execute lines 10-11. This removes any early procedure terminating conditions from Algorithm 3.4. Then lines 4-6 that has been moved out of the loop and put inside this new conditional statement can now be explored for further parallelism. Lines 4-5 need to be executed sequentially because line 5 is dependent on the arc created in line 4. However, as shown in Algorithm 3.8, the three parts of line 6 can be parallelized to run concurrently even along with lines 4-5. Again, here the circle events check can lead to new events being added, but since these events are just added and not used elsewhere, we can put adding events part of the code inside critical sections and still parallelize. However, we will not be able to parallelize lines 10-11 of Algorithm 3.4 because its execution needs to be sequential. So, in this portion we are able to parallelize the search phase and lines 4-6 after they have been moved outside. As shown by Algorithm 3.8, Lines 4-6 from Algorithm 3.4 will have four sections - first section would comprise of lines 4-5 and the other three sections would comprise each of the three parts of line 6.

Next we look at Algorithm 3.5 which performs circle event check, the checking of the circular event done at line 2 is a fairly sequentially ordered portion. The purpose of the second argument of Algorithm 3.5 is to detect false alarms. Here, due to the changes done in the other calling algorithms, we need to make sure that the new events creation and appending are done inside critical sections. Algorithm 3.6 has a loop but this also loop needs to be processed in sequential because each call for event processing changes the events list. Finally, Algorithm 3.7 has a loop that can be easily parallelized because the segments processed in the loop here has no side effect elsewhere.

Algorithm 3.8 ProcessPoint(point p) with directives

```
1: Input point p, initialize bool doesIntersect = False

   #pragma omp parallel for num_threads(threadCount)
2: for arc i in beachline B do
       j ← index of arc i in beachline B
3:     if intersects(p,i) then
4:         ind = j
5:         doesIntersect = True
           #pragma omp cancel for
6:     end if
       #pragma omp cancellation point for
7: end for

8: if (doesIntersect == True) then
9:     arc i ← B[ind]
       #pragma omp parallel sections
       {
           #pragma omp section
           {
10:        Add new arc at p.x to beachline B
11:        Connect new arc to prev and next segments of i
           }
           #pragma omp section
12:        CheckCircleEvent(i, p.x)
           #pragma omp section
13:        CheckCircleEvent(i.prev, p.x)
           #pragma omp section
14:        CheckCircleEvent(i.next, p.x)
       }
15: else
16:    arc i ← last arc in B
17:    Insert segment between p and i
18: end if
```

3.5 Results

Random points were generated using a uniform random probability distribution. The generated points were controlled via rejection to avoid degenerate cases. Degenerate cases include two points that are at the same x-coordinate for a horizontal plane sweep and collection of points that are extremely clustered and not well distributed in the target region.

An OpenMP implementation of code was created using the analysis in section 3.4.1 and executed on data with varying number of sites. The machine used to run the OpenMP codes has the Intel Xeon E5 CPU v4 E5-2695 with 18 cores and 45MB cache and base frequency of 2.10GHz. Given there are four parallel sections in each parallelized part of our code, it would only be fitting to use four threads. To avoid threads re-creation overheads, it would be advisable to create threads beforehand and reuse them.

Table 3.1: Timings of running the code in sequential and with OpenMP

Sites	**Sequential**	**OpenMP**	**SpeedUp**
2k	0.456s	0.165s	2.761
4k	0.758s	0.419s	1.809
8k	2.06s	0.995s	2.070
16k	6.496s	2.748s	2.364
32k	13.748s	5.162s	2.663
64k	38.847s	18.029s	2.155
128k	84.396s	39.305s	2.147

Figure3.5 shows the execution time for different number of site events. Even with the overhead of parallelization, the OpenMP version beats its sequential counterpart. Our solution can be combined with data parallel versions of the algorithm. We can see from Table 3.1 and Figure 3.6, we get almost above 2x speedup. The distribution of points affects the runtime of our algorithm and we have observed that having some types of distribution of points improves the

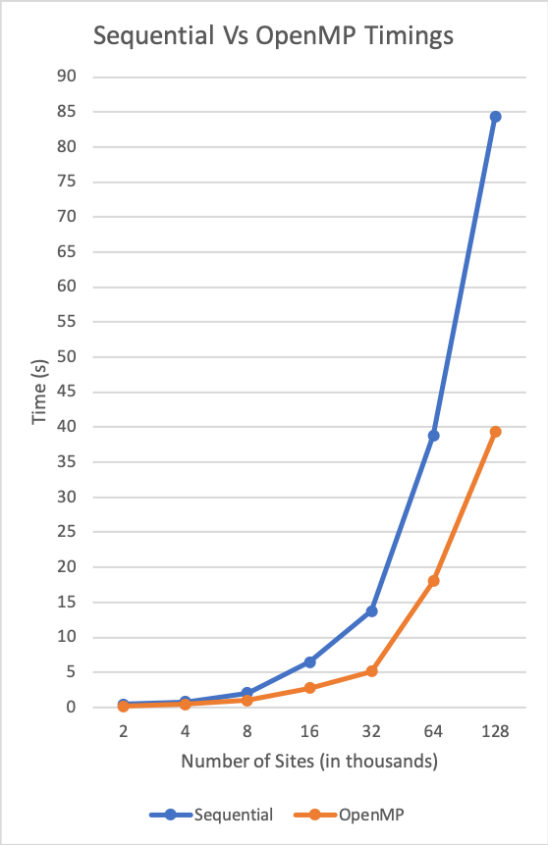

Figure 3.5: Sequential vs OpenMP timings

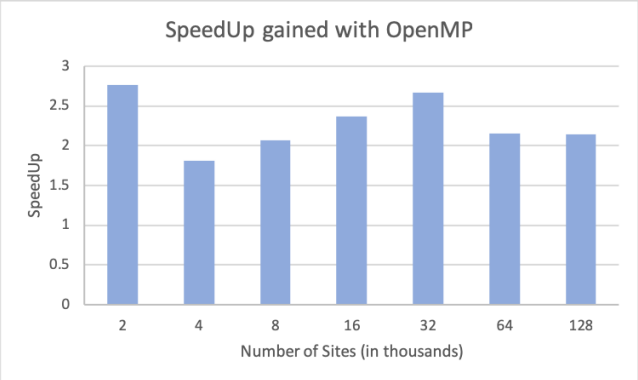

Figure 3.6: SpeedUp gained with OpenMP (4 threads)

performance of our algorithm. As we can see in figure 3.6, the speedup is varying for different number of sites. This is because the time it takes to search for an arc corresponding to the event being processed is variable. In Algorithm 3.8, there are two code blocks which have been parallelized using OpenMP. There is a sequential

dependency between block 1 (lines 2-7) and block 2 (lines 8-18). Eventhough the for-loop is highly parallelizable, the second block with OpenMP sections can only use few threads. In the worst case scenario, the execution time for block 1 depends on the number of active arcs there are in the beachline but on average case, the intersection test can terminate much earlier. We have found that beyond four threads there is a degradation in efficiency.

3.6 Future Direction

The performance improvement gained by simply parallelizing Fortune's algorithm with compiler directives seem to be very modest. So, without tweaking or refactoring the algorithm to some extent we might not be able to gain any more speedup. For doing so in ways that can be easily achieved with directives based programming, following are some of the methods that we have observed to have some potential or need experimentation to gauge them.

3.6.1 Backtracking

The sweepline progresses in an unidirectional manner and at each sweep-position, the sweepline computes a corresponding beachline. The beachline is a collection of active arcs when the sweepline is at a particular position and can change at each sweep-position as points or events gets processed. Then at each sweepline, using the sweep-position and the beachline, circle events are checked which results in Voronoi vertices. This computation of finding all the Voronoi vertices can be done independently for each Voronoi site if the graph building phase is to be separated from the Voronoi vertices finding phase. A final sweep to connect all the vertices will then be required to complete the diagram. For this, in each sweep position instead of moving in the direction of the original sweep, we will move in the reverse direction, creating a beachline at the sweepline position but in the

same orientation as the beachline would have been if we were moving forward. We would continue traversing backwards until the traversing backwards does not change the beachline any further because no more points that could contribute to the beachline at that sweep-position exists. In this way, we can have the beachline for each sweepline, and each of these sweepline and beachline pair can be processed in parallel to find circle events that will give us the Voronoi vertices. Because of nearby sweeplines having almost the same arcs in their beachline, this process of backtracking from each sweep-position would increase the redundancy of the work done while finding the beachline at each sweepline. However, since we can parallelize processing of each sweepline and each sweepline and beachline pair can be found in parallel, we will be able to parallelize the entire process to gain much better parallelism. Backtracking will overall have three phases of which the first two are parallelizable - one to find the corresponding beachline of each sweepline, next to process each sweepline and last to build the final Voronoi diagram by sweeping through connecting all the Voronoi vertices.

3.6.2 Transformation

Computation of Voronoi diagram is dependent on the order of distance between points in the target region from the sites. Voronoi edges are created in such a way that they can group all points in the target region so that they can have a common closest site. Any transformation to the target area would also change the Voronoi diagram if the transformation changes the distribution of distance between the points in the target to become further or closer in the transformed region. So, any transformation that we do to the target region for simplifying or speeding-up Voronoi computation, in terms of mutability should have these two general features:

1. The transformation should not change the order of distance distribution between points and sites.

2. The transformation should be back-transformable in such a way that the Voronoi diagram can be retained.

Sometimes the best possible transformation that we might obtain may not be suitable for any distribution of points in the target region. In such cases, we might have transformations that are not universal but only applicable to certain distribution of points and sites. As long as the transformation criteria are satisfied for the given points distribution, we should be fine. For example, it has been observed that having an oblique distribution of points reduces the time spent on computation with our algorithm, so, any transformation that transforms our target region to be oblique and satisfies the criteria could be used for pre-processing and a corresponding back-transformation could be used after the Voronoi diagram was computed in the transformed space to get a Voronoi diagram in the original space.

3.6.3 Gridding

The simplest way to process the target region in parallel would be by splitting it and processing each split in parallel. Making grids would be the easiest way to split the target region. Grids can have different designs but the most common one would be uniform repeating equal rectangular boxes that span the whole area. Also, based on our previous observation that oblique target spaces gives us better performance, the gridding can be done in a manner so that we get oblique divisions of the target space. For approaches like gridding, one of the major costs is the combine step of joining the Voronoi diagrams from each grid together. Having fewer grids would be computationally more efficient because it would mean lesser join costs. With our parallelization approach instead of making more grids to process in parallel, we can make bigger grids that gets processed in parallel. Hence by making bigger grid sizes and thus reducing the number of grids, we can reduce the cost of joining Voronoi diagrams by having less number of grids to join.

Voronoi diagrams have a very geometrically local solution i.e. adding or removing an arbitrary site from the target space only distorts the Voronoi diagram around that site. If instead of a contiguous geometric grid, the points are selected at random, then joining the result from would be an extremely complex process. Thankfully the distortion is not propagated beyond the immediate neighbors of the added/removed site. This means that dividing the points by making good geometric grids is an extremely viable approach. However, joining the grids to obtain a final solution will still have overheads. Also, in some cases deliberately increasing the work, but having the ability to do the overall work in parallel might not be such a bad idea. In such cases, we can have different patterns for gridding which has overlaps with its neighbors. This overlap will give us continuity in edges between the grid edges by redundantly computing Voronoi edges for the overlap. But if this overlap is big enough to reduce the need for a merge operation by replacing it with just a filter operation then it might be worth it. In this way we might even get good speedup.

3.6.4 Sorting

If each of the sites for which the Voronoi partition would occur were somehow closely indexed in a contiguous list, then we could easily just take a site and sites from its neighboring indices to calculate Voronoi cell of each site. In such cases, we could just traverse the list to calculate Voronoi cells for each site. The entire process could be parallelized by taking a site and its sites at its neighboring indices and concurrently calculating Voronoi cells for each of them. Since, the edges will be redundant among neighboring Voronoi cells, instead of running a complicated merge, we can just discard one of the repeating edge. This would be possible if we are able to sort the points in such a way that all points in the sorted list have points that are around it in the target region at nearby indexed positions in the list.

3.6.5 Heuristics

Heuristics in computing are an approach to problem solving using shortcuts-like methods to reduce computation time or nudge us in the direction of a solution. Heuristics still have a possibility of leading in an entirely wrong direction or not reaching an actual or optimal solution. But in most cases, if the heuristics are not completely stochastic, we should be able to gain some benefits. An example could be the use of heuristics in approximating boundaries for the Voronoi diagram and re-sweeping the target space iteratively to reach a solution. This would be a viable approach if we have good enough heuristics where approximating and re-sweeping is less computationally expensive than running the entire algorithm. Here parallelization can be explored in the approximation or re-sweeping phase. Similarly another example of a heuristic can be approximating the beachline in each scan-position. Heuristics could also help in determining what types of transformation to use, what pattern of gridding to consider or what number of neighboring points to take from a sorted list.

3.6.6 Machine Learning

Machine learning and neural networks have proven to be an efficient method to solve a large variety of problems and are also easily parallelizable. So, it might be possible to train a machine learning algorithm or a neural network to compute Voronoi diagrams. This idea needs further exploration and there is a possibility that the solutions will only be an approximate one, which is suitable for applications where precision might not be the primary concern.

3.7 Conclusion

Our experiments and exploration in directives-based parallelization of Fotune's algorithm has yielded a shared memory implementation that gives around

2x speedup compared to the sequential version. Considering the amount of work required for injecting directives, this amount of parallelization is modest. A four threaded parallelization is however extremely useful for applications that run on personal devices that are mostly quad cores or on cloud instances where the most common instance of compute nodes usually has four cores. The exploration into each step of the implementation and parallelization attempts using directives should open up some of the challenges in parallelizing computational geometry paradigms like the planesweep and expose the most challenging areas for directive-based parallelization.

CHAPTER 4: ACCELERATION OF SEGMENT TREE GEOMETRIC DATA STRUCTURE

Segment tree is a static binary tree data structure used for storing and querying line intervals (segments). This data structure is widely used in computational geometry and Geographic Information Systems (GIS). In this chapter, we have successfully parallelized segment tree construction and query operations using a completely compiler directive-based approach with minimal changes to the sequential code. Furthermore, we have used the segment tree data structure in two computational geometry operations - 1) reporting rectangle intersections (overlaps) and 2) point-in-MBR tests. MBR stands for Minimum Bounding Rectangle.

Using OpenMP, we have explored loop-based and task-based parallelization for segment tree construction algorithm. Our loop-based formulation and its parallel implementation outperforms the task-based implementation. Using OpenMP on the Intel Xeon E5 CPU, which is a 18 core (36 threads) CPU, we have achieved upto 29x speedup for tree construction and 23x for batch querying using 32 threads. To minimize data movement between cache and main memory for segment tree construction, we present a cache-efficient segment tree construction method which yielded upto 1.2 to 1.4x speedup compared to standard tree construction. Using OpenACC on Nvidia TITAN V, we have achieved speedup upto 100x for the tree construction phase and speedup upto 62x for the batch query phase.

4.1 Introduction

Segment tree is a tree data structure which is used for efficient storage and retrieval of interval or line segment information [28]. The stored segments or intervals can be queried for a given point. It is usually a structure that is not

modified once created, hence a static data structure.

Segment tree is basically a binary search tree, where each of the leaf nodes represents an elementary interval obtained from the list of segments/edges. The abscissas of the points on all the edges are taken and sorted to create elementary intervals for a given list of edges. Each parent node is a union of its children nodes. In this way the root node spans the entire finite coordinate space for the given list of edges. Each node stores a list of zero or more input edges, which is called a cover list. This list is used to construct the output of the point query.

Since Segment tree is a static structure, all the segments or intervals are required a priori to start initializing and building the tree structure. Once, the Segment tree is built, we can use it to run multiple queries in a batch or have streaming queries. The static nature of the tree allows high degree of concurrency in query operations without having to worry about the read/write lock scenarios.

Unable to update the segment tree once it is build is a limitation, but it also provides us with the opportunity to query the tree concurrently without having to worry about the read/write lock scenario. Furthermore, we can have different types of read based operations like queries running either in parallel or in a streaming fashion always giving the same results irrespective of the timing of the operation.

Segment tree and its variants have been studied in literature theoretically [9, 27, 28, 29] and experimentally [32, 33, 34, 35, 36, 37] in computational geometry. In theoretical parallel algorithms literature, this data structure allows development of optimal parallel plane sweep algorithms which is a fundamental computational geometry paradigm [9, 10, 109].

We present a novel cache-efficient method of segment tree construction that exploits temporal locality to reuse the data already loaded into the cache. This method is different from Van Emde Boas layout optimization which is applicable in

cache-oblivious query operations for our problem, but not in cache-efficient data structure construction [41, 42, 43]. Modification in tree storage layout exploits spatial locality inherent in tree-search algorithms but requires change in parent-child access method in search algorithms. Our method keeps the data organization same as standard segment tree construction algorithm. Therefore, it does not need any change in standard query algorithms. Optimizing data structure construction is beneficial where the construction time is non-negligible compared to query execution time.

This chapter presents the first GPU-based parallelization of Segment tree. Each node of a Segment tree contains variable number of elements. This irregularity is data dependent and challenging to optimize on SIMD/SIMT architectures. Although there has been successful parallelization of some tree-based data structures on GPU, Segment tree construction presents unique challenges and design opportunities that are not encountered in other data structures like B-tree, Range tree and R-tree (Rectangle tree) [44, 46, 47].

In this chapter, we present two computational geometry applications that utilize our parallel segment tree to gain speedup. First application is in rectangle intersection problem where two collections of rectangles are the inputs and the output consists of all the overlapping pairs of rectangles. This application is used in the filter phase of polygon overlay and spatial join in Geographic Information System (GIS) and spatial database respectively. Second application is a parallel version of Point-in-Rectangle algorithm. This algorithm tests whether a point of a rectangle from one collection is contained in a rectangle from another collection. This is a variant of the standard point-in-polygon test where polygon is approximated by a rectangle (minimum bounding rectangle).

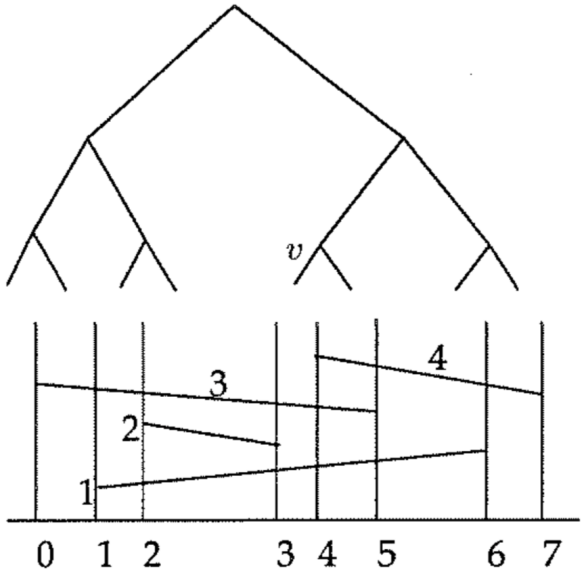

Figure 4.1: Basic Structure of a Segment Tree

4.2 Contributions of this chapter:

1. Parallelization of Segment Tree building and querying on multi-core devices using OpenMP directives
2. Parallelization of Segment Tree building and querying on GPUs using OpenACC directives
3. Cache optimization of Segment Tree build phase
4. Speeding up applications that use Segment Tree: 1) minimum bounding rectangle intersection and 2) point-in-MBR tests. MBR stands for Minimum Bounding Rectangle.

Section 4.3 describes background, motivation and related works about Segment tree and its parallelization. Section 4.4 describes the design and implementation of constructing segment trees on multicore CPUs and GPUs along with their algorithmic complexities and cache efficiencies. It also presents the implementation of queries. Section 4.7 presents different geometric operations using

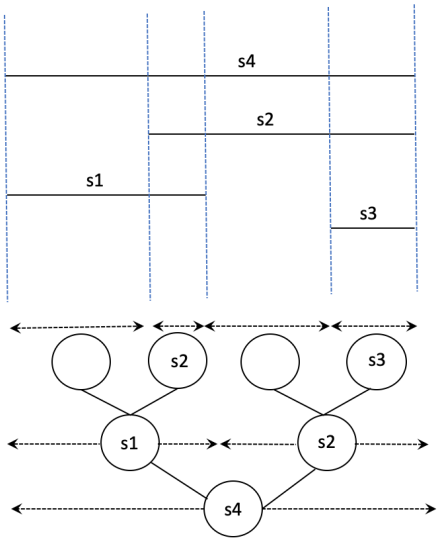

Figure 4.2: Segment tree with four input line segments
Line segments s1, s2, s3 and s4 and their corresponding four elementary intervals are shown in the leaf level as horizontal dotted lines. The interval associated with a non-leaf nodes is the union of the intervals of its two children. The line segments in the cover-list of each node is also shown.

Segment tree. Section 4.8 presents the experimental section with performance results.

4.3 Background and Related Work

Segment tree data structure was introduced by John Louis Bentley in 1977 [28]. One of the important operations on a segment tree is a stabbing query, which takes a query point p as an input to report all the line segments overlapping with an imaginary vertical ray passing through x coordinate of p. An example of a segment tree with four input segments is shown in Figure 4.2.

Cover-list: After building the skeleton of the tree, each node v has an interval associated with it, namely, Interval(v). For instance, in Figure 4.2, elementary intervals at leaf level and intervals made by union of elementary intervals are shown. Next step is to populate cover-list at each node with a subset of input segments that satisfy a condition. The condition is that an input segment

belongs to the cover-list of node v, if it contains Interval(v) and does not contain Interval(parent(v)). An example of contains relationship is as follows: a segment with start point (0,0) and end point (5,0) contains an interval with start point (1,0) and end point (4,0). Moreover, by definition, a segment with the same start and end points as the interval also contains it. Each node stores the segments that span through its interval, but do not span through the interval of its parent [28]. For instance, s1 is not kept at the root node because it does not span through the root's interval. Similarly, s1 is not kept at the leaf level even though it contains the two elementary intervals at the leaf level because it contains the interval of leaf node's parent. This condition makes sure that if a segment is stored in node's cover list, it does not get stored in its left and right subtrees. Therefore, subsequent query going from root to a leaf, does not find duplicate results. An input segment can be present in at most two nodes per level of the tree.

For n line segments as input, the time complexity of building the tree is $O(n \log n)$ [28]. The space complexity of building the tree is $O(n \log n)$. The time complexity of the stabbing query is $O(\log n + k)$ where k is the number of line segments in the output of the query [28]. An important application of segment tree is in parallelizing plane sweep algorithms for computing line segment intersections using PRAM model [9, 10, 29]. External memory segment tree algorithms have been presented in [30, 31].

Parallel and distributed algorithms for segment tree data structure are presented in [32, 33, 34, 35, 36, 37]. Segment tree construction on a hypercube architecture to solve next element search problem (also known as first hit) is presented in [32]. A parallel algorithm using PRAM (Parallel Random Access Machine) model of computation and implementation on connection machine was presented in [34]. Bulk Synchronous Parallel (BSP) model has also been used to

develop and analyze segment tree algorithms [35]. Segment tree was used in parallel PRAM algorithm for polygon clipping in [12].

The influence of caches on the performance of heap data structure was presented earlier by LaMarca and Ladner [38]. A variant of binary heap optimized for virtual memory environments was presented as B-heap [39]. B-heap keeps subtrees in a single page of virtual memory and performs well for large heaps.

In external memory algorithms, a segment tree variant has been designed to minimize data movement by increasing the fanout of the node and recursively splitting the node among its children [30, 40]. Compared to the earlier theoretical work [10, 30, 40], we present practical algorithms that allow parallelism in computational geometry applications as well as minimize data movement between fast memory and slow memory.

A cache-oblivious method known as Van Emde Boas (vEB) layout has been presented earlier to store static binary search trees recursively in a cache-efficient manner to minimizes data movement in a query operation [41, 42, 43]. Conceptually, vEB layout transforms a static binary tree by recursively splitting the tree at the middle level of edges so that the tree nodes get grouped together to minimize data movement in search operations [42]. These methods are not geared towards cache efficiency in the data structure construction phase. In contrast, our cache-efficient method is targeted towards building the data structure only.

OpenMP Tasking was introduced in 2008 as a major addition to OpenMP 3.0. This feature allowed convenience in implementing irregular and recursive algorithms using compiler pragmas supported by OpenMP specification. A general way of using tasking is by creating chunks of work that can be executed concurrently. Then these chunks (tasks) are stored in a queueing system. The assignment of threads to the tasks is dynamic and handled by the runtime system.

Computational geometry algorithms are often expressed as recursive programs [28]. This is the case with Segment tree algorithms as well. Before the introduction of tasking feature in OpenMP, manual transformation from recursive functions to iterative functions was usually done.

GPU has been used for implementing data structures like Btree [44] and computational geometry data structures like KD-tree [45], R-tree [46] and range tree [47]. Here we focus on OpenACC which is a compiler pragma-based approach to do GPU parallelization.

4.4 Design and Implementation

4.5 Building Segment Trees

The input of a segment tree is a set of segments (edges). Since the segment tree is a static structure, we need to first build it to start querying it. Building a segment tree entails:

1. Loading all the edges into memory.
2. Getting all the elementary intervals from the edges
3. Then combining the elementary edges to create a tree structure. Each parent node will be do the union of the intervals associated with its children nodes.
4. Adding each edge to the cover list of the nodes.

```
struct SegTree {
    int  treeSize, treeHeight, numEdges;
    Array<Edge> elementaryEdges;
    Array<Node> treeNode;
}
```

Listing 4.1: Data Structure for SegTree

```
struct Edge {
    Real start;
    Real end;
    int id;
}
```

Listing 4.2: Data Structure for Edge

```
struct Node {
    Edge interval;
    int coverCount;
    Array<int> coverList;
    int count;
}
```

Listing 4.3: Data Structure for Node

The data structure listings show the basic C/C++ like structure for Segment tree (*SegTree*), Edge and tree node. Variable *treeSize* refers to the total number of nodes in the Segment tree. Variable *treeHeight* refers to the height of the tree i.e. the number of steps between the head to a leaf node. Variable *numEdges* stores the number of input edges. The variables *start* and *end* in the Edge data structure represent the start and end points. The variable *id* is given to each edge so that it can have a unique identifier which can be used in referencing query results. Real is a typedef used to denote real decimal numbers like floating point of the precision required by the user or use case.

Algorithm 4.1 shows the overall steps required to build the Segment tree. Algorithms 4.3 and 4.4 further explain each step required to complete the steps in Algorithm 4.1. Algorithm 4.1 takes all the input segment data and builds a tree

Algorithm 4.1 Segment Tree

```
1: E ← {e_1, e_2, ..., e_n} edges as line segments
2: Make a sorted vector of elementary intervals from E
3: Initialize a SegTree structure with the elementary intervals
4: Build the skeleton for segTree
5: for edge e in E do
6:     insert e into the segTree
7: end for
```

ready for query.

Algorithm 4.2 Elementary Intervals(Array<Egdes> edges)

```
1: Initialize an empty set of points: SP
2: for edge e in edges do
3:     for point p in e do
4:         if p not in SP then
5:             add p to SP
6:         end if
7:     end for
8: end for
9: Sort all unique points p in SP
```

Algorithm 4.2 takes all the segment edges from the input data set and creates a set of elementary edges from it.

First we generate all the elementary intervals from segment or interval dataset. This requires finding unique points and sorting those points. Elementary intervals fill up the bottom most leaf layer in the tree. From there we start building up the tree. Algorithm 4.3 takes the elementary intervals as input and assigns the parent of children nodes to be the union of the children nodes. It does this from the leaf level up to the root of the tree. In this way, the root node will span the complete range of its input intervals. By the end of algorithm 4.3, the segment tree will have all the nodes initialised with their proper span and connected properly with their children nodes. Algorithm 4.4 takes the initialised segment tree and updates the cover list of all the nodes by inserting the set of input segments from the root node. This is the final step of the build process. After this, the segment

tree will not need to be updated anymore and we can perform query on it later.

Algorithm 4.3 Initialize SegTree(Array<Egdes> elementaryEdges)

1: Initialize treeNode with 2*(size of elementaryEdges)
2: Assign the elementary edges to all the leave nodes
3: Recursively assign the parent nodes to be the union of the children intervals

Algorithm 4.4 Build SegTree(Array<Node> treeNode, Array<Egdes> edges)

1: **for** edge e in *edges* **do**
2: **traverse** node n in array $treeNode$
3: **if** n contains e **then**
4: add e to cover list of n
5: **end if**
6: **end traversal**
7: **end for**

Algorithm 4.5 shows the steps involved in a single sequential query on a Segment tree. The inputs are Segment tree edges and a query point. The output depends on the type of query.

Algorithm 4.5 Query SegTree(Point q)

1: **for** edge e in *edges* **do**
2: **traverse** node n in $treeNode$
3: **if** q is found **then**
4: traverse backward to return result
5: **end if**
6: **end traversal**
7: **end for**

Edge list contains the actual intervals in our input. When all the endpoints of the interval are sorted, elementary intervals are each of the consecutive intervals that are in the sorted group of endpoints. Cover list for each node is the list of all intervals that fall completely within that node's interval. Since we are building a tree using 1D array, we can traverse the tree in a binary heap-like fashion. If index of node is i, then its children are at index $(2 \times i)$ and $(2 \times i + 1)$. Similarly, parent of child at index i is located at index $i/2$. Root node is at index 1.

Algorithm 4.7 shows a version of the segment tree build phase with the commonly available recursive code with STL data structures used. We have parallelized this version with OpenMP task for recursive functions. Parallelization of recursive code requires a main caller which creates the parallel region. Then inside, the single directive calls the recursion function that does the main work. The recursion function is encapsulated in the task directive which runs everything inside in a parallel thread. Whenever a shared object has to accessed we can use openMP locks. The locks have to be pre-initialised and have to correspond to each node. Whenever we are accessing a particular node inside a parallel construct we can use set the lock corresponding to that node to be locked and unlock in exit. This locking mechanism is also useful while adding to coverlist of each node. Since locks have a one to one relation with each node, there is no chance of synchronisation errors.

```
For initialization
nodeIdx = 1
root = SegTree.treeNode[1]

for left traversal
leftIdx = 2*nodeIdx
leftNode = SegTree.treeNode[leftIdx]

for right traversal
rightIdx = 2*nodeIdx + 1
rightNode = SegTree.treeNode[rightIdx]

for back traversal
parentIdx = currentIdx/2
```

```
parentNode = SegTree.treeNode[parentIdx]
```

Listing 4.4: Traversal of the Segment Tree

4.5.1 Segment Tree Construction on CPU

Each of the nodes of the segment tree has a cover list. When building sequentially, each of the node's cover list gets updated. However, when building in parallel, we need to lock the node that is getting updated to avoid multiple threads accessing the same node. This is shown in Algorithm 4.6.

Algorithm 4.6 Build SegTree(...) in Parallel

```
1: #pragma parallel for loop
2: for edge e in edges do
3:     traverse node n in treeNode
4:     if n contains e then
5:         #pragma set lock
6:         add e to coverlist of n
7:         #pragma unset lock
8:     end if
9:     end traversal
10: end for
```

4.5.2 Time and Space Complexity

Time complexity of building the Segment Tree includes finding and sorting all the elementary intervals in $O(n \log n)$. n is the number of line segments. The number of elementary intervals is at most twice the number of edges. Cover lists are generated by inserting all edges into the tree. The overall time complexity is $O(n \log n)$.

A stabbing query requires a depth-first traversal of the tree. Hence, time complexity of a single query is $O(h)$. Segment tree is a balanced binary search tree. So, query requires $O(\log n)$ comparisons. By design, an edge or a segment can be at most in two nodes at a given level. This leads to $O(n \log n)$ space complexity.

Algorithm 4.7 Recursive Build Skeleton with task directive

```
1: function recBuildSkeleton(Node node, int start, int end)
2:       if (start == end) then
3:         #pragma omp set lock
4:           node.interval = elementaryVec[start]
5:         #pragma omp unset lock
6:         return
7:       else
8:         int mid = (start + end) / 2
9:         Node left = leftChild(node)
10:        Node right = rightChild(node)
11:      #pragma omp task firstprivate(left, start, mid)
12:           { recBuildSkeleton(left, start, mid) }
13:      #pragma omp task firstprivate(right, mid, end)
14:           { recBuildSkeleton(right, mid+1, end) }
15:        #pragma omp taskwait
16:        node.interval = left.interval ∪ right.interval
17:      end if
18: end function

19:  #pragma omp parallel
20:  {
21:    #pragma omp single nowait
22:      { recBuildSkeleton(root, 0, lastElementIndex) }
23:  }
```

Algorithm 4.8 Regular Construction of Tree Skeleton (Not Cache-optimized)

```
1: for (i = H; i > 0; i = i − 1) do
2:     for (j = 2^i; j < 2^(i+1); j = j + 2) do
3:         parent = getParentNode( j/2 )
4:         child = getChildNode ( j )
5:         sibling = getSiblingNode ( j+1 )
6:         parent.interval = child.interval ∪ sibling.interval
7:     end for
8: end for
```

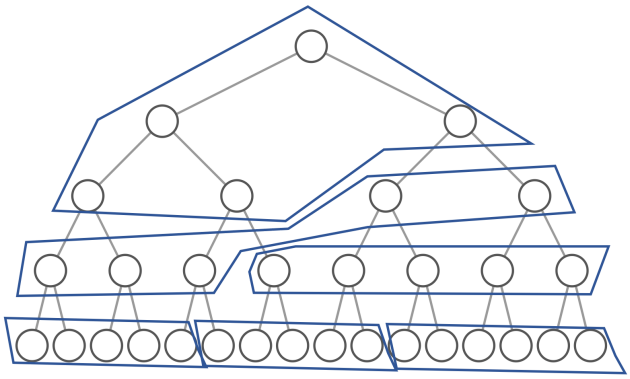

Figure 4.3: Segment Tree stored in a binary heap-like fashion.
Nodes stored level by level. Five consecutive nodes are grouped together in the figure to show six partitions.

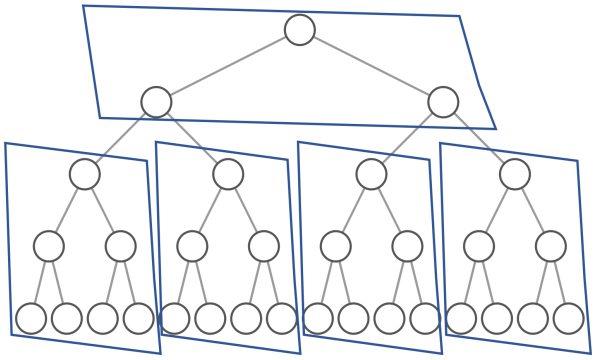

Figure 4.4: Illustration of cache-aware subtree-based segment tree construction
The nodes in a subtree are grouped together in a parallelogram to show that nodes in a subtree are processed first before processing the nodes in the next subtree. The nodes in a subtree and the overall tree are accessed in a bottom-up fashion.

4.5.3 Cache Efficient Segment Tree

For analyzing the cache efficiency of the data structure, we use similar terminology and assumption as used in external memory and cache-oblivious algorithms literature. Lets assume a total cache memory of M, with cache block of size B each. Then the cache can store upto M/B blocks [43]. Each read operation from the main memory will trigger a transfer of B words to the cache. Assuming each node in the tree is of size U, U/B nodes will be read into each block of cache. Lets assume a balanced binary tree of height H with 2^H leaves in the tree.

Algorithm 4.9 Cache Efficient Construction of Tree Skeleton

```
1: for (i = H − 1; i > 0; i = i − 1) do
2:     for (b = 0; b < (2^i/B); b = b + 1) do
3:         for (j = (2^i + b ∗ B); j < (2^i + b ∗ B + B); j = j + 1) do
4:             parent = getParentNode( j )
5:             child = getChildNode ( 2*j )
6:             sibling = getSiblingNode ( 2*j+1 )
7:             parent.interval = child.interval ∪
8:                                 sibling.interval
9:         end for
10:    end for
11: end for
```

During the build phase, when the nodes from any level are read, U/B nodes are loaded into cache. For simplicity, we assume that U is greater than B. And if the parents of those nodes are loaded, another U/B nodes are loaded. But this chunk of parents also contain parent nodes of the other contiguous U/B children nodes. In the standard algorithm, all children nodes at the leaf level are accessed first by their parent nodes to compute the union of the child node intervals during the construction of the skeleton of the Segment tree. This access pattern is shown in Algorithm 4.8. Since the parent nodes will be accessed again in the higher levels of the tree, this may not be cache-efficient because the parent nodes may get evicted from cache because of least recently used (LRU) policy.

In order to leverage temporal locality, we present a novel subtree-based access pattern in Algorithm 4.9 where we access the nodes of the tree in a different manner than the standard way [28]. Our new algorithm exploits temporal locality by reusing the nodes already loaded in the cache. Our results show that our optimization leads to less cache misses than the standard algorithm.

In order to leverage temporal locality, we have used blocking/tiling approach in segment tree construction. Cache-optimized tree construction steps are shown in Algorithm 4.9. When compared to Algorithm 4.8, Line number 2 of Algorithm 4.9

shows the additional for loop that has been added to traverse the array in terms of blocks of size B. Different levels of the tree has different number of tree nodes. The number of blocks in a given level decreases as the tree is traversed in a bottom-up approach. So, the number of blocks for a given level is calculated in Line number 2. The innermost loop is adjusted to process elements within a given block. Our experimental results show that the new access pattern leads to less cache misses than the standard algorithm.

For multiple concurrent queries, we execute queries in multiple sequential batches. If the queries are sorted and split into batches, each batch will have a temporal locality while accessing the nodes of the tree, and each query within a batch will be able to reuse the cache entries loaded by the previous query, thereby reducing the number of main memory accesses.

4.6 Communication Avoiding Distributed Segment Tree

Distributed Segment Trees usually are of two types: a naive one, where the tree nodes are just distributed among the processor and a bit enhanced one implemented with Distributed Hash Tables. Our analysis here, we are looking into naive segment trees. Assuming a tree of Height H with N number of nodes and P number of processors. If the tree nodes are simply distributed among the processors, Communication costs would be $O(H)$ because each traversal could lead to a different node in a different processor and atmost we would have to traverse H steps.

We can reduce communication in Distributed Segment Trees if traversals to the tree doesn't jump to another processor. This can be created by simply portioning whole tree into distinct segment trees in a way that each processor in the distributed system has a tree node which is a subtree of the complete tree but also is a whole segment tree for a given interval. If the height of each sub tree were to be H_2, then number of jumps between processor during traversal would decrease by a

factor of H_2 and hence reducing communication costs. Each subtree would have $(2^H/P)$ number of nodes and height of $H_2 = H - logP$.

In a simply distributed tree, communication costs would be $O(H)$ but now with the subtree approach the communication costs would reduce to $O(H/H_2)$.

4.6.1 Building on GPU

Segment tree memory space requirement can be determined using its space complexity of $O(n \log n)$. However, the distribution of input line segments in the nodes is data dependent and irregular. In particular, the size of cover list for each node of a Segment tree is variable. This variability makes the efficient implementation challenging on a GPU. Therefore, some pre-processing is required to count the size of each cover list to organize the input edges in a hierarchical tree layout in the construction phase. This can be implemented by doing memory allocation in CPU and by using atomics on GPU while inserting an interval in a cover list to allow concurrent insertions on a shared cover list by multiple threads.

In our design, we create a 2D array layout of $n \times m$ dimensions, where n is the number of nodes in the segment tree and m is the upper bound of any node's cover list size. This way, we use extra memory space, but we avoid the overhead of maintaining carefully indexed cover list array. Variable sized cover list array organization would result in irregular memory access pattern which is not ideal for executing parallel SIMD instructions. Moreover, we avoid the necessity of locks because of our data structure design.

In the 2D array layout, each m dimension array belongs to one of the n node. We have individual variables that keeps track of how many elements (edge ids) have been written to each m dimension array i.e. the current count. Whenever a parallel thread has to write into a location in any m dimensional array, we use atomic

update to increment the current count by one. This incremented value is kept in a thread local variable. So even if another thread updates the same count value, the thread local variable will have the value which just increased and will point to an index that is empty and where we can add the edge id.

Estimating cover list size: In Geographic Information System, the distribution of line segments in a real-world map has been studied earlier. For plane sweep algorithms, there exists research on estimating the number of line segments passing through an imaginary vertical line crossing through a given point. For instance, for N_e input line segments, there are at most sqrt(N_e) lines passing through an arbitrary vertical line [110].

For randomly distributed intervals of size N_e, the average cover list size of a segment tree node is around $\log_2 N_e$. We have fixed cover list size (m) to $(2 \times \log_2 N_e)$. This ensures lock avoidance in tree construction because we do not need to lock any part of the tree while populating the cover lists in parallel. For a 100K edge dataset, we have used m to be $2 * \log_2(100k)$ which is approximately 32. For *Parks* and *Cities* datasets, we built the two trees using 200K intervals present in *Parks*. The tree had a maximum cover list size of $2 * \log_2(200k)$ which is approximately 35.

In OpenACC, we need to know the amount of data we are passing from CPU host to GPU device. Once we have this information, we can allocate memory in the host and pass it to the device and do post-processing in CPU after kernel function has finished execution. So, we pre-allocate extra memory and do filtering later as a post-processing step. We have used two times more memory space than a regular design to simplify array indexing and to avoid locks. As we show in the result section, this design has worked well.

4.6.2 Implementing Parallel Stabbing Query

Single stabbing Query: After building the Segment tree, we can run a type of query called stab query; which is basically a query to find all the edges that exist in a particular x position.

This type of query can be basically of three types:

1. One is a boolean type of query where we try to find out if any edge exists or not at a stab location
2. Second is query where we get the number of edges that exist at a stab location
3. Third is a query where we get the list of all edges that exist at a stab location

Each type of stab query can have different applications.

Multiple stabbing Query: We can run a collection of multiple stabbing queries in a batch. Segment Tree is a static data structure which means that it does not change after it is built. Therefore, queries on a segment tree do not update the tree and hence there will be no conflict in running multiple queries concurrently. In this chapter, each individual query has not been parallelized. Single query parallelization did not result in performance gain. Our datastructure supports concurrent multiple queries.

The main challenge in implementing query operation on GPU is the output-sensitive nature of query. In other words, the query efficiency depends not only on the size of the input but also on the size of the results returned for that query.

Multi-threaded CPU query: Since the tree is a static structure, we store pointers to the intervals to return the result of stabbing query on a CPU. For parallelization, each query runs on its own thread. Therefore, the queries are

independent and do not require locks.

Query operation on GPU: Again, as with building the segment tree, querying also will give us variable results. For this, each query result is stored in an array of ids. We can also do this in two steps, first run a query to find the number of intervals in a stab and then allocate memory to accommodate all the ids accordingly in the host and send it to the device. Or, we could do it in one step by creating a 2D layout with a heuristic on the upper bound for queries. However, since traversal is inexpensive, we choose to run it in two steps where we first get an array with indices corresponding to the number of edges in the stab. Next, we allocate memory accordingly in the host. Also, we calculate a cumulative sum array in the host and we pass this also to device for straightforward indexing. In the device, we fill this new array with ids from each stab query and bring it back to the host. Using the cumulative sum array, we find the edges belonging to each stab query.

4.7 Geometric Operations using Segment Tree

4.7.1 SweepLine with Segment Trees

The sweepline approach for finding line intersections sweeps through all the endpoints and if found new intersection points. Once the segment tree is built, we can use the stab queries in parallel to behave like sweep.

4.7.2 MBR Intersections with Segment Trees

In problems such as polygon-polygon intersection, we want to find line segments from the two polygons that intersect with each other. Line segment intersections within one polygon (self-intersections) are not reported. This is implemented by labeling line segments of one polygon with one abstract color (say red) and labeling line segments of another polygon with another abstract color (say blue).

The combination of horizontal and vertical intervals from a polygon makes the Minimum Bounding Rectangle (MBR) of the polygon. We used segment tree to find polygon pairs with intersecting MBRs for city and park boundaries datasets. Two segment trees, one for vertical intervals and one for horizontal intervals were used. An id parameter was used to keep track of polygons and the red-blue property to keep track of whether it belongs to park dataset or city dataset.

4.7.3 Point-in-MBR Test

In GIS, polygonal geometries are often approximated as a minimum bounding rectangle (MBR) in the filter-and-refine based algorithms. An MBR takes less storage space and using MBR as a proxy for a polygon speeds up spatial join and polygon overlay workloads. Point-in-MBR test is a simple variant of point-in-polygon test. Point-in-MBR test can be used a filter for the actual point-in-polygon test.

Point-in-MBR check determines if a query point lies within an arbitrary polygon's MBR or not. We used two segment trees, one for vertical intervals and another for horizontal intervals, corresponding to the horizontal and vertical intervals of the MBRs. A single query has an input point (x,y) to be queried against one or more polygons. All the polygons are referred to by their ids. Each MBR will be represented with the same id in both horizontal and vertical segment tree. Then the point is queried as a stabbing query that we discussed earlier. The result of x and y queries produces two list of ids from each segment tree. Then we compare the two lists for matching ids. These matching ids represent the MBRs that contain queried point.

4.8 Experimental Results

We have tested our algorithm with 2 types of data

1. Simulated Data: We generated uniform random intervals for a given search space.

2. Real Data: Cities and Park Boundaries from the SpatialHadoop dataset were used to get real data distribution. We randomly selected 200K edges from the *Parks* dataset and 500K edges from *Cities* dataset. [111] Then, keeping the distribution intact they were scaled and transformed to our selected search space for consistency across experiments.

Queries: For all cases, queries were randomly generated using a uniform random distribution in the designated search space.

Experiments were performed on the Intel Xeon E5 CPU, which is a 18 core 36 thread CPU with a processor base frequency of 2.10 GHz and on Nvidia TITAN V, which has 5120 cuda cores at base frequency of 1.20 GHz. On the Intel Xeon E5 v4, there is L1 instruction cache of 32KB per core and similarly L1 data cache of 32KB per core. There is mid-level cache (MLC) or L2 of 256 KB per core. The last level cache L3 is a shared inclusive cache of 2.5 MB per core. In the following tables, 1T, 2T, 4T and so on refer to the number of OpenMP threads used. 1T is basically equivalent to running the program sequentially. The Edges column refer to the size of the static segment tree, as to how many intervals or segments was the tree build up with. The column Queries refer to the number of queries used in the experiment. Also, K stands to thousands and M stands for Millions. To print in table some of the values have been rounded.

As shown in Figure 4.5, we can see that the speedup is consistently proportional to the number of threads. The speedup can be observed to be better when there are more edges (line segments). Input per thread increases as the size of the input dataset increases. Therefore, the work per thread also increases as the size of the dataset increases. It takes around 2 seconds for building a segment tree with

Table 4.1: OpenMP Build Time (in seconds)

Edges	1T	2T	4T	8T	16T	32T
10k	1.97	1.51	0.70	0.41	0.22	0.13
20k	7.02	4.79	2.36	1.17	0.69	0.40
40k	24.67	14.90	8.08	3.75	2.11	1.16
60k	60.20	26.45	15.02	7.49	4.22	2.22
80k	90.45	45.40	22.51	12.49	6.69	3.51
100k	141.56	67.92	32.14	17.32	9.62	4.93

Table 4.2: OpenMP Build Speedup(x) compared to 1T

Edges	2T	4T	8T	16T	32T
10k	1.305	2.814	4.805	8.955	15.154
20k	1.466	2.975	6.000	10.174	17.55
40k	1.656	3.053	6.579	11.692	21.267
60k	2.276	4.008	8.037	14.265	27.117
80k	1.992	4.018	7.242	13.52	25.769
100k	2.084	4.404	8.173	14.715	28.714

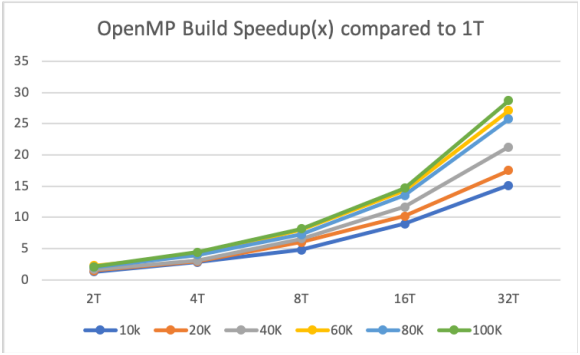

Figure 4.5: OpenMP Build Speedup

10K edges using 1 thread and 5 seconds to build a segment tree with 100K edges using 32 threads. Synchronization overheads due to locks on segment tree also vary as the data is scaled. Even though the edges are inserted to the segment tree concurrently during the build phase, the vast number of edges makes locks occur at different sections and thus potentially reduce the number of lock conflicts and hence yield a proportional speedup compared to the number of threads. This lock conflict happens at a higher rate for higher numbers of edges and we can clearly see this behavior from the graph where the higher number of edges have a higher speedup.

Figure 4.6 shows the timings for our OpenMP loop-based implementation compared to the commonly used STL based recursive code. In the figure, the dashed curves represent the recursive version for different datasets with varying number of OpenMP threads and the solid curves represent our loop-based iterative implementation. We can observe that the iterative loop-based OpenMP implementation consistently performs better than its corresponding recursive implementation. The difference is not that significant for smaller datasets but the gap grows as the datasets grow larger and more threads are used. The execution time of the recursive task-based version starts plateauing after 8 threads for larger datasets. On the other hand, the iterative version performs better than the recursive version and continues to give significant speedups upto 32 threads.

Table 4.3: OpenMP Query Time (in seconds)

Edges	**Queries**	**1T**	**2T**	**4T**	**8T**	**16T**	**32T**
10K	100M	9.64	5.50	3.40	2.25	1.51	0.91
20K	200M	19.37	10.97	6.67	4.64	2.99	1.66
40K	400M	34.78	18.67	9.43	5.15	2.92	1.53
60K	600M	49.03	25.99	13.92	7.61	4.31	2.29
80K	800M	48.95	24.92	12.65	7.14	4.06	2.12
100K	1000M	61.28	31.39	16.65	9.53	5.05	2.62

Table 4.4: OpenMP Query Throughput (queries per second)

Edges	**Queries**	**1T**	**2T**	**4T**	**8T**	**16T**	**32T**
10K	100M	10M	18M	29M	44M	66M	110M
20K	200M	10M	18M	30M	43M	67M	120M
40K	400M	11M	21M	42M	78M	137M	261M
60K	600M	12M	23M	43M	79M	139M	262M
80K	800M	16M	32M	63M	112M	197M	377M
100K	1000M	16M	32M	60M	105M	198M	381M

Since segment trees are static structures, once the structure is built, multiple querying is entirely an embarrassingly parallel operation. We can observe from Figure 4.7 how the speed up is linearly proportional to the number of threads. For 1 thread to run 100M queries on a segment tree with 10k edges it took about 9.64

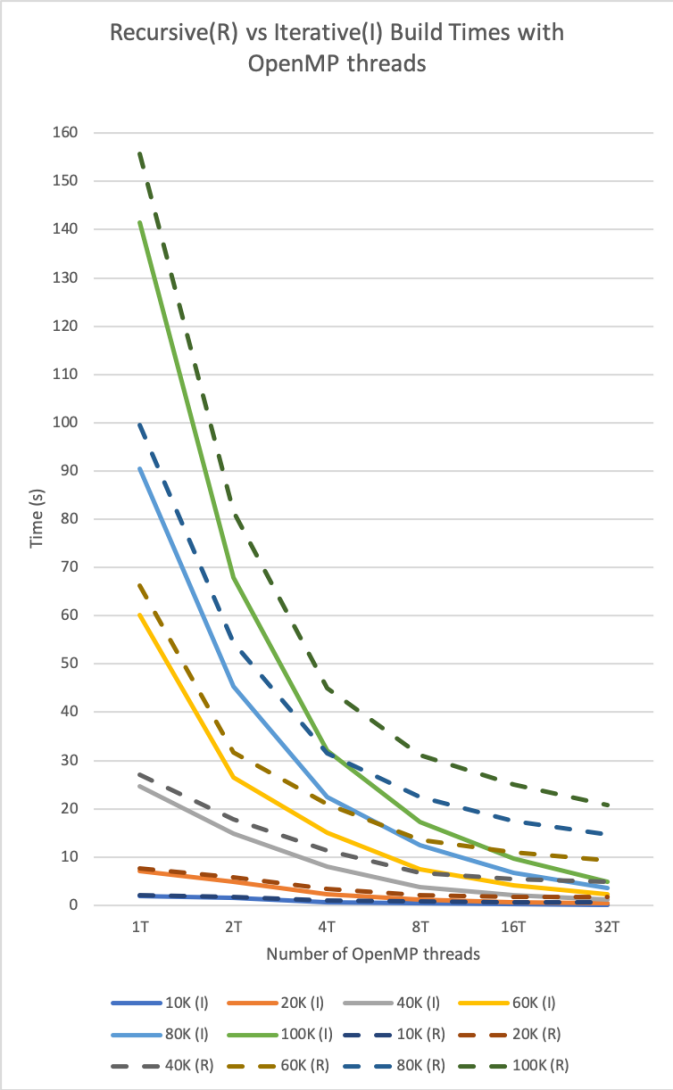

Figure 4.6: Comparison of Iterative OpenMP Vs Recursive task-based OpenMP Iterative OpenMP Code (Solid Curves) with Recursive task-based OpenMP Code (Dashed Curves) for Segment tree construction. Execution time is reported for different sizes of data and number of threads. Best viewed in color.

Table 4.5: OpenMP Query Speedup(x) compared to 1T

Edges	**Queries**	**2T**	**4T**	**8T**	**16T**	**32T**
10k	100M	1.753	2.835	4.284	6.384	10.593
20k	200M	1.766	2.904	4.175	6.478	11.669
40k	400M	1.863	3.688	6.753	11.911	22.732
60k	600M	1.886	3.522	6.443	11.376	21.410
80k	800M	1.964	3.870	6.856	12.057	23.090
100k	1000M	1.952	3.680	6.430	12.135	23.389

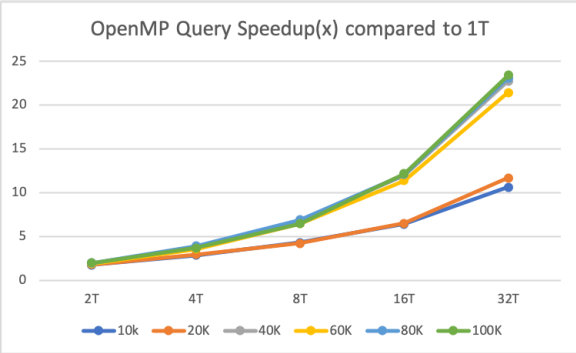

Figure 4.7: OpenMP Query Speedup for Segment Tree

seconds while for 32 threads to run 1000M queries on a segment tree with 100k edges it only took 2.62 seconds. One interesting thing we can observe is that the speedup for the 10k and 20k queries is lower than for the others. This behavior can be attributed to smaller number of queries having a higher chance of colliding in the same section of the tree and slowing down the query process. However, as the number of edges and queries grow, the chances of collision also decreases and we can see almost uniform speedup with the increase in number of threads.

Table 4.6: OpenACC Build and Query Time (in seconds) and Speedup compared to sequential run

Edges	**Queries**	**Build Time**	**Queries Time**
10K	100M	0.06 (33x)	0.16 (60x)
20K	200M	0.13 (54x)	0.31 (62x)
40K	400M	0.25 (99x)	0.62 (56x)
60K	600M	0.72 (84x)	0.90 (54x)
80K	800M	1.29 (70x)	1.18 (41x)
100K	1000M	1.78 (80x)	1.54 (40x)

Table 4.7: OpenACC Build and Query Speedup(x) compared to 1T sequential

Edges	**Queries**	**Build Speedup**	**Queries Speedup**
10K	100M	32.824	60.25
20K	200M	54.019	62.484
40K	400M	98.688	56.097
60K	600M	83.608	54.478
80K	800M	70.116	41.483
100K	1000M	79.526	39.792

Even when run on accelerators, segment trees hold up the performance and speedup. We can observe from Table 4.6 that there is a consistent speedup in the build for different edge sizes. However, it can also be observed that the speedup is not constant or linear. This can be attributed to how the segment tree gets mapped in the gpu memory and when edges are inserted in parallel how different parts of the segment tree are accessed. In general the less the collisions during insertion, the more will be the speedup. We are not limiting the GPU resources in our experiment and are allowing the program to take the as much resources as required. Under optimal conditions we can see that we can achieve almost upto 100 times speedup for building the segment tree.

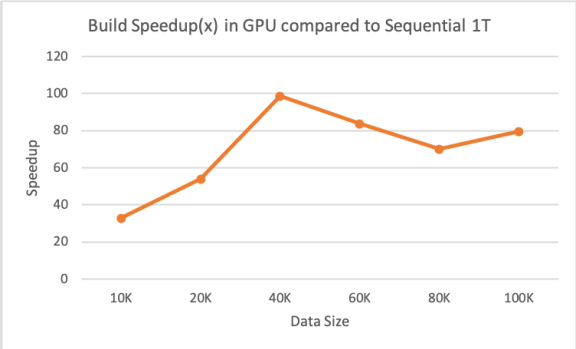

Figure 4.8: OpenACC Build Speedup

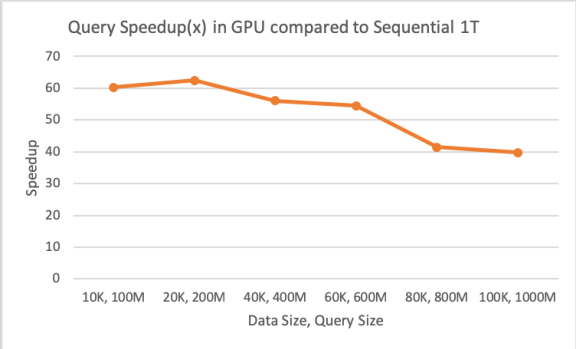

Figure 4.9: OpenACC Query Speedup

Similarly, when we query the segment tree on GPU, we can observe a consistent speedup in the speedup portion of Table 4.6. However, the speedup seems to declining with larger tree size and query size. Since the complete GPU is being

used in all the test cases, as the workload increases, it is natural to see an increase in time and a decrease in speedup. However, the overall speedup is still significantly higher.

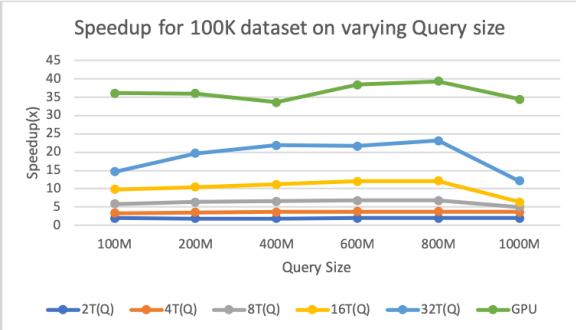

Figure 4.10: Speedup Comparison for 100k Dataset

Table 4.8: Speedup for 100K dataset on varying Query size

Queries	**2T**	**4T**	**8T**	**16T**	**32T**	**GPU**
100M	1.952	3.361	5.884	9.888	14.606	36.049
200M	1.911	3.525	6.413	10.418	19.668	36.04
400M	1.918	3.721	6.548	11.237	21.899	33.557
600M	1.985	3.742	6.807	12.075	21.685	38.418
800M	1.954	3.812	6.848	12.098	23.124	39.353
1000M	1.952	3.68	5.073	6.432	12.145	34.428

Table 4.9: Comparison of Regular and Cache Optimized Time (in seconds) and speedup(x)

Edges	**Regular**	**Cache Optimized**	**Speedup**
10K	1.97	1.47	1.340
20K	7.02	5.49	1.280
40K	24.67	17.48	1.411
60K	60.2	48.59	1.239
80K	90.45	68.07	1.329
100K	141.56	117.10	1.209

From Figure 4.10, we can observe the speedup performance for the 100K edges segment tree when massive queries are performed. We can observe that the speedup is highest in the case of GPUs and it acts like the upper bound for all these query sizes. Also, it can be observed that the speedup holds for most cases and

consistently drops for the 1000M queries case. This shows that having a big tree size is good to avoid collision upto a certain query size but if we keep increasing the query size then at some point we will start to see the drop in speedup.

Table 4.9 shows speedup when compared with the cache optimized segment tree. We can observe that the speedup is around 1.3 times with the cache optimized segment tree. Furthermore, we used perf tool to analyze the cache miss rate with the optimized algorithm and found it to be lowered by around 25%. For multiple querying we observed that if the queries were in an unsorted order, it would not be much beneficial from cache point of view. However, sorting the queries would give a lower cache miss rate but increase the collision chances and would hamper per-query rate. The way to avoid this would be to run contiguous blocks of query for each thread, and having no two threads query up from the same leaf level.

Our segment tree implementations were also compared with the implementations from the Computational Geometry Algorithms Library (CGAL) [99]. While CGAL performance was better for sequentially building segment trees, especially large trees with higher number of intervals, our GPU implementation and multi-threaded implementations were consistently better by factors of 20x and 8x respectively. However, the performance of our implementations were far better, more than 2500x on average better, for running massive queries on large trees. This behaviour can mostly be attributed to CGAL using STL libraries and dynamic on-the-go memory allocation for queries compared to ours bare-bones approach.

Table 4.10: Finding MBR pairs from cities and park data with Segment Tree [Only Query Time (in seconds)]

Parks(200K) Cities(500K)	**Query Time**
R-tree(GEOS) Sequential	11.73
1T Sequential	20.57
16T OpenMP	3.89
GPU OpenACC	0.48

From Table 4.10, we can observe the time taken to find MBR pairs where Parks was the base layer and Cities was the query layer. We have used the GEOS library as a baseline for this comparison. As we can observe, our sequential implementation doesn't perform better than heavily optimized GEOS library. However when we use multiple threads, the performance heavily increases and furthermore the use of a GPU completely outperforms any other test case. With GPUs, we get a speedup of almost 20 times when compared to the base case.

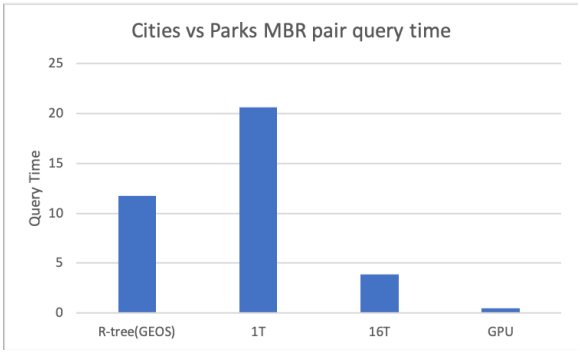

Figure 4.11: Cities Vs Park MBR pair query times

Table 4.11 show the timing for performing point in polygonMBR queries with segment trees. We can observe that with increasing number of threads for each number of queries the timing improves. GPU timings are however the best ones in this case too.

Table 4.11: Point in Polygon MBR for 100K dataset on varying Query size [Only Query Time (in seconds)]

Queries	**1T**	**2T**	**4T**	**8T**	**16T**	**32T**	**GPU**
100M	12.29	6.31	3.57	1.97	1.21	0.78	0.35
200M	24.46	12.84	6.92	3.80	2.32	1.23	0.65
400M	49.01	25.47	13.15	7.49	4.28	2.29	1.47
600M	74.55	37.52	19.96	10.99	6.14	3.47	1.95
800M	98.41	50.28	25.83	14.34	8.12	4.26	2.51
1000M	122.56	62.78	33.29	24.18	19.10	10.11	3.61

4.9 Conclusion and Future Work

We presented compiler pragma-based parallelization of segment tree construction, query and its computational geometry applications. We demonstrated performance improvements due to parallelization in the experimental results section. Performance improvement in terms of speedup is upto 29x speedup for tree construction and upto 23x speedup for batch querying using 32 threads using OpenMP compared to single-threaded version. Using OpenACC, speedup upto 100x during the tree construction phase and speedup upto 62x during the batch query phase have been achieved. Evidently, the speedup for querying is proportional to the parallelization due to the number of threads. We have also demonstrated speedup gains when Segment trees are used in reporting overlapping MBR pairs and Point-in-MBR tests. Also, the cache optimized version of the code is 1.3x faster on average and the cache miss rate is reduced by almost 25%.

We plan to use the proposed parallel segment tree data structure in a third computational geometry application that takes two polygons as input and uses a Segment tree to find out the overlapping area (intersection) between the input polygons. In this application, Segment tree is used to find line segment intersections and point-in-MBR tests in parallel which is required in spatial join and map overlay algorithms.

Segment Tree data structures can be extended to become lock free data structure to eliminate the need of locking during the build phase. Furthermore, Future work also include vectorization of tree operations on CPUs and warp-level parallelism on GPUs using CUDA. It can also be enhanced and applied to do more complex geometric operation. The Communication Avoiding distributed segment tree can be experimented with its Distributed Hash Tree (DHT) counterparts too.

CHAPTER 5: ACCELERATION OF SPATIAL AUTOCORRELATION COMPUTATION

Geographic information systems deal with spatial data and its analysis. Spatial data contains many attributes with location information. Spatial autocorrelation is a fundamental concept in spatial analysis. It suggests that similar objects tend to cluster in geographic space. Hotspots, an example of autocorrelation, are statistically significant clusters of spatial data. Other autocorrelation measures like Moran's I are used to quantify spatial dependence.

Large scale spatial autocorrelation methods are compute-intensive. Fast methods for hotspots detection and analysis are crucial in recent times of COVID-19 pandemic. Therefore, we have developed parallelization methods on heterogeneous CPU and GPU environments. To the best of our knowledge, this is the first GPU and SIMD-based design and implementation of autocorrelation kernels. Earlier methods in literature introduced cluster-based and MapReduce-based parallelization. We have used Intrinsics to exploit SIMD parallelism on x86 CPU architecture. We have used MPI Graph Topology to minimize inter-process communication.

Our benchmarks for CPU/GPU optimizations gain upto 750X relative speedup with a 8 GPU setup when compared to baseline sequential implementation. Compared to the best implementation using OpenMP + R-tree data structure on a single compute node, our accelerated hotspots benchmark gains a 25X speedup. For real world US counties and COVID data evolution calculated over 500 days, we gain upto 110X speedup reducing time from 33 minutes to 0.3 minutes.

5.1 Introduction

In spatial statistics and spatial data mining, there are many methods to discover and explore interesting patterns in spatial data. Spatial autocorrelation is one such class of methods that are used in spatial data analysis. Spatial datasets often are not independent and identically distributed (i.i.d) [49]. Spatial datasets exhibit statistically significant clustering in attribute values under study.

Hotspots analysis is a technique in geospatial analysis used to visualize geographic data in order to show areas where a higher density or cluster of activity occurs. For example, in a city, we can collect crime data from different locations and with hotspot analysis we can see if there are clusters in the city with significantly higher/lower incidence of crime than so by random chance. Two concepts - similarity of values and proximity of locations, or lack of those, are crucial to calculating hotspots and hence requires spatial statistics. Hotspot detection is useful in many fields like public health, crime analysis, schooling, sales, agriculture etc.

We focus on Getis-Ord (Gi*) statistic which is computed for each feature in a dataset. The resultant z-scores and p-values show where features with either high (or low values) cluster spatially. In short, each feature is evaluated within the context of neighboring features. To be a statistically significant hotspot, a feature will have a high value and be surrounded by other features with high values as well.

Hotspots are sometimes confused with a similar spatial visualization technique known as heatmaps. Hotspots differ from heatmaps where point data is analyzed in order to create an interpolated surface showing the density of occurrence where each cell is assigned a density value and the entire layer is visualized using a gradient.

We present performance engineering for Hotspots kernel using SIMD on

CPUs and SIMT (Single Instruction Multiple Thread) on GPUs for exploiting fine-grained vector/data parallelism. For relative speedup calculations, we have used sequential implementation with spatial sorting as a baseline. For absolute speedup calculation, we have used R-tree data structure based implementation. Based on this R-tree baseline, we have demonstrated absolute speedup upto 16X using SIMD + multi-threading on a single compute node. For scalability, our system leverages multiple GPUs using MPI. Our benchmarks for CPU/GPU optimizations gain upto 750X relative speedup with a 8 GPU setup when compared to baseline sequential implementation.

Earlier methods for hotspots problem have used pointer-based tree data structures like quadtree for storing location data and for range query. For effective SIMD/SIMT parallelization, instead of tree data structure, we have designed a novel spatial locality-preserving 2D array-based data structure for weight matrix. On a distributed memory environment, this weight matrix further aids in creating task interaction graph which can be utilized to minimize communication using MPI graph topology functions.

The rest of the chapter is organized as follows. Section 5.2 presents the motivation and background. Section 5.3 presents the parallel formulation for the problem. Section 5.4 presents the acceleration techniques on CPUs and GPUs. Section 5.5 presents the experimental results. Finally, we conclude in Section **??**.

5.2 Motivation and Background

Finding patterns helps us identify causes and predict future trends. For instance, finding hotspots of Covid-19 occurrences enable us to study disease spread and efficient resource allocation to combat the problem at hand. We have identified important autocorrelation kernels in spatial domains for parallelization. In the existing work, the focus has been on coarse-grained approaches with less attention

to data movement aspects and communication complexity [122].

Spatial hot spots detection is crucial in tracking the Covid-19 pandemic and guiding policy by focusing resources to combat its growth. Since it is a world-scale phenomena, real time tracking requires large scale parallelization to implement fast prevention rather than slow intervention.

5.2.1 Spatial autocorrelation

The notion of spatial autocorrelation is related to first law of geography: Everything is related to everything else, but nearby things are more related than distant things [48]. The value of attributes at a given location tend to vary gradually over space. For instance, weather of two adjacent areas tend to be similar. In many cases, events in a given area are influenced by the events at neighboring areas. In spatial statistics, this property is called spatial autocorrelation [49]. A famous example of application of this concept was finding the link between Cholera outbreak and contaminated water in London in 1855 by looking at the clustering of disease occurrences (hotspots) around a water pump. An example of hotspots map is shown in Figure 5.1.

Spatial interdependence of attributes exhibited in data with respect to location and distance is captured by statistical measures like Moran's I. There are many local and global auto-correlation kernels. We focus on a representative and popular kernel - Hotspots. For a set of disease occurrences, finding hotspots aim at detecting disease outbreaks well before it results in a large number of cases. Hotspots are statistically significant clusters of observations based on similarities of values and locations. Hotspot detection is used in many fields like public health, crime analysis, etc.

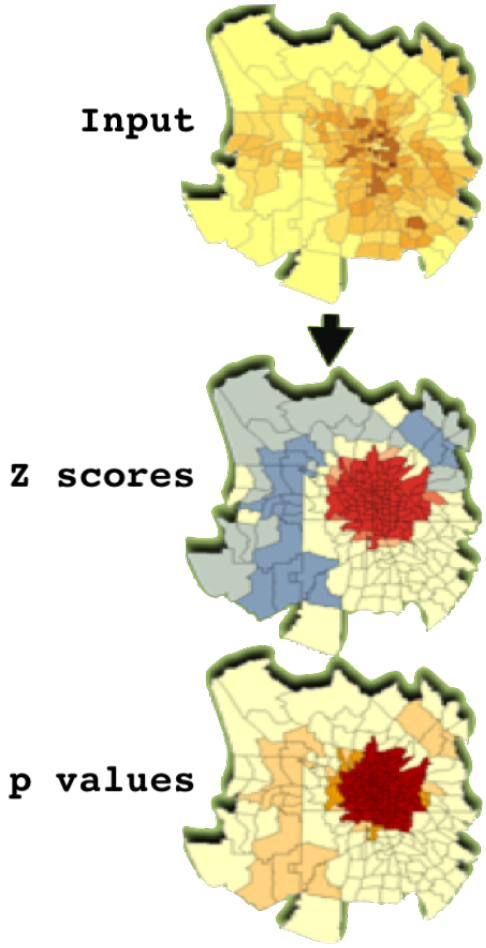

Figure 5.1: Polygon boundaries with their corresponding z scores and p values [3]

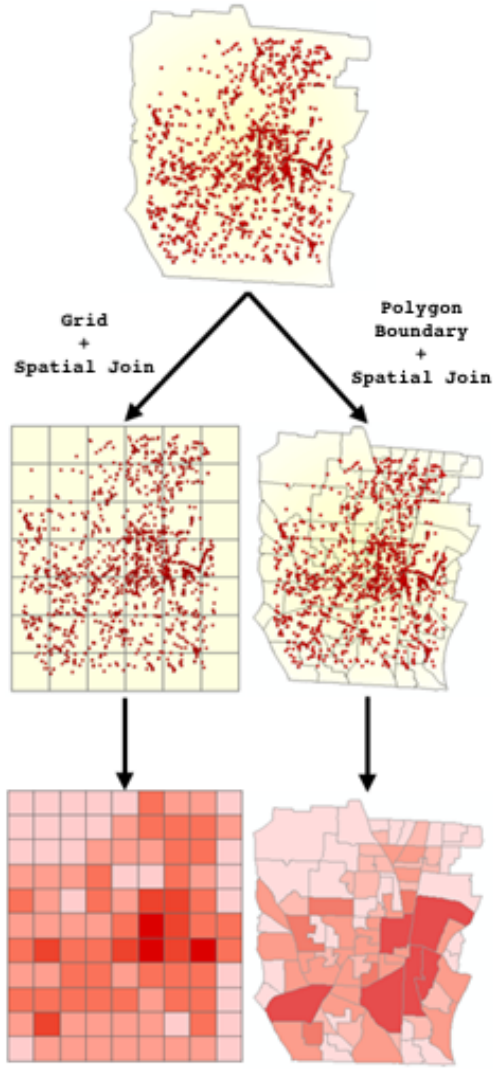

Figure 5.2: Point data overlaid on a Grid vs Polygonal Boundaries [3].

5.2.2 Common Dataset Structures

Data for geo-spatial autocorelation analysis can usually come in 3 forms:

1. Aggregated Boundary data: This is the most typical type of available dataset for which usually a boundary is given and a value corresponding to the boundary is available. The boundary can be a known regular shape like square, rectangular, hexagonal or an irregular polygonal boundary. An example of this would be county level covid cases data. For each county, there is a defined polygonal boundary which is not a regular shape and for each county there would be a corresponding attribute value like active covid cases.

2. Unit point incidence data: This is the type of data where we have geolocation instances of incidents. Here we would have multiple points where each point corresponds to a single incident. Common example of this type of dataset is the crime dataset where each point relates to a reported criminal activity. A covid related example would be having a dataset of all the people who tested positive in a given area. In this dataset, each person would represent an individual incident and the geolocation of their home address would be an incident point.

3. Aggregated point incidence data: This is the type of data where we have instances from an area aggregated at a point. In the crime dataset, the geolocation of the police station could be the incident point and number of complaints are aggregated to get one single attribute value per incident point. A covid related example would be having a list of rapid testing centers, where the geolocation of the testing center is the incident point and the number of all tested positive cases are the aggregate attribute value.

In geospatial analysis, to calculate and show hotspots, boundaries are

required. In the second case, the data can be overlaid on a regular grid of squares, rectangles, or hexagonal shapes. Another approach is to overlay the data on top of a polygonal layer, for instance, boundaries of zipcodes. All the values inside the boundary can be aggregated and used as the corresponding attribute value for the polygonal boundary. Figure 5.2 shows an example of data being overlaid on a regular grid and a polygonal map. Depending on the choice of data overlay, the computational cost will vary. A regular shape boundary, moreover a square grid, would have the least amount of computation and complexity.

In the third case, approaches from the second case can combine data from multiple points to make it fall inside a boundary. However, the disadvantage of doing this is there might be an imbalance in the distribution of data among boundaries. For example, in the covid cases data from the free rapid testing centers, it would be a reasonable assumption that people went to the testing center closest to them. Similarly, for the crime dataset, it would be a reasonable assumption that complaints were reported to the closest police station. If such proximity assumptions are reasonable, then a better way to divide boundaries for each aggregate incident points would be a voronoi distribution. The voronoi distribution guarantees that there is an unique boundary and area for each point and for any location inside the boundary, the given incident point is the closest incident point.

5.2.3 Parallelization

Vector/SIMD Intrinsics: Vector/SIMD extensions of Instruction Set Architecture are provided by modern CPUs for single instruction steam, multiple data stream (SIMD) processing. For x86 CPUs, special wide registers and vector instructions are provided for parallel processing at the instruction set level. For instance, x86 processors provide AVX (advanced vector extensions) instructions. ARM processors provide neon extensions. In this chapter, for effective SIMD

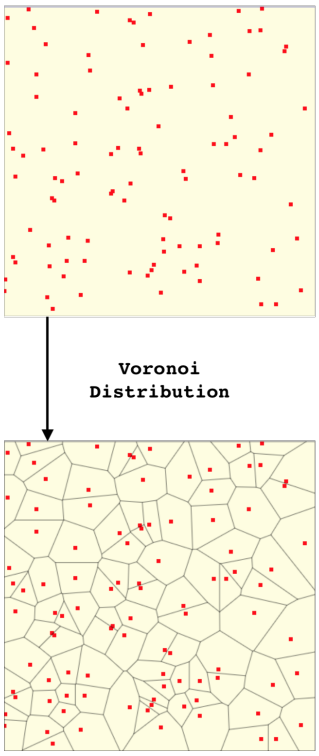

Figure 5.3: Voronoi Boundaries for aggregated point incidence data

parallelization, we have used AVX instructions through C functions (called intrinsic functions). Intrinsics are replaced directly to vector instructions without the overhead of function calls. In this chapter, we achieved better performance when compared to compiler generated vectorization of our computational kernels.

MPI Graph Topology: Given a process interaction graph, MPI provides support to map the processes on a compute cluster. The application level topology can be mapped to the the physical topology of a network using cartesian and graph topology functions in MPI. Since a good mapping of processes to network topology reduces the data communication volume across the network, we have used graph topology functions in our implementation.

5.2.4 Related Work

With the volume of data increasing due to its spatio-temporal nature, parallelization of existing algorithms have been done [50, 51, 52, 53]. Existing approaches use spatial partitioning methods like quadtree for parallelization [50].

A Matlab-based shared memory parallelization has been described in [53]. Hadoop MapReduce has been used to parallelize Getis-Ord based Hotspots detection problem using quadtree-based decomposition of spatial data [50]. Apache Spark framework has also been used to parallelize spatial hotspot computation [51, 52]. Spark MapReduce papers are short papers from GIS Cup competition organized with SIGSPATIAL conference [51, 52]. Hadoop and Spark based projects make good use of thread-level and coarse-grained parallelism but do not take full advantage of HPC resources (e.g., SIMD, GPUs) thus leaving performance on the table [50, 51, 52].

Compared to related literature, our chapter further explores additional hardware and software parallelization opportunities. GPU SIMT parallelization and CPU SIMD parallelization along with communication optimizations are the novelties compared to related literature.

5.3 Parallel formulation of spatial autocorrelation

We can use Getis-Ord algorithm to calculate the G_i^* statistic for each feature in a dataset [123]. In geospatial analysis, it gives a Z-score statistic G_i^* where x_j is the value for polygon j. $w_{i,j}$ is a weight parameter between polygons i and j which is inversely proportional to the active distance between them. N is equal to the total number of polygons in our dataset. Positive and negative G_i^* values denote hot and cold spots respectively and the absolute value of G_i^* is proportional to the intensity of clustering for the i^{th} polygon.

The equations to the Getis-Ord algorithm are as follows:

$$\overline{X} = \frac{\sum_{j=1}^{n} x_j}{n} \tag{5.1}$$

$$\overline{X^2} = \frac{\sum_{j=1}^{n} x_j^2}{n} \tag{5.2}$$

$$S_X = \sqrt{(\overline{X^2}) - (\overline{X})^2} \tag{5.3}$$

$$W_{X_i} = \sum_{j=1}^{n} w_{i,j} x_j \tag{5.4}$$

$$W_i = \sum_{j=1}^{n} w_{i,j} \tag{5.5}$$

$$W_i^2 = \sum_{j=1}^{n} w_{i,j}^2 \tag{5.6}$$

$$S_i = \sqrt{\frac{\left[n * W_i^2 - \left(W_i\right)^2\right]}{n-1}} \tag{5.7}$$

$$G_i^* = \frac{W_{X_i} - \overline{X} * W_i}{S_X * S_i} \tag{5.8}$$

For Moran's I:

$$W = \sum_{i=1}^{n} \sum_{j=1}^{n} w_{i,j} \tag{5.9}$$

$$I = \frac{n}{W} \frac{\sum_{i=1}^{n} \sum_{j=1}^{n} w_{i,j}(x_i - \overline{X})(x_j - \overline{X})}{\sum_{i=1}^{n}(x_i - \overline{X})^2} \tag{5.10}$$

Values of I usually range from -1 to $+1$. Values significantly below $(1 - N)^{-1}$ indicate negative spatial autocorrelation and values significantly above $(1 - N)^{-1}$ indicate positive spatial autocorrelation. For statistical hypothesis testing, Moran's I values can be then transformed to z-scores.

Geary's C:

$$C = \frac{n-1}{2W} \frac{\sum_{i=1}^{n} \sum_{j=1}^{n} w_{i,j}(x_i - x_j)^2}{\sum_{i=1}^{n}(x_i - \overline{X})^2} \tag{5.11}$$

N is the number of spatial units indexed by i and j. x is the variable of interest; $\bar{x}$ is the mean of x; $w_{i,j}$ is a matrix of spatial weights with zeroes on the diagonal (i.e., $w_{ii} = 0$ and W is the sum of all $w_{i,j}$.

The value of Geary's C lies between 0 and some unspecified value greater than 1, usually lower than 2. Values significantly lower than 1 demonstrate increasing positive spatial autocorrelation. Values significantly higher than 1 illustrate increasing negative spatial autocorrelation. Geary's C is inversely related to Moran's I. Moran's I is a measure of global spatial autocorrelation, while Geary's C is more sensitive to local spatial autocorrelation.

5.3.1 Algorithm

The Algorithm for Getis-Ord is as follows:

1. Load all the Points and their x attribute values.
2. Calculate the mean of all the x values, denoted by $\overline{X}$.
3. Calculate the mean of all the x^2 values, denoted by $\overline{X^2}$.

4. Calculate S, the standard deviation of all the x values.
5. Calculate the values for $w_{i,j}$, the weight metric between polygon i and polygon j.
6. Calculate $w_{i,j}^2$ from $w_{i,j}$.
7. For each i, calculate W_i from $w_{i,j}$.
8. For each i, calculate W_i^2 from $w_{i,j}^2$.
9. For each i, calculate S_i from W_i and W_i^2.
10. For each i, calculate W_{X_i} from $w_{i,j}$ and x values.
11. For each i, calculate G_i^*.

5.3.2 Complexity

The time complexity of this algorithm is $O(N^2)$ and the space complexity of this algorithm is $O(N)$. This analysis of time complexity is contingent on the assumption that inverse distance squared (impedance) is used for $w_{i,j}$ and any similar $O(c)$ method of calculating $w_{i,j}$ would keep the analysis the same. Similarly, for the space complexity no pre-calculations of $w_{i,j}$ are assumed. Pre-calculations of $w_{i,j}$s would make the space complexity to become $O(N^2)$ too.

5.3.3 Weight Matrix

The most common technique of calculating $w_{i,j}$ is the metric called the inverse distance. Distance could be different types but most typically the euclidean distance. Inverse distance is a metric would be a high value for things that are closer and low value for things that are spatially further apart. It should be noted that $w_{i,j} = k \ \forall (i = j)$, where k is a value of no consequence and is just used as a placeholder because in this case both i, j would be the same point so no distance

and undefined inverse distance. On, the other end, objects further than a certain threshold can be deemed to have a inverse distance value of zero i.e. $w_{i,j} = 0$ if $invDist(i, j) < \epsilon$. Also, $w_{i,j} = w_{j,i}$ because both are distance-based quantities which does not vary on direction. Hence, if w was to be modeled as a matrix, it would be a $n \times n$ symmetric matrix with diagonals all k. Basically, it is an adjacency matrix where $w_{i,j}$ corresponds to the weight, as it relates to the spatial relation between two areas i and j.

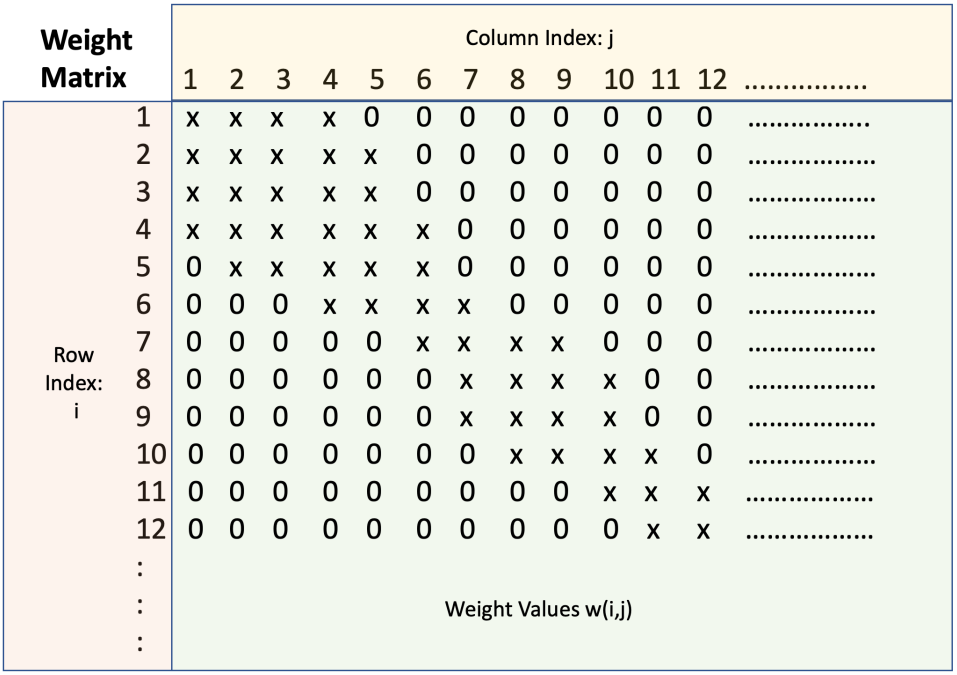

Figure 5.4: Slice of the Weight Matrix
Each row and column index corresponds to a polygon id. For any two polygons i and j, element at index (i, j) is the inverse of the euclidean distance between centroids of i and j.

It might prove efficient for accesses in certain cases if the whole matrix is available even though the second half across the diagonal is just duplication by symmetry.

5.3.4 Spatial Sorting

Spatial sorting is used to arrange 2-dimensional points in 1-dimensional order based on spatial proximity (locality). Space filling curves are used for spatial sorting, such as Z-order [124] and H-order (also known as Hilbert curve). For illustration, let us assume that we have a list of tuples, where the first entry is the x-coordinate and the second entry is the y-coordinate of a point. After sorting the list spatially, points that are closer to each other in the xy plane would appear closer in the list. Proximity of the points - difference in their index values in the sorted list would be an indication of proximity of the points in euclidean space and vice versa.

Having the polygons from our data sorted has special implications for our application and acceleration objectives, especially the affect it has on the weight matrix. Looking at Figure 5.4, we can observe that if the polygons are spatially sorted, then in each row i, the columns that have non-zero entries are only the columns numbered close to the value of i. This is because, as polygons get further apart, their inverse distance decreases and beyond a threshold, they simply become zero. So, for each row i, the columns j for whose values are further apart, their values are simply zero because it represents the underlying property that polygon i and j are just spatially further away from each other.

Expanding upon this property, we will find that for each row i there are only columns in the range $(i - l_i, i + r_i)$ for which the weight values are non-zero. Let l_i be the number of entries to the left of i that are non-zero and r_i be the number of entries to the right of i that are non-zero. Given a large map with lots of polygons, the range $(l_i + r_i)$ can become significantly small, making our matrix a sparse matrix with only elements around the main diagonal being non-zero and elements further away from the diagonal being mostly zeros. For example, with 100k polygons the max range $(l_i + r_i)$ was less than 200.

Furthermore, for the rapid recalculation part, in events where we only have new data for a few polygons and we want to update the scores, the only polygons that require recalculation would be the polygons which have new data and the polygons with which it has a non-zero weight relationship.

Comparison with R-tree: An alternative to using the weight matrix would be the use of a R-tree like approach. Here, our cutoff threshold ϵ from the weight matrix would translate to a certain distance and we would then query the tree to get all polygons within that distance range from the query polygon. We could then calculate weights $w_{i,j}$ for each query polygon i and queried polygons denoted by j. If we use this approach, rather than the sorting and pre-calculating weights, then it would add overheads needed to build a tree. This is in contrast to the tradeoff of sorting all the polygons. Since the locations of the polygons are static, the tree would only be needed to be built once just like the sorting. The advantage of using weight matrix is that the weights will be available in memory easily accessible for SIMD operations. Also, in the cases of the square tiles, sorting is extremely efficient and building a tree would just be an overhead. In an R-tree approach, each polygon will be able to query its list of neighbours and then calculate the corresponding weights with each neighbour. Since the polygons will be unsorted, each weight calculation will access arbitrary areas of the memory and no cache-based gain will be achieved. Also, using a vectorized approach will not be possible without further sorting and ordering because the results of the query may not be in a contiguous memory. The distinct advantage of using R-trees can be that their build cost is not high, their query can be easily parallelizable and storing the weight matrix might not be necessary.

5.4 Acceleration Techniques

5.4.1 Cache Access Optimization

We have three arrays of size N – two are arrays that have the x-location and y-location for each point, and another is an array of attribute values of each point. Let's denote the first two arrays by p and the next array by x. We need to fill a 2D array of size $n \times n$ with $w_{i,j}$s. Let's call this array w. Assuming there is a cache block size of B, whenever calculating any $w_{i,j}$, we get two B blocks of p and one B block of w loaded into the cache, so in this case, instead of linearly calculating the values of w, we calculate all the combination of $w_{i,j}$ that we can from these two blocks of p in an order where we can write into the loaded B block of w. Once we have a filled $w_{i,j}$ matrix array, whenever looping through it, we need to make sure that we access it in the proper order. Looping through $\sum_{j=1}^{n} w_{i,j}$ for a fixed i might be expensive in column-major architectures than looping through $\sum_{j=1}^{n} w_{j,i}$ but since $w_{i,j} = w_{j,i}$ doing both will give the same result.

5.4.2 Weight Matrix Storage Optimization

Since the weight matrix is symmetric, we can store only the upper triangular matrix. Furthermore, since the non-zero values are only near the diagonal we would only need to store at most $maxr = \forall_i \max r_i$ values for each polygon. So, in the worst case, the weight matrix would need $n * maxr$ space compared to its n^2 size. But this approach makes SIMD operations inefficient because we would need to index up or down to find the neighbours to the left of polygon i. Due to symmetry,n^2 and $maxl = \forall_i \max l_i$ would be equal. So, we could store a $n * (2 * maxr)$ array, which is still better than the n^2 array. Here the N rows will be the polygons and $(2 * maxr)$ columns would be weight with the non-zero neighbours. This way, although the storage is doubled from the most compressed

form, being able to access a contiguous memory of weights will significantly improve the cache access and make SIMD operations easily accessible. Furthermore, if the weight matrix is now stored in a file, then, that too can be easily read with contiguous memory access and the amount needed to be read by each process decreases significantly, almost by a factor of $n/maxr$.

5.4.3 OpenMP Parallelization

OpenMP parallelization is based on the equations of the Getis-Ord algorithm as shown earlier. The steps from Getis-Ord algorithm 5.3.1, Step 2, 3 and 5 were parallelized using parallel loops with reduction. All the steps, including calculating each of the G_i^*, are parallelized. If recalculation of results is not required, then steps 5 through 10 can be parallelized to run by each thread for each polygon i along with a second level of parallelism inside the loop for calculating all the sums and G_i^* values. Once the base C/C++ code is written, OpenMP parallelization is extremely straightforward and can be easily achieved using compiler directives.

5.4.4 OpenACC Parallelization

OpenACC compiler pragmas support both CPU and GPU parallelization. We have used OpenACC for GPU parallelization. Compared to OpenMP, additional steps include data copy to GPU (in and out). We have used reduction pragma in OpenACC for additions. For example, in Algorithm 5.3.1, Step 1, once the x values are copied to the GPU, for Steps 2 and 3, we can do reductions to get the summation results. Only the output G_i^* values are copied back to the host CPU. Our OpenACC implementation leverages our existing C/C++ code.

5.4.5 CUDA Parallelization

We have also used CUDA for GPU parallelization of our kernels. Compared to OpenACC, CUDA gives more control in using the GPU. For algorithm 5.3.1, we

added CUDA kernels for each steps. For large datasets that do not fit in the GPU memory, especially the weight matrix whose size grows quadratically in the number of inputs, we do calculations in batches by moving data in and out of the GPU. Data movement between GPU and Host can be an expensive step compared to computation especially when done multiple times.

5.4.6 MPI Graph Topology (Distributed Memory)

Using MPI, process ids are used to split the data among multiple compute nodes for a distributed memory parallelization. We use allreduce collective function to merge the partial results from Steps 2 and 3 of algorithm 5.3.1. We need to broadcast the reduced values to all the ranks as well. Also, for Step 5, each polygon needs to calculate the $w_{i,j}$ values and the MPI ranks need communication to share the location information. We assign a MPI rank to each polygon. This process mapping scheme helps in creating better MPI process topology, which we discuss next.

Given the nature of weights which decays with increasing distance, polygons that are further from each other have a weight of zero. This means that only polygons that are close to each other need to communicate with each other. The Weight matrix can then be utilized to create an adjacency matrix (for graph) where entries in this new matrix are 1, if the weights are greater than zero, and zero otherwise. We translate this polygon adjacency matrix to MPI processes adjacency matrix for each process as required by Graph Topology function in MPI. MPI has methods that can take this adjacency matrix and arrange processes in such a way that minimizes the amount of communication among processes. We have used the following function for graph topology in MPI.

```
MPI_Dist_graph_create_adjacent( MPI_COMM_WORLD, degree,
    neighbours, MPI_UNWEIGHTED, degree, neighbours,
```

```
MPI_UNWEIGHTED, MPI_INFO_NULL, 1, &new_dist_comm);
```

Listing 5.1: Adjacent distributed graph creation

The designated MPI method to use the adjecency matrix would be

```
int MPI_Dist_graph_create_adjacent
(
    MPI_Comm comm_old,
    int indegree,
    const int sources[],
    const int sourceweights[],
    int outdegree,
    const int destinations[],
    const int destweights[],
    MPI_Info info, int reorder,
    MPI_Comm *comm_dist_graph
)
```

Listing 5.2: MPI function to create adjacent distributed graph

Since the weight matrix is symmetric, the indegrees are equal to the outdegrees and the sources are same as the destinations. We have used MPI_UNWEIGHTED because the volume of communication is the same when communication takes place. It is important to set reorder equal to 1, if we want MPI to figure out the best configuration to reduce the amount of cross-node communication. Setting reorder to be true, means that in the new MPI_Comm, the ranks of MPI processes will be different from the global ranks in MPI_COMM_WORLD. Hence, to avoid double loading of the input data (before and after process reordering), we divide the overall data loadin into two stages. In

the first stage we load partial data that is necessary and then load all the other remaining data only after this reorder has taken place. This is efficient and it also ensures that MPI processes will not have data corresponding to their old ranks.

We should also note that there will be designated ranks in which all reduce computations will done for the mean $\overline{X}$ and standard deviation S values and a subsequent broadcast to relay the calculated values to all ranks. Hence, when constructing the adjacency matrix, it would be a good idea to include this information too. However, since these steps are most likely to be a all-to-one and one-to-all types of communication, using the world communicator would suffice too. But passing this information in the adjacency matrix could nonetheless be useful for MPI to arrange the designated rank in the best possible network location.

5.4.7 Communication Efficiency on Distributed Memory

If we have P processes, each process will have N/P polygons and each of them will have to calculate N/P G_i^* values. However, $\overline{X}$ and S are the same for N polygons. So, each N/P process have to calculate those values only once. $\overline{X}$ and S are simply mean and standard deviation, and we can use any of the existing communication efficient algorithms to calculate those. The main communication bottleneck here is that for each polygon i to calculate G_i^*, it needs $w_{i,j}$ and x_j for all N js which means P all-to-all communication steps which is $O(P^2)$ communications. Each broadcast would have to send the appropriate x_j values along with parameters to calculate $w_{i,j}$ values. Using graph topology built on top of a weight matrix that preserves neighborhood information for each MPI process, the communication can be potentially optimized to $O(P)$ communication steps.

However, if during the read phase or reorder phase, the polygons were distributed in such a way that each processes only had contiguous and connected polygons from a region and the neighboring rank processes had polygons from its

neighboring region based on the network topology, we could derive a way to estimate the weight parameter $w_{i,j}$ based on the processes ranks. Further, if we selected the weight parameter such that $w_{i,j}$ is a function of i and j, and for polygon i and j that are far apart $w_{i,j} \to 0$ each G_i^* would have the x_j needed within the processor and its neighbors because the x_js that are far apart would just get multiplied by zero. The $w_{i,j}$ values could just be calculated using the function for far apart polygons or an actual distance metric for the ones within the process or neighbors. This would mean that there would be communication only between neighboring processes and instead of having $O(P^2)$ communication steps, we would only have $O(P)$ communication steps in between neighboring processes.

5.4.8 Vectorization with compiler intrinsics

For single precision floating point data type (32 bits), 8-way parallelism can be potentially exploited by using 256 bit vector register supported by Advanced Vector Extensions (AVX) [125]. AVX-512 intrinsics can support 16-way parallelism because of wider SIMD registers. Intrinsic functions work like inline functions. There is no overhead of function calls because compilers replace these functions with corresponding vector assembly instructions. Our implementation of equations 5.8, 5.1 and 5.3 is geared towards exploiting vectorization via intrinsics. Arithmetic (summations, multiplications, etc), data movement (load/store), and comparison operations are fully vectorized. The denominator and numerator terms for equation 5.8 are also vectorized efficiently.

Assume that a vector floating type can hold v number of floats. In the machines we used this number was $v = 8$. To facilitate vectorization with compiler intrinsics, whenever allocating memory, it is better to allocate a aligned memory. If we look at equations 5.8, 5.1 and 5.3 we can see that there are many summations. These summations can be done vectorically and finally the v floats can be summed

sequentially to get the final result. All calculation for squaring like x^2 and w^2 squares can be done vectorically by loading them into vector types, and multiplying them with themselves and then storing them. So, $\overline{X}$ in equation 5.1 can be calculated vectorically, reducing the number of operations needed to calculate $\overline{X}$ by v. The x^2s needed for S in equation 5.3, can also be calculated vectorically and then be summed vectorically. Same applies for the denominator and numerator terms for equation 5.8. Finally, all each of the G_i^* values can be calculated vectorically. [125]

In Algorithm 5.1, we show an example of using advanced vector intrinsics to calculate the weight matrix using the inverse euclidean distance and setting all weight values below threshold epsilon (epi) to be zero. Broadcast function is used to set all the elements of a SIMD register with the same value that was passed to it as an argument. Please refer to [125] for details on the functions used here.

It can be seen that the code is optimized enough to start vector operations always at aligned memory for each i loop using the second j loop and control variable k. Also, the code only does one calculation for $w_{i,j}$ and $w_{j,i}$ values because they are the same due to symmetry. There is a post-processing step done after this to fill the $w_{j,i}$ values. This will ensure that whenever we need $w[i]$ for any polygon i, we will have the full contiguous memory of size N with values for all $w_{i,j}$.

5.4.9 OpenMP & Vectorization

The next step in speeding up computations would be to combine the techniques to get get even faster code. So, we took the vectorized C/C++ code and used OpenMP threads to parallelize it for shared memory. As long as threads get concurrent access, we will be able to exploit the cache and register operation benefits. On top of our vectorized code, we added thread-level data parallelism using OpenMP to leverage multiple vector units available on modern multi-core CPUs. For this combined parallelization, cache and register memory availability

Algorithm 5.1 Intrinsics based algorithm for calculating weights

Input: N, cutoff value *epi*

Output: populated weights w

```
 1: declare __m256 epis, x1, x2, xx, y1, y2, yy, z
 2: declare int i, j, k and assign k ← 8
 3: epis ← mm256_broadcast_ss(epi)
 4: for (i = 0; i < N; i++) do
 5:     for (j = i + 1; j < k; j++) do
 6:         w[i*N + j] ← invEucDist(x, y, i, j, epi)
 7:     end for
 8:     for (j = k; j < N; j = j + 8) do
 9:         x1 ← mm256_broadcast_ss( x + i )
10:         x2 ← mm256_load_ps( x + j )
11:         xx ← mm256_sub_ps( x2, x1 )
12:         xx ← mm256_mul_ps( xx, xx )
13:         y1 ← mm256_broadcast_ss( y + i )
14:         y2 ← mm256_load_ps( y + j )
15:         yy ← mm256_sub_ps( y2, y1 )
16:         yy ← mm256_mul_ps( yy, yy )
17:         z ← mm256_add_ps( xx, yy )
18:         z ← mm256_rsqrt_ps( z )
        // SIMD compare if z > epis
19:         bmask ← mm256_cmp_ps(z, epis, CMP_GT_OQ )
20:         z ← mm256_and_ps(z, bmask) // (z & bmask)
21:         mm256_store_ps( w + i*N + j, z )
22:     end for
23:     k ← (((i + 1)/8) + 1) * 8
24: end for
```

with multiple parallel threads are the main issues. With reference to code, algorithm 5.1, the approach that gave us the most benefit was to run the i loop in OpenMP parallel regions while maintaining contiguous data access for each thread. If t is the number of OpenMP parallel threads, this can be achieved with using a guided OpenMP schedule with chunk size ck such that $1 < ck < (N/t)$. Having a lower value of ck will split the iterations into threads in such a way that the first among the earlier threads will have the largest chunk size and less memory access overhead, but later threads will have smaller chunks size and higher cache overhead. Also, with multi-threading, it is necessary to keep in mind that depending on the processor, each core will have only a limited number of SIMD registers (usually 32) and limited L1 cache size, so choosing a thread count t that does not overwork each core is necessary to see any benefits from the combined acceleration approach.

5.4.10 MPI & Multiple GPU (CUDA)

If multiple nodes with GPU are available, then MPI can be used to offload much of the processing to the GPUs by combining the MPI and CUDA codes. Once each MPI process has the data it is going to be processing, it can easily copy it to GPU device and get results. This will work even if there are multiple MPI processes running in the node. Even if each node has multiple GPUs, MPI processes can use their rank to select one of the available GPUs and offload their computation. If there are multiple nodes each with multiple GPUs, this same approach will work with the combined MPI. The best way to use MPI with CUDA is to have a separate cuda file with extern C functions that are capable of executing the cuda kernels. Pointer to the data structures from the host's main memory can be passed into this function with useful information like the rank of the MPI process that's calling it. Using cudaGetDeviceCount, cudaSetDevice and the MPI rank, the function can call the kernel and copy back the memory after computation to host using the host

pointers. If multiple GPUs are going to be used in a node, it is also a good idea to minimize all cudaMemcpyHostToDevice and cudaMemcpyDeviceToHost to because that is the step that consumes the most time. So, a preprocessing step to allocate memory on the GPUs and passing back the device pointers to host to use in further calculations is recommended.

5.4.11 Rapid Recalculation

Even in scenarios where the data emerges or changes at certain time intervals, the location based data and spatial relationships remains constant. For example, in the COVID data cases, the number of daily cases would be different but the distance between two counties would remain the same. So, whenever we would need to re-calculate the results, we would need to only recalculate some of the equation, i.e. the equations that are dependent on x. The equations independent of x could be pre-calculated and stored for easy access and retrieval. The equations independent of x in equation 5.8 for G_i^* are equation 5.5 for W_i and equation 5.7 for S_i and their dependent equations. Hence, for each polygon, W_i and S_i remain unchanged for newer values of x and do not need to be recalculated from the beginning.

Next, lets consider a boundary case where we have a new value for only one polygon and there is change in only one value of x. In such a case, the global values of $\overline{X}$ and S_x would change and would need to be updated across all polygons. However, we would only need to recalculate W_{X_i} for cases $w_{i,j} \neq 0, j = k$ where x_k is the existing polygon value and Δx_k is the change in value for x_k.

So the equations become

$$\overline{X}_{new} = \overline{X} + \frac{\Delta x_k}{n} \tag{5.12}$$

$$\overline{X^2}_{new} = \overline{X^2} + \frac{2 * x_k * \Delta x_k + \Delta x_k^2}{n} \tag{5.13}$$

$$\begin{aligned} S_{X}^{2}{}_{new} &= \overline{X^2}_{new} - (\overline{X}_{new})^2 \\ &= \overline{X^2} + \tfrac{2*x_k*\Delta x_k + \Delta x_k^2}{n} - (\overline{X} + \tfrac{\Delta x_k}{n})^2 \\ &= \overline{X^2} + \tfrac{2*x_k*\Delta x_k + \Delta x_k^2}{n} - (\overline{X})^2 - (\tfrac{2*\overline{X}*\Delta x_k}{n}) - (\tfrac{\Delta x_k}{n})^2 \\ &= S_X^2 + \tfrac{2*x_k*\Delta x_k - 2*\overline{X}*\Delta x_k}{n} \end{aligned}$$

$$S_{X}^{2}{}_{new} = S_X^2 + \frac{(2 * \Delta x_k) * (x_k - \overline{X})}{n} \tag{5.14}$$

$$W_{X_i new} = W_{X_i} + w_{i,k}\Delta x_k \tag{5.15}$$

Next, lets consider the general case where there are multiple new x values for multiple polygons. In this case, we would only need to recalculate W_{X_i} for cases $w_{i,j} \neq 0 \ \forall \ j = k$ where x_ks are the updated polygon values. In this case, the equations become:

$$\overline{X}_{new} = \overline{X} + \frac{1}{n}\sum_k \Delta x_k \tag{5.16}$$

$$\overline{X^2}_{new} = \overline{X^2} + \frac{1}{n}\sum_k (2 * x_k * \Delta x_k + \Delta x_k^2) \tag{5.17}$$

$$S_{X}^{2}{}_{new} = S_X^2 + \frac{2}{n} * \sum_k (\Delta x_k * (x_k - \overline{X})) \tag{5.18}$$

$$W_{X_i new} = W_{X_i} + \sum_k w_{i,k} \Delta x_k \tag{5.19}$$

Hence, if there are only few polygons with updated values, and if we have pre-calculated values from previous iterations, then we can calculate the difference and use the difference to reduce a lot of recalculations. For example, if there were only 100s of counties that had updated data from the previous day, then we could rerun calculations for just those 100 and update the G_i^* values. Also, note that Δx_k values can be negative too, in case of decrease in x values.

5.5 Experimental Results

For the experiments, both real world data and simulated/generated data were used to test the implementations.

5.5.1 Real World COVID Data

One of the primary motivation for this work was to track COVID hotspots, especially as they were emerging and altering. One of the main sources of COVID related data was the United States Center for Disease Control. Different geographic level (like cities, districts, county, states) based data on daily reported values are available. This data had necessary COVID related statistics like active cases, new cases, closed cases, deaths, recovered etc. However, for geospatial analysis, we require geographic data too. For the experimental timing results provided in this chapter, we focused on the county level analysis. Geographic data required are county locations and boundaries. This information was available from the Census Bureau's MAF/TIGER geographic database U.S. County Boundaries TIGER dataset [1]. For autocorrelation calculations, we require only certain properties from the geographic data. For each county, we required its boundary information to calculate its centroid. This centroid information was further used to calculate the

inverse distance for the weight values among county polygons. Next, we needed to match the county polygons with its corresponding COVID data. Counties have unique identifiers called GEOID, so each of these county polygons had a unique five digit identifier known as the FIPS code. Also, the county level COVID data along with each county information had a corresponding FIPS code. This common unique id made it easier to join the COVID data with the geographic data. The counties geographic data had 3,233 polygons along with other data entities of which the extra unnecessary information were discarded and this was processed to get a dataset with the following entities: County, State, FIPS, and Centroid. Then for each date, the entities for the available COVID data were: Date, County, State, FIPS, new cases, active cases, recovered cases, total cases, new deaths, and total deaths.

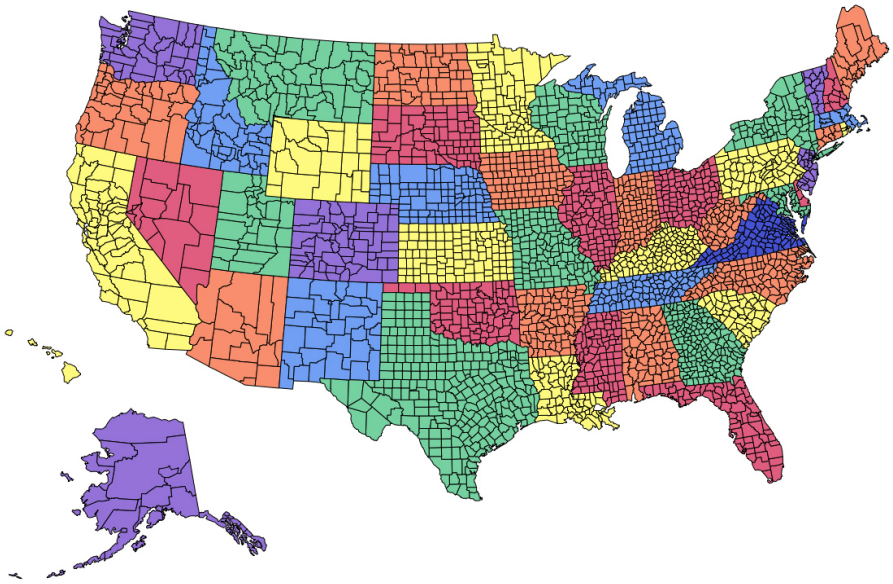

Figure 5.5: Map of US Counties and Boundaries

5.5.2 Simulated/Generated Datasets

Simulated data were generated mostly for the unit point incidence data and the aggregated point incidence data. The data was generated randomly. For the unit point incidence data, the sample space was divided into a uniform square grid,

and each square cell was considered as the polygon for that region. Next, the centroid for each of the square tile was calculated. Then, using different random distributions, x-values (attributes) were assigned to each square tile. The x-values were used to simulate the count of events inside the square tile. Finally the data entities for each square tile were: id, centroid, x1, x2, x3, ..., xn. Using the centroid values to calculate the inverse distances among the square tiles, the weight matrix was populated.

For the aggregated point incidence data, first location for the aggregation points were generated from a uniform random distribution across the sample space. Then a fast Voronoi boundary calculation was used to generate the boundaries for each unit point. These boundaries represented the polygon for that region and the aggregation points were used as centroids for that region. Next, similar to data generation with the square grids, different random distributions were used to simulate x-values which were assigned to each polygon. Finally the data entities for each aggregation points polygon was: id, centroid, x1, x2, x3, ..., xn. Using the centroid values to calculate the inverse distances among the polygon boundaries, the weight matrix was populated.

5.5.3 Hardware Description

Experiments were performed on two machines with the following hardware configurations. Machine 1 (M1) has two Intel Xeon E5 v4 CPUs (2.10 GHz), where each CPU has 18 cores (36 thread). M1 has 500 GBs of RAM. M1 also has an Nvidia TITAN V GPU with 5120 CUDA cores. On the Intel Xeon E5, there is L1 cache of 32KB per core. L2 and L3 cache sizes are 256 KB and 2.5 MB. L2 cache is per core. L3 cache is per NUMA node. The gcc verision is 4.8.5, nvcc is V11.2.67 and pgcc is 21.2.0.

Machine 2 (M2) is a medium sized compute cluster with multiple nodes used

for running experiments with a scheduler. Compute nodes in M2 contains AMD Rome which is a 64 core (128 thread) CPU with a base frequency of 2 GHz, NVIDIA Tesla V100 GPUs which has 5120 CUDA cores at base frequency of 1.20 GHz and 512 GBs RAM. Compute nodes and storage are connected via a 100 GB/s Infiniband network. On the AMD Rome, there is L1 instruction cache of 32KB per core and similarly L1 data cache of 32KB per core. There is mid-level cache (MLC) or L2 of 512 KB per core. AMD Rome has 16 x 16 MB L3 cache which is the last level cache and is a shared cache of 16 MB per 4 core. The gcc version is 9.2.0, mpi is mvapich2, nvcc is V11.2.152 and pgcc is 21.11.0.

5.5.4 Performance Engineering Results

Table 5.1 show the aggregation of speedup gained from different methods from multiple experiments at different data sizes. Every acceleration method improves the computation speedup and combining different approaches has even greater yield. For OpenMP and MPI, the shown speedup holds as long as the threadCount or numProcess is less than the number of cores.

Table 5.1: Parallelization Method and Corresponding Best Speedup (25K dataset)

Parallelization	**Speedup**
GPU CUDA (single node)	100×
GPU OpenACC (single node)	100×
OpenMP (16 thread)	15.4×
AVX2 intrinsics	6×
AVX2 + OpenMP	90×
MPI (16p)	15×
MPI (16p) + AVX2	90×
MPI (8 gpu nodes) + CUDA	750×
MPI (4 gpu nodes) + CUDA	380×

The AVX2 codes were implemented in both Intel and AMD CPUs and the gain in performance was similar across both. Because 8 single precision floating point variables can be loaded in 256 bits of a SIMD register, there is potentially

8-way SIMD parallelism that can be exploited when compared to scalar code. We observe upto 6x speedup using SIMD-optimized code. We used linux perf tool to measure the impact of improved vectorization through intrinsic functions on x86 processors. An analysis through the perf tool showed that with intrinsics the number of CPU cycles were reduced by a factor of almost 40x while the instructions per cycle (IPC) doubled. Higher IPC value represents better CPU utilization. Also, the number of branches decreased by almost 50x while branch misses reduced by 1.5x. This is attributed to the advantages of loop unrolling on line number 8 of Algorithm 1 (loop variable j is incremented by 8). Reduction in branch misses leads to higher instruction level parallelism through instruction pipelining because of reduction in control hazards. Furthermore, cache loads decreased by 16x and cache misses decreased by more than 2x.

From a vectorization perspective, the difference in performance is attributed to the choice of SIMD registers and vector instructions selected by the compiler with/without intrinsics. We used GCC compiler with -O3 flag to enable compiler auto-vectorization. In compiler generated code, XMM registers with 128 bits width were used for critical parts of the kernel. In the version with intrinsics, compiler generated code had YMM registers with 256 bits width. Wider registers have the benefit of packing more data elements in a single register. We looked at the assembly code generated with/without intrinsics using double precision floating point data. For data movement, *vmovsd* was generated in the sub-optimal code instead of *vmovapd*. *s* stands for scalar in vmovsd. *p* stands for packed in *vmovapd*. Similarly, *vmulsd* was generated by compiler in the suboptimal code instead of *vmulpd*.

Figure 5.6 shows the time (in log_2 scale) for different sizes of data. Average from multiple runs of the experiments are shown. The best implementation remains the MPI+CUDA approach.

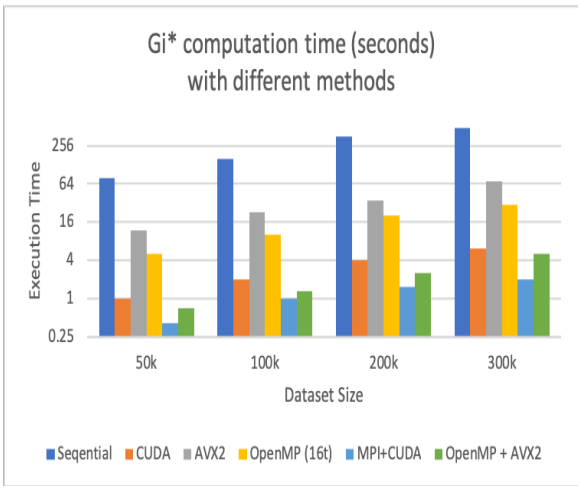

Figure 5.6: Comparison of Gi^* computation Vs data sizes. OpenMP version is running on 16 threads.

Execution times from an experiment with 300,000 polygons are shown in Table 5.3. Using a non-optimized sequential C code, it takes about 36 minutes to run from start to finish. The computationally intensive parts can be divided into three parts. First part is the spatial sorting. Second part is calculating and populating the weight matrix. Final part is calculating all G_i^* values. The above mentioned speedups in Table 5.1 are mostly gained in the second and third parts. OpenACC and CUDA brings down 780 seconds to calculate the weight matrix down to about 9 seconds. AVX2 intrinsics brings it down to almost 110 seconds. Adding OpenMP parallelization to AVX2, with a thread count of 16 threads brings the time down to almost 7 seconds and its performance is very similar to that of MPI. The MPI+CUDA results is using 4 GPUs concurrently which is the fastest. MPI+CUDA took 2 seconds. Table 5.2 shows the average speedup and efficiency of using multiple OpenMP threads.

Table 5.4 shows R-tree based execution time for 300K polygons. This sequential version performs better than the version with spatial sorting because of R-tree data structure. This version does not use spatial sorting, as shown in Table 5.4. OpenMP parallelization speeds up query operations and calculation of

Table 5.2: OpenMP Speedup and Efficiency

Threads	**Avg Speedup**	**Speedup/thread**
2	1.9	0.950
4	3.7	0.925
8	7.7	0.963
16	15.4	0.963
32	30.1	0.941

Table 5.3: Average Execution Times for 300k polygons

Method	**Sorting**	**Wmatrix**	G_i^*	**Total (minutes)**
Sequential	900s	780s	480s	36
CUDA	10s	9s	6s	0.42
AVX2	500s	110s	69s	11.4
OpenMP (16 t)	150s	51s	30s	4
MPI+CUDA	10s	2s	2s	0.24
OpenMP+AVX2	150s	7s	5s	2.7

Table 5.4: Rtree based times for 300k polygons

	Building	**Querying**	G_i^*	**Total (minutes)**
Sequential (No Sort)	20s	60s	520s	10
OpenMP (16 t)	20s	4s	37s	1
OpenMP+AVX2	20s	4s	10s	0.6

G_i^* values compared to the sequential baseline. SIMD parallelization using AVX2 is applied to G_i^* calculations only. The best performance on a single compute node is by using 16 threads accelerated by AVX2 SIMD extensions.

Table 5.5 shows the use of acceleration and rapid recalculation techniques applied to calculate daily G_i^* values for the US Counties using real world COVID data for 500 days to see the evolution of the spread of infection over the time period.

5.6 Conclusion and Future Direction

We have demonstrated successful acceleration of spatial autocorrelation kernel. This acceleration can be used for industrial and scientific application requiring faster solutions and the techniques mentioned in the chapter can be transferred to apply to wide variety of similar statistical kernels. Future directions

Table 5.5: 500 days time series G_i^* calculation for Real US Counties daily COVID data [1] [2]

Method	**Time (minutes)**
Sequential	33
CUDA	0.5
AVX2	6
OpenMP (16t)	3
MPI+CUDA	0.3
OpenMP+AVX2	1

of this work can be extending the rapid recalculation work for streaming and online real-time solutions and expanding the scope of the work for cloud infrastructures where different acceleration techniques are combined to automatically achieve the best acceleration depending on hardware configuration and availability.

CHAPTER 6: CONCLUSION AND FUTURE DIRECTION

As shown in each chapter, this dissertation shows successful acceleration of each of the corresponding computational geometry problem in each chapter using at-least one of the aforementioned acceleration techniques. This dissertation work presented a fine-grained parallel algorithm targeted to GPU architecture for a non-trivial computational geometry code. It 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. The experiments and exploration in directives-based parallelization of Fotune's algorithm has yielded a shared memory implementation that gives around 2x speedup compared to the sequential version. Directives based parallelization of segment tree construction, query have been achieved along with performance improvements due to parallelization in terms of speedup is upto 29x speedup for tree construction and upto 23x speedup for batch querying using 32 threads using OpenMP compared to single-threaded version. Using OpenACC, speedup upto 100x during the tree construction phase and speedup upto 62x during the batch query phase have been achieved. The cache optimized version of the segment tree is 1.3x faster on average and the cache miss rate is reduced by almost 25%. This dissertation has also demonstrated successful acceleration of spatial autocorrelation kernel. CPU/GPU optimizations gain upto 750X relative speedup with a 8 GPU setup when compared to baseline sequential implementation. Compared to the best implementation using OpenMP + R-tree data structure on a single compute node, our accelerated hotspots benchmark gains a 25X speedup. For real world US counties and COVID data evolution calculated over 500 days, we gain upto 110X speedup reducing time from 33 minutes to 0.3 minutes.

Similar to how general purpose GPU computing accelerated the computational possibility of all the computational geometry problems mentioned in the previous chapters, a new class of hardware has been recently introduced - Data Processing Units (DPUs). DPUs could be used as a stand-alone embedded processor or incorporated with a network interface controller (SmartNIC). For high volume data processing, DPUs could be the intermediary that could save a lot of memory and communication related overheads, especially by freeing the CPU from it. Furthermore in case of spatial data, DPUs could be used to filter, grid or partition data, leaving only the computation to the processors. DPUs could be the next step in accelerating extreme scale computing but due to its novelty a lot of research in this area is warranted.

So, from a broader perspective, the future of this research work would also include working on exploring the capabilities of DPU and how that can be applied to acceleration of computational geometry algorithms for high performance computing based geo-spatial big data analysis.

BIBLIOGRAPHY

[1] https://www.census.gov/geographies/mapping-files/time-series/geo/tiger-line-file.html.

[2] https://covid.cdc.gov/covid-data-tracker/index.html.

[3] https://pro.arcgis.com/en/pro-app/latest/tool-reference/spatial-statistics/h-how-hot-spot-analysis-getis-ord-gi-spatial-stati.htm.

[4] C. E. Leiserson, N. C. Thompson, J. S. Emer, B. C. Kuszmaul, B. W. Lampson, D. Sanchez, and T. B. Schardl, "There's plenty of room at the top: What will drive computer performance after moore's law?" *Science*, vol. 368, no. 6495, p. eaam9744, 2020.

[5] E. Carson, J. Demmel, L. Grigori, N. Knight, P. Koanantakool, O. Schwartz, and H. V. Simhadri, "Write-avoiding algorithms." IEEE, May 2016, pp. 648–658. [Online]. Available: https://ieeexplore.ieee.org/document/7516061

[6] J. Demmel, D. Eliahu, A. Fox, S. Kamil, B. Lipshitz, O. Schwartz, and O. Spillinger, "Communication-optimal parallel recursive rectangular matrix multiplication." IEEE, May 2013, pp. 261–272. [Online]. Available: https://ieeexplore.ieee.org/document/6569817

[7] G. Ballard, A. Buluc, J. Demmel, L. Grigori, B. Lipshitz, O. Schwartz, and S. Toledo, "Communication optimal parallel multiplication of sparse random matrices," ser. SPAA '13. ACM, Jul 23, 2013, pp. 222–231. [Online]. Available: http://dl.acm.org/citation.cfm?id=2486196

[8] J. L. Bentley and T. A. Ottmann, "Algorithms for reporting and counting geometric intersections," *IEEE Transactions on computers*, no. 9, pp. 643–647, 1979.

[9] M. T. Goodrich, "Intersecting line segments in parallel with an output-sensitive number of processors," *SIAM Journal on Computing*, vol. 20, no. 4, pp. 737–755, 1991.

[10] M. J. Atallah and M. T. Goodrich, "Efficient plane sweeping in parallel," in *Proceedings of the second annual symposium on Computational geometry*. ACM, 1986, pp. 216–225.

[11] M. T. Goodrich, M. R. Ghouse, and J. Bright, "Sweep methods for parallel computational geometry," *Algorithmica*, vol. 15, no. 2, pp. 126–153, 1996.

[12] S. Puri and S. K. Prasad, "Output-sensitive parallel algorithm for polygon clipping," in *43rd International Conference on Parallel Processing, ICPP 2014, Minneapolis, MN, USA, September 9-12, 2014*, 2014, pp. 241–250. [Online]. Available: https://doi.org/10.1109/ICPP.2014.33

[13] S. Audet, C. Albertsson, M. Murase, and A. Asahara, "Robust and efficient polygon overlay on parallel stream processors," in *Proceedings of the 21st ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*. ACM, 2013, pp. 304–313.

[14] W. R. Franklin, C. Narayanaswami, M. Kankanhalli, D. Sun, M.-C. Zhou, and P. Y. Wu, "Uniform grids: A technique for intersection detection on serial and parallel machines," in *Proceedings of Auto Carto*, vol. 9. Citeseer, 1989, pp. 100–109.

[15] D. Aghajarian and S. K. Prasad, "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*. ACM, 2017, p. 56.

[16] M. McKenney and T. McGuire, "A parallel plane sweep algorithm for multi-core systems," in *Proceedings of the 17th ACM SIGSPATIAL international conference on advances in geographic information systems*. ACM, 2009, pp. 392–395.

[17] A. B. Khlopotine, V. Jandhyala, and D. Kirkpatrick, "A variant of parallel plane sweep algorithm for multicore systems," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 32, no. 6, pp. 966–970, 2013.

[18] M. McKenney and et al., "Multi-core parallelism for plane sweep algorithms as a foundation for gis operations," in *Geoinformatica*. Springer, 2017, p. 151–174.

[19] A. Biniaz and G. Dastghaibyfard, "A faster circle-sweep delaunay triangulation algorithm," *Advances in Engineering Software*, vol. 43, no. 1, pp. 1–13, 2012. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0965997811002420

[20] K. Wong and H. A. Muller, "An efficient implementation of fortune's plane-sweep algorithm for voronoi diagrams," 1991.

[21] E. F. Bollig, "Centroidal voronoi tessellation of manifolds using the gpu," Ph.D. dissertation, 1982. [Online]. Available: http://diginole.lib.fsu.edu/etd/3606

[22] I. Majdandzic, C. Trefftz, and G. Wolffe, "Computation of voronoi diagrams using a graphics processing unit." IEEE, 2008, pp. 437–441. [Online]. Available: https://ieeexplore.ieee.org/document/4554342

[23] Z. Yuan, G. Rong, X. Guo, and W. Wang, "Generalized voronoi diagram computation on gpu." IEEE, 2011, pp. 75–82. [Online]. Available: https://ieeexplore.ieee.org/document/5988951

[24] G. Rong, Y. Liu, W. Wang, X. Yin, X. D. Gu, and X. Guo, "Gpu-assisted computation of centroidal voronoi tessellation," *IEEE Transactions on Visualization and Computer Graphics*, vol. 17, no. 3, pp. 345–356, 2011. [Online]. Available: https://ieeexplore.ieee.org/document/5438988

[25] A. Tsidaev, "Parallel algorithm for natural neighbor interpolation," 2016.

[26] J. Nievergelt and F. Preparata, "Plane-sweep algorithms for intersecting geometric figures," pp. 739–747, Oct 1, 1982. [Online]. Available: http://dl.acm.org/citation.cfm?id=358681

[27] S. G. Akl and K. A. Lyons, *Parallel computational geometry.* Prentice-Hall, Inc., 1993.

[28] M. d. Berg, O. Cheong, M. v. Kreveld, and M. Overmars, *Computational geometry: algorithms and applications.* Springer-Verlag TELOS, 2008.

[29] A. Aggarwal, B. Chazelle, L. Guibas, C. Ó'Dúnlaing, and C. Yap, "Parallel computational geometry," *Algorithmica*, vol. 3, no. 1-4, pp. 293–327, 1988.

[30] G. Blankenagel and R. H. Güting, "External segment trees," *Algorithmica*, vol. 12, no. 6, pp. 498–532, 1994.

[31] L. Arge, "External-memory algorithms with applications in gis," in *Advanced School on the Algorithmic Foundations of Geographic Information Systems.* Springer, 1996, pp. 213–254.

[32] F. Dehne and A. Rau-Chaplin, "Implementing data structures on a hypercube multiprocessor, and applications in parallel computational geometry," in *International Workshop on Graph-Theoretic Concepts in Computer Science.* Springer, 1989, pp. 316–329.

[33] A. Chan, F. Dehne, and A. Rau-Chaplin, "Coarse-grained parallel geometric search," *Journal of Parallel and Distributed Computing*, vol. 57, no. 2, pp. 224 – 235, 1999. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0743731598915271

[34] P. Su and R. L. S. Drysdale, "Building segment trees in parallel," Dartmouth College, Computer Science, Tech. Rep., 1992. [Online]. Available: http://www.cs.dartmouth.edu/reports/TR92-184.pdf

[35] A. V. Gerbessiotis, "An architecture independent study of parallel segment trees," *Journal of Discrete Algorithms*, vol. 4, no. 1, pp. 1–24, 2006.

[36] C. Zheng, G. Shen, S. Li, and S. Shenker, "Distributed segment tree: Support of range query and cover query over dht." in *IPTPS*, 2006.

[37] S. Guobin, Z. Changxi, P. Wei, and L. Shipeng, "Distributed segment tree: A unified architecture to support range query and cover query," Tech. Rep., Mar 2007.

[38] A. LaMarca and R. Ladner, "The influence of caches on the performance of heaps," *Journal of Experimental Algorithmics (JEA)*, vol. 1, pp. 4–es, 1996.

[39] P.-H. Kamp, "You're doing it wrong," *Communications of the ACM*, vol. 53, no. 7, pp. 55–59, 2010.

[40] L. Arge, D. E. Vengroff, and J. S. Vitter, "External-memory algorithms for processing line segments in geographic information systems," *Algorithmica*, vol. 47, no. 1, pp. 1–25, 2007.

[41] K. Berney, H. Casanova, A. Higuchi, B. Karsin, and N. Sitchinava, "Beyond binary search: parallel in-place construction of implicit search tree layouts," in *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2018, pp. 1070–1079.

[42] L. Arge, M. de Berg, and H. Haverkort, "Cache-oblivious r-trees," in *Proceedings of the twenty-first annual symposium on Computational geometry*, 2005, pp. 170–179.

[43] E. D. Demaine, "Cache-oblivious algorithms and data structures," *Lecture Notes from the EEF Summer School on Massive Data Sets*, vol. 8, no. 4, pp. 1–249, 2002.

[44] M. A. Awad, S. Ashkiani, R. Johnson, M. Farach-Colton, and J. D. Owens, "Engineering a high-performance gpu b-tree," in *Proceedings of the 24th symposium on principles and practice of parallel programming*, 2019, pp. 145–157.

[45] T. Foley and J. Sugerman, "Kd-tree acceleration structures for a gpu raytracer," in *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, 2005, pp. 15–22.

[46] S. K. Prasad, M. McDermott, X. He, and S. Puri, "Gpu-based parallel r-tree construction and querying." IEEE, 05/2015, pp. 618–627. [Online]. Available: https://ieeexplore.ieee.org/document/7284367

[47] M. K. Maramreddy and K. Kothapalli, "Gpu accelerated range trees with applications," in *European Conference on Parallel Processing*. Springer, 2014, pp. 740–751.

[48] W. R. Tobler, "A computer movie simulating urban growth in the detroit region," *Economic geography*, vol. 46, no. sup1, pp. 234–240, 1970.

[49] S. Shekhar, P. Zhang, and Y. Huang, "Spatial data mining," in *Data mining and knowledge discovery handbook*. Springer, 2009, pp. 837–854.

[50] Y. Liu, K. Wu, S. Wang, Y. Zhao, and Q. Huang, "A mapreduce approach to gi(d) spatial statistic," in *Proceedings of the ACM SIGSPATIAL International Workshop on High Performance and Distributed Geographic Information Systems*, 2010, pp. 11–18.

[51] P. Mehta, C. Windolf, and A. Voisard, "Spatio-temporal hotspot computation on apache spark (gis cup)," in *24th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, 2016.

[52] S. Peng, H. Wei, H. Li, and H. Samet, "Simplification and refinement for speedy spatio-temporal hot spot detection using spark (gis cup)," in *24th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, 2016.

[53] M. Li, "MS Thesis: A Parallel Algorithm and Implementation to Compute Spatial Autocorrelation (Hotspot) Using MATLAB," *MS Thesis*, 2020.

[54] Y. Liu and S. Puri, "Efficient filters for geometric intersection computations using gpu," in *Proceedings of the 28th International Conference on Advances in Geographic Information Systems*, 2020, pp. 487–496.

[55] Y. Liu, J. Yang, and S. Puri, "Hierarchical filter and refinement system over large polygonal datasets on cpu-gpu," in *2019 IEEE 26th International Conference on High Performance Computing, Data, and Analytics (HiPC)*. IEEE, 2019, pp. 141–151.

[56] J. Yang and S. Puri, "Efficient parallel and adaptive partitioning for load-balancing in spatial join," in *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2020, pp. 810–820.

[57] H.-J. Lee, J. Robertson, and J. Fortes, "Generalized cannon's algorithm for parallel matrix multiplication," ser. ICS '97. ACM, Jul 11, 1997, pp. 44–51. [Online]. Available: http://dl.acm.org/citation.cfm?id=263591

[58] E. Solomonik, D. Matthews, J. R. Hammond, and J. Demmel, "Cyclops tensor framework: Reducing communication and eliminating load imbalance in massively parallel contractions." IEEE, May 2013, pp. 813–824. [Online]. Available: https://ieeexplore.ieee.org/document/6569864

[59] R. C. Agarwal, S. M. Balle, F. G. Gustavson, M. Joshi, and P. Palkar, "A three-dimensional approach to parallel matrix multiplication," *IBM Journal of Research and Development*, vol. 39, no. 5, pp. 575–582, Sep 1995.

[60] J. Choi, D. W. Walker, and J. J. Dongarra, "Pumma: Parallel universal matrix multiplication algorithms on distributed memory concurrent computers," *Concurrency: Practice and Experience*, vol. 6, no. 7, pp. 543–570, Oct 1994. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.4330060702

[61] W. F. McColl, W. F. McColl, A. Tiskin, and A. Tiskin, "Memory-efficient matrix multiplication in the bsp model," *Algorithmica*, vol. 24, no. 3, pp. 287–297, Jul 1999.

[62] C. P. Kruskal, L. Rudolph, and M. Snir, "Techniques for parallel manipulation of sparse matrices," *Theoretical Computer Science*, vol. 64, no. 2, pp. 135–157, 1989. [Online]. Available: http://dx.doi.org/10.1016/0304-3975(89)90058-3

[63] A. Buluc and J. R. Gilbert, "Challenges and advances in parallel sparse matrix-matrix multiplication." IEEE, Sep 2008, pp. 503–510. [Online]. Available: https://ieeexplore.ieee.org/document/4625887

[64] G. Ballard, J. Demmel, O. Holtz, and O. Schwartz, "Minimizing communication in numerical linear algebra," *SIAM Journal on Matrix Analysis and Applications*, vol. 32, no. 3, pp. 866–901, Jul 2011. [Online]. Available: https://search.proquest.com/docview/1372742712

[65] E. Solomonik, A. Bhatele, and J. Demmel, "Improving communication performance in dense linear algebra via topology aware collectives," ser. SC '11. ACM, Nov 12, 2011, pp. 1–11. [Online]. Available: http://dl.acm.org/citation.cfm?id=2063487

[66] B. Lipshitz, G. Ballard, J. Demmel, and O. Schwartz, "Communication-avoiding parallel strassen," ser. SC '12. IEEE Computer Society Press, Nov 10, 2012, pp. 1–11. [Online]. Available: http://dl.acm.org/citation.cfm?id=2389133

[67] E. Solomonik and J. Demmel, *Communication-Optimal Parallel 2.5D Matrix Multiplication and LU Factorization Algorithms*, ser. Euro-Par 2011 Parallel Processing. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, vol. 6853, pp. 90–109.

[68] G. Ballard, J. Demmel, O. Holtz, B. Lipshitz, and O. Schwartz, "Communication-optimal parallel algorithm for strassen's matrix multiplication," ser. SPAA '12. ACM, Jun 25, 2012, pp. 193–204. [Online]. Available: http://dl.acm.org/citation.cfm?id=2312044

[69] D. Irony, S. Toledo, and A. Tiskin, "Communication lower bounds for distributed-memory matrix multiplication," *Journal of Parallel and Distributed Computing*, vol. 64, no. 9, pp. 1017–1026, Sep 2004. [Online]. Available: http://dx.doi.org/10.1016/j.jpdc.2004.03.021

[70] R. A. Van De Geijn and J. Watts, "Summa: scalable universal matrix multiplication algorithm," *Concurrency: Practice and Experience*, vol. 9, no. 4, pp. 255–274, Apr 1997. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/(SICI)1096-9128(199704)9:4⟨255::AID-CPE250⟩3.0.CO;2-2

[71] J. Yang, A. Paudel, and S. Puri, "Spatial data decomposition and load balancing on hpc platforms," ser. PEARC '19. ACM, Jul 28, 2019, pp. 1–4. [Online]. Available: http://dl.acm.org/citation.cfm?id=3333266

[72] S. Puri, A. Paudel, and S. Prasad, "Mpi-vector-io," ser. ICPP 2018. ACM, Aug 13, 2018, pp. 1–11. [Online]. Available: http://dl.acm.org/citation.cfm?id=3225105

[73] J.-W. Hong and H. T. Kung, "I/o complexity: The red-blue pebble game," Tech. Rep., Mar 1981. [Online]. Available: http://www.dtic.mil/docs/citations/ADA104739

[74] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran, "Cache-oblivious algorithms," *ACM Transactions on Algorithms (TALG)*, vol. 8, no. 1, pp. 1–22, Jan 1, 2012. [Online]. Available: http://dl.acm.org/citation.cfm?id=2071383

[75] G. Blelloch, Y. Gu, J. Shun, and Y. Sun, "Parallel write-efficient algorithms and data structures for computational geometry," ser. SPAA '18. ACM, Jul 11, 2018, pp. 235–246. [Online]. Available: http://dl.acm.org/citation.cfm?id=3210380

[76] Y. Sun and G. Blelloch, "Implementing parallel and concurrent tree structures," ser. PPoPP '19. ACM, Feb 16, 2019, pp. 447–450. [Online]. Available: http://dl.acm.org/citation.cfm?id=3302576

[77] G. Kwasniewski, M. Kabić, M. Besta, J. VandeVondele, R. Solcà, and T. Hoefler, "Red-blue pebbling revisited: near optimal parallel matrix-matrix multiplication," Aug 26, 2019. [Online]. Available: https://arxiv.org/abs/1908.09606

[78] G. Blelloch, P. Gibbons, and H. Simhadri, "Low depth cache-oblivious algorithms," ser. SPAA '10. ACM, Jun 13, 2010, pp. 189–199. [Online]. Available: http://dl.acm.org/citation.cfm?id=1810519

[79] M. Frigo and V. Strumpen, "The cache complexity of multithreaded cache oblivious algorithms," *Theory of Computing Systems*, vol. 45, no. 2, pp. 203–233, Aug 1, 2009. [Online]. Available: https://search.proquest.com/docview/237221492

[80] Y.-J. Chiang, "Experiments on the practical i/o efficiency of geometric algorithms: Distribution sweep versus plane sweep," *Computational Geometry: Theory and Applications*, vol. 9, no. 4, pp. 211–236, 1998. [Online]. Available: http://dx.doi.org/10.1016/S0925-7721(97)00020-5

[81] N. Sitchinava, "Computational geometry in the parallel external memory model," *SIGSPATIAL Special*, vol. 4, no. 2, pp. 18–23, Jul 1, 2012. [Online]. Available: http://dl.acm.org/citation.cfm?id=2367578

[82] D. Ajwani and N. Sitchinava, "Empirical evaluation of the parallel distribution sweeping framework on multicore architectures," Jun 19, 2013. [Online]. Available: https://arxiv.org/abs/1306.4521

[83] L. Arge, M. Goodrich, M. Nelson, and N. Sitchinava, "Fundamental parallel algorithms for private-cache chip multiprocessors," ser. SPAA '08. ACM, Jun 14, 2008, pp. 197–206. [Online]. Available: http://dl.acm.org/citation.cfm?id=1378573

[84] R. Sharathkumar, M. T. C. Vinaykumar, P. Maheshwari, and P. Gupta, "Efficient external memory segment intersection for processing very large vlsi layouts." IEEE, 2005, pp. 740–743 Vol. 1. [Online]. Available: https://ieeexplore.ieee.org/document/1594207

[85] D. Ajwani, N. Sitchinava, and N. Zeh, "I/o-optimal distribution sweeping on private-cache chip multiprocessors." IEEE, May 2011, pp. 1114–1123. [Online]. Available: https://ieeexplore.ieee.org/document/6012918

[86] P. Agarwal, L. Arge, T. Mølhave, and B. Sadri, "I/o-efficient efficient algorithms for computing contours on a terrain," ser. SCG '08. ACM, Jun 9, 2008, pp. 129–138. [Online]. Available: http://dl.acm.org/citation.cfm?id=1377698

[87] F. Dehne, S. Mardegan, A. Pietracaprina, and G. Prencipe, "Distribution sweeping on clustered machines with hierarchical memories." IEEE, 2002, p. 6 pp. [Online]. Available: https://ieeexplore.ieee.org/document/1015508

[88] T. M. Chan and E. Y. Chen, "Optimal in-place and cache-oblivious algorithms for 3-d convex hulls and 2-d segment intersection," *Computational Geometry: Theory and Applications*, vol. 43, no. 8, pp. 636–646, 2010. [Online]. Available: http://dx.doi.org/10.1016/j.comgeo.2010.04.005

[89] L. Arge, T. Mølhave, and N. Zeh, *Cache-Oblivious Red-Blue Line Segment Intersection*, ser. Algorithms - ESA 2008. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, vol. 5193, pp. 88–99.

[90] L. A. Arge, "External-memory algorithms with applications in gis," Jan 1, 1997.

[91] P. Agarwal, L. Arge, A. Danner, and B. Holland-Minkley, "Cache-oblivious data structures for orthogonal range searching," ser. SCG '03. ACM, Jun 8, 2003, pp. 237–245. [Online]. Available: http://dl.acm.org/citation.cfm?id=777828

[92] M. T. Goodrich, J.-J. Tsay, D. E. Vengroff, and J. S. Vitter, "External-memory computational geometry." IEEE, 1993, pp. 714–723. [Online]. Available: https://ieeexplore.ieee.org/document/366816

[93] S. Puri and S. K. Prasad, "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)*, vol. 00, May 2015, pp. 576–585. [Online]. Available: doi.ieeecomputersociety.org/10.1109/CCGrid.2015.43

[94] S. Prasad, D. Aghajarian, M. McDermott, D. Shah, M. Mokbel, S. Puri, S. Rey, S. Shekhar, Y. Xe, R. Vatsavai, F. Wang, Y. Liang, H. Vo, and S. Wang, "Parallel Processing Over Spatial-Temporal Datasets From Geo, Bio, Climate And Social Science Communities: A Research Roadmap," *6th IEEE International Congress on Big Data, Hawaii*, 2017.

[95] D. Agarwal, S. Puri, X. He, and S. K. Prasad, "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, May 21-25, 2012*, 2012, pp. 1433–1439. [Online]. Available: https://doi.org/10.1109/IPDPSW.2012.180

[96] M. McKenney, G. De Luna, S. Hill, and L. Lowell, "Geospatial overlay computation on the gpu," in *Proceedings of the 19th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*. ACM, 2011, pp. 473–476.

[97] OSM. (2017) OpenStreet Map Data. [Online]. Available: http://spatialhadoop.cs.umn.edu/datasets.html

[98] S. Puri, A. Paudel, and S. K. Prasad, "MPI-Vector-IO: Parallel I/O and Partitioning for Geospatial Vector Data," in *Proceedings of the 47th International Conference on Parallel Processing*, ser. ICPP 2018. New York, NY, USA: ACM, 2018, pp. 13:1–13:11. [Online]. Available: http://doi.acm.org/10.1145/3225058.3225105

[99] CGAL. The Computational Geometry Algorithms Library. [Online]. Available: https://www.cgal.org

[100] S. You, J. Zhang, and L. Gruenwald, "High-performance polyline intersection based spatial join on gpu-accelerated clusters," in *Proceedings of the 5th ACM SIGSPATIAL International Workshop on Analytics for Big Geospatial Data*. ACM, 2016, pp. 42–49.

[101] S. P. Satish Puri, Danial Aghajarian. Mpi-gis : High performance computing and i/o for spatial overlay and join" (research poster). [Online]. Available: http://sc16.supercomputing.org/sc-archive/tech_poster/tech_poster_pages/post241.html

[102] S. Puri, D. Agarwal, and S. K. Prasad, "Polygonal Overlay Computation on Cloud, Hadoop, and MPI," *Encyclopedia of GIS.*, pp. 1598–1606, 2017.

[103] D. Aghajarian, S. Puri, and S. K. Prasad, "GCMF: an efficient end-to-end spatial join system over large polygonal datasets on GPGPU platform," *Proceedings of the 24th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems, 2016*, 2016.

[104] F. P. Preparata and M. I. Shamos, *Computational geometry: an introduction*. Springer Science & Business Media, 2012.

[105] A. Paudel and S. Puri, "Openacc based gpu parallelization of plane sweep algorithm for geometric intersection," in *International Workshop on Accelerator Programming Using Directives*. Springer, 2018, pp. 114–135.

[106] (2002). [Online]. Available: https://www.cs.hmc.edu/~mbrubeck/voronoi.html

[107] S. Fortune, "A sweepline algorithm for voronoi diagrams," pp. 153–174, Nov 1987.

[108] (2006). [Online]. Available: http://www.ams.org/publicoutreach/feature-column/fcarc-voronoi

[109] C. Rüb, "Line-segment intersection reporting in parallel," *Algorithmica*, vol. 8, no. 1-6, pp. 119–144, 1992.

[110] L. Becker, A. Giesen, K. H. Hinrichs, and J. Vahrenhold, "Algorithms for performing polygonal map overlay and spatial join on massive data sets," in *International Symposium on Spatial Databases*. Springer, 1999, pp. 270–285.

[111] A. Eldawy and M. F. Mokbel, "SpatialHadoop: A MapReduce Framework for Spatial Data," in *31st IEEE International Conference on Data Engineering, ICDE 2015, Seoul, South Korea, April 13-17, 2015*, 2015, pp. 1352–1363. [Online]. Available: http://dx.doi.org/10.1109/ICDE.2015.7113382

[112] S. Chandrasekaran and G. Juckeland, *OpenACC for programmers*. Boston ; Columbus ; Indianapolis [und 21 weitere]: Addison-Wesley, 2018.

[113] R. van der Pas, E. Stotzer, and C. Terboven, *Using OpenMP – The Next Step*. Cambridge: MIT Press, 2017. [Online]. Available: https://ieeexplore.ieee.org/servlet/opac?bknumber=8169743

[114] F. P. Preparata and M. I. Shamos, *Computational geometry*, 3rd ed. New York u.a: Springer, 1990.

[115] S. G. Akl and K. A. Lyons, *Parallel computational geometry*. Englewood Cliffs, NJ: Prentice Hall, 1993.

[116] M. T. Goodrich, "Intersecting line segments in parallel with an output-sensitive number of processors," *SIAM Journal on Computing*, vol. 20, no. 4, pp. 737–755, 1991.

[117] B. R. Vatti, "A generic solution to polygon clipping," *Communications of the ACM*, vol. 35, no. 7, pp. 56–63, 1992.

[118] S. G. Akl, *Parallel computation: models and methods*. Prentice Hall Upper Saddle River, 1997, vol. 4.

[119] B. Chazelle, "Reporting and counting segment intersections," *Journal of Computer and System Sciences*, vol. 32, no. 2, pp. 156–182, 1986.

[120] M. A. Bender, E. D. Demaine, and M. Farach-Colton, "Cache-oblivious b-trees," *SIAM Journal on Computing*, vol. 35, no. 2, pp. 341–358, 2005.

[121] A. Paudel, J. Yang, and S. Puri, "Parallelization of plane sweep based voronoi construction with compiler directives," in *2019 IEEE 43rd Annual Computer Software and Applications Conference (COMPSAC)*, vol. 1. IEEE, 2019.

[122] S. D. Stoller, M. Carbin, S. Adve, K. Agrawal, G. Blelloch, D. Stanzione, K. Yelick, and M. Zaharia, "Future directions for parallel and distributed computing: Spx 2019 workshop report," *NSF Workshop Reports*, Oct 2019. [Online]. Available: http://par.nsf.gov/biblio/10127824

[123] J. K. Ord and A. Getis, "Local spatial autocorrelation statistics: distributional issues and an application," *Geographical analysis*, vol. 27, no. 4, pp. 286–306, 1995.

[124] https://developer.nvidia.com/blog/thinking-parallel-part-iii-tree-construction-gpu.

[125] https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html.

[126] M. P. Armstrong and R. Marciano, "Massively parallel processing of spatial statistics," *International Journal of Geographical Information Systems*, vol. 9, no. 2, pp. 169–189, 1995.

[127] M. P. Armstrong, C. E. Pavlik, and R. Marciano, "Parallel processing of spatial statistics," *Computers & Geosciences*, vol. 20, no. 2, pp. 91–104, 1994.

[128] http://resources.esri.com/help/9.3/ArcGISEngine/java/Gp_ToolRef/Spatial_Statistics_tools/how_hot_spot_analysis_colon_getis_ord_gi_star_spatial_statistics_works.htm.

[129] S.-l. Zhang and K. Zhang, "Comparison between general moran's index and getis-ord general g of spatial autocorrelation," *Acta Scientiarum Naturalium Universitatis Sunyatseni*, vol. 4, p. 022, 2007.

[130] W. Gropp, T. Hoefler, R. Thakur, and E. Lusk, *Using advanced MPI: Modern features of the message-passing interface*. MIT Press, 2014.

[131] CGAL. Spatial Sorting in CGAL. [Online]. Available: https://doc.cgal.org/latest/Spatial_sorting/index.html

[132] P. Mohan, R. E. Wilson, S. Shekhar, B. George, N. Levine, and M. Celik, "Should sdbms support a join index? a case study from crimestat," in *Proceedings of the 16th ACM SIGSPATIAL international conference on Advances in geographic information systems*, 2008, pp. 1–10.