MRSweep: Distributed In-Memory Sweep-line for Scalable Object Intersection Problems

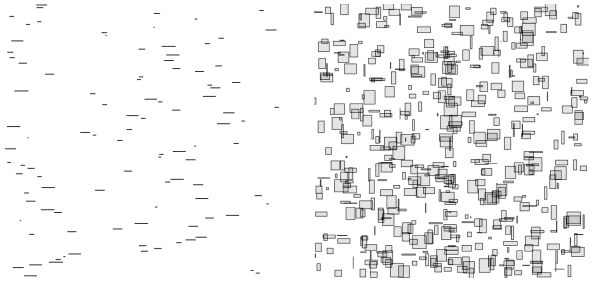
Tilemachos Pechlivanoglou, Mahmoud Alsaeed and Manos Papagelis
Lassonde School of Engineering, York University, Toronto, Canada
tipech@eecs.yorku.ca, mahmoud2@eecs.yorku.ca, papaggel@eecs.yorku.ca

***Abstract*—Several data mining and machine learning problems can be reduced to the computational geometry problem of finding intersections of a set of geometric objects, such as intersections of line segments or rectangles/boxes. Currently, the state-of-the-art approach for addressing such intersection problems in Euclidean space is collectively known as the *sweep-line* or *plane sweep* algorithm, and has been utilized in a variety of application domains, including databases, gaming and transportation, to name a few. The idea behind sweep line is to employ a conceptual line that is swept or moved across the plane, stopping at intersection points. However, to report all K intersections among any N objects, the standard sweep line algorithm (based on the Bentley-Ottmann algorithm) has a time complexity of $O((N + K)logN)$, therefore cannot scale to very large number of objects and cases where there are many intersections. In this paper, we propose MRSWEEP and MRSWEEP-D, two sophisticated and highly scalable algorithms for the parallelization of sweep-line and its variants. We provide algorithmic details of fully distributed in-memory versions of the proposed algorithms using the MapReduce programming paradigm in the Apache Spark cluster environment. A theoretical analysis of the proposed algorithms is presented, as well as a thorough experimental evaluation that provides evidence of the algorithms' scalability in varying levels of problem complexity. We make source code and datasets available to support the reproducibility of the results.**

***Index Terms*—parallel and distributed data mining, big data analytics, computational geometry, intersection problem, overlaps, sweep-line**

I. INTRODUCTION

Several data mining and machine learning problems employ the abstraction of *axis-aligned geometric objects* in some dimension D to represent *temporal*, *spatial* or other semantic information of events or entities. An axis-aligned object (also known as axis-parallel or axis-oriented) is an object in D-dimensional space whose shape is aligned with the coordinate axes of the space, such as *line segments* in 1-*D*, *rectangles* or *boxes* in 2-*D*, *cuboids* in 3-*D*, and so forth (see Fig. 1). For example, an event typically has a starting time and an ending time and can be represented as a 1-*D* line segment; the daily pickup and drop-off locations of a taxi in New York City (NYC) can be represented as a 2-*D* rectangle or box covering all locations visited that day; the trajectories of a drone can be modeled as a 3-*D* cuboid that contains its trajectories. Now, consider the case where given a large set of axis-aligned geometric objects in dimension D, we want to perform an analysis of the *relationships* between these objects. By a relationship we refer to questions of whether two objects intersect each other, and if yes, what is *the size of their intersection* or their *overlap*. Following the aforementioned motivating examples, we might be interested to find overlaps between events, which areas are better served by taxis in NYC, or areas of possible collision of drones, respectively. All these problems can effectively be reduced to the computational geometry problem of finding intersections of a set of geometric objects and computing their pair-wise overlap.



(a) Sample line-segments (b) Sample rectangles

Fig. 1: Axis-aligned geometric objects in 1-D (*line-segments*), 2-D (*rectangles*) and higher dimensions are commonly utilized to represent temporal, spatial or other semantic information of events or entities in diverse data mining applications. Exploring the relationship of millions of these objects in Euclidean space, requires scalable algorithms for determining their *intersection* and *size of overlap*.

Methods that address this problem have found application in various domains, including spatial databases [1], graph mining [2], VLSI physical design [3], computer graphics and crowd simulation collision detection [4]. The naive approach to this problem is to simply compare every object to every other in a brute-force fashion, a process that is prohibitively expensive and inefficient. Instead, more efficient, well-established methods exist that utilize what is collectively known as the *sweep-line* or *plane sweep* algorithm. This is an algorithmic paradigm that employs a conceptual sweep line to solve various intersection problems in Euclidean space [5]. While this paradigm has been successfully used in a number of problems in diverse domains and applications [6], it is a particularly resource intensive algorithm and therefore its application is limited to smaller and/or sparser data sets. The main limitation of the algorithm can be attributed to its usage of memory space, especially for cases where the dataset is large (i.e., large

978-1-7281-8206-3/20/$31.00 ©2020 IEEE
DOI 10.1109/DSAA49011.2020.00046

number of objects) and there is a large number of pair-wise intersections. Such data characteristics will effectively render the algorithm impractical. There have been some attempts to address the memory space limitation using specialized data structures and techniques but these result in an increase of the runtime complexity of the algorithm [7], [8].

Another approach is to develop variants of the sweep-line algorithm that can carry out computations simultaneously using principles of *parallel computing*. Conceptually, the sweep-line algorithm can be easily parallelized following a divide and conquer algorithm design paradigm. A divide-and-conquer algorithm works by recursively breaking down a problem into sub-problems, until they become simple enough to be solved independently. The outcome of the sub-problems can then be combined to give a solution to the original problem. As such, several well-established solutions have been proposed for the parallelization of the sweep-line algorithm, usually by statically or dynamically segmenting incoming data and performing the sweep-line operation in parallel for each segment [9]–[12]. However, these methods either introduce synchronization checkpoints that slow down execution or have been developed specifically for *single-node multi-core* architectures with *shared memory* and *shared data storage*. As such, they cannot accommodate larger instances of the problem. Furthermore, most of the current state-of-the-art implementations assume that the input data are already sorted, an assumption that is not necessarily true; however, the sorting operation is expensive and can introduce significant additional processing time if not properly parallelized. Finally, as most prior research deals with adaptations of the sweep-line algorithmic paradigm for domain-specific problems, the sensitivity of those implementations to different data characteristics and other parameters has not been properly studied.

Motivated by these limitations, our main objective is to design and develop methods that are adequate for much larger instances of the problem, by taking advantage of a distributed processing environment, including *a large cluster with multiple nodes*, *distributed storage* and *distributed memory*. Towards that end, we design and propose (i) *Map Reduce Sweep-line* (MRSWEEP), and (ii) *Map Reduce Sweep-line across Dimensions* (MRSWEEP-D): two sophisticated and highly scalable methods that implement the original sweep-line method and its variants. We provide algorithmic details of fully distributed in-memory versions of the proposed methods in a cluster environment with distributed memory and storage, using the MapReduce programming paradigm and the Apache Spark framework. To achieve very high levels of parallelization, we utilise different data partitioning techniques. In summary, the major contributions of this work are:

- We design and implement MRSWEEP, a distributed in-memory sweep-line method, using the Apache Spark framework, which can efficiently address demanding axis-aligned object intersection problems. The method can gracefully address instances of the problem in any dimension (1-*D*, 2-*D*, ..., D-*D*).

TABLE I: Summary of Notations

Notation	Description
N	Number of objects in the dataset
D	Number of dimensions
R	A rectangle or box (2-D geometric object)
(x_R^0, y_R^0)	(x, y)-coordinates of the bottom left corner of R
(x_R^1, y_R^1)	(x, y)-coordinates of the top right corner of R
(x_R^c, y_R^c)	(x, y)-coordinates of the center of R
Z	Sample size
P	Number of partitions
K	Number of intersecting object pairs

- We design and implement MRSWEEP-D, a variant of MRSWEEP that is more effective than MRSWEEP under specific data assumptions and conditions.
- We present a thorough experimental evaluation of MRSWEEP and MRSWEEP-D to demonstrate their scalability effectiveness. We also examine the behavior of the algorithms under various assumptions of the input dataset and parameters of the distribution environment.
- We make *source code* and *data* publicly available to encourage reproducibility of results.

The remainder of this paper is organized as follows: Section II introduces preliminaries of the problem and provides background information on the current state-of-the-art solutions to address the axis-aligned object intersection problem. Our proposed distributed in-memory methods, the overall algorithmic analysis and implementation details are presented in Section III. Section IV presents an experimental evaluation of the different methods. After reviewing the related work in Section V, we conclude in Section VI.

II. BACKGROUND

In this section, we introduce notation and preliminaries of the problem of interest and provide some background of the main approaches to address it.

A. The Object Intersection Problem

The 1-D case of the problem of interest (detecting intersecting line-segments) constitutes a simplified version of the D-dimensional case. Furthermore, the case of more than two (2) dimensions, includes steps that are analogous of those required for the 2-D case (detecting intersections of rectangles). Therefore, for the rest of the manuscript, we only provide algorithmic details and examples using the case of 2-D objects. The distributed methods presented, as well as the implementation details can be applied directly to axis-aligned object dataset of arbitrary dimensions. Whenever special treatment is required for the case of 1-D objects, we explicitly mention that and provide a brief explanation.

The Problem in 2-D: Consider the Cartesian plane $\mathbb{R}^2 = \mathbb{R} \times \mathbb{R}$, where $\mathbb{R}$ is the set of all real numbers. Let a rectangle R be defined by the (x, y)-coordinates of two points in $\mathbb{R}^2$, one point representing its bottom-left corner (x_R^0, y_R^0) and one representing its top-right corner (x_R^1, y_R^1), respectively. As the two points represent the diagonal corners of the rectangle R,

it is $x_R^0 < x_R^1$ and $y_R^0 < y_R^1$. Now, for a pair of rectangles A and B, consider the following conditions:

$$max(x_A^0, x_B^0) \leq min(x_A^1, x_B^1) \quad (1)$$

$$max(y_A^0, y_A^1) \leq min(y_B^0, y_B^1) \quad (2)$$

The two rectangles A and B are intersecting if and only if both (1) and (2) are true. Note that (1), (2) check whether the rectangles are intersecting in the X-axis and Y-axis, respectively.

When two rectangles A and B are intersecting, then their overlapping area defines a new rectangle, called an *overlap* and denoted as O_{AB}. The rectangle coordinates of O_{AB} are $(max(x_A^0, x_B^0), max(y_A^0, y_A^1))$ and $(min(x_A^1, x_B^1), min(y_B^0, y_B^1))$. Note that the dimensions of the overlap O_{AB} are given by:

$$width_{O_{AB}} = min(x_A^1, x_B^1) - max(x_A^0, x_B^0) \quad (3)$$

$$height_{O_{AB}} = min(y_B^0, y_B^1) - max(y_A^0, y_A^1) \quad (4)$$

The size $S_{O_{AB}}$ of the overlap O_{AB} is given by:

$$S_{O_{AB}} = width_{O_{AB}} \times height_{O_{AB}} \quad (5)$$

Note that in the case of 1-D, two line-segments A, B are intersecting if and only if the condition (1) is satisfied. Their overlap defines a new line-segment O_{AB} that starts at $(max(x_A^0, x_B^0))$ and ends at $(min(x_A^1, x_B^1))$. The size $S_{O_{AB}}$ of the overlap O_{AB} is given by (3).

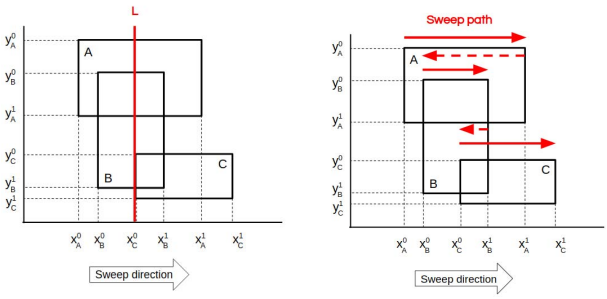
In the standard rectangle intersection problem, we are given a set of N rectangles and asked to detect all pair-wise rectangle intersections K, sometimes along with determining the size of their pair-wise overlap. A worst-case scenario exists where $K = N^2$, meaning that all rectangles are overlapping with each other; this, however, is a degenerate case for most real-world scenarios or applications (otherwise there is no reason to study the intersection analysis on the dataset). We further elaborate on the complexity of the various methods to address the problem in the next paragraphs.

B. Naive Approach

The naive approach to the object intersection problem involves simply comparing every object in the data set to every other using a nested loop. This operation is of course extremely expensive and inefficient, always requiring exactly $\frac{1}{2}N^2$ comparison steps and K steps to report the output, resulting in a computation cost of $O(N^2 + K)$. Therefore this method, although very simple to implement and needing $O(1)$ memory space, is unsuitable for all but the smallest datasets and applications.

C. SWEEP-LINE Algorithm

The much more efficient alternative to the naive approach is the SWEEP-LINE, or Bentley-Ottmann algorithm [5]. This is an algorithmic paradigm that uses a conceptual sweep line to identify and report intersections in Euclidean space. Given a set of rectangles, the first preliminary step of the algorithm involves constructing a list that includes the *start* and *stop* points of all regions in each dimension and sorting them. Then, a *conceptual line*, moves (sweeps) from left to right across the plane in the X dimension, examining the rectangles, one by one, in order. During the sweep, the *active rectangles* (i.e., the ones that the line is currently traversing over) are maintained in a list. When a new region is encountered by the line, we determine whether it is intersecting with any other rectangle in the list of actives in the Y dimension as well, using equation (2). For any pair where this is true, the pair constitutes an intersecting pair, and included in the results. Afterwards, the new rectangle is added to the list of active rectangles. When the upper corner (ending point) of a rectangle is reached by the line, that rectangle is removed from the list of actives.



(a) SWEEP-LINE (b) SCAN-LIST

Fig. 2: Illustrative example of sweep-line algorithm variations.

At any given time during the sweep, the list of active rectangles has size L. The sweep line will visit N lower rectangle corners and N upper ones, where at every lower corner there will be L comparisons performed. Of course, K additional steps are required to output the intersections found. Therefore, the number of operations is $O(\sum_{i=1}^{N} L_i + N + K)$. As mentioned before in a worst case scenario where all the objects are intersecting, L can reach as high as N. Therefore, the upper bound for the computation cost is $O(N * N + N + K) = O(N^2 + K)$. Usually, however, $L \ll N$, and the Bentley-Ottmann algorithm significantly outperforms the naive one in terms of time. In terms of memory space required, as the list of active rectangles is maintained throughout the sweep, the memory required is $O(L) = O(N)$. An illustrative example of the process can be seen in Figure 2a.

It is possible to reduce the upper bound for the computational complexity of the SWEEP-LINE algorithm further, by replacing the list of active rectangles with a balanced binary search tree, instead [7]. The binary tree structure is constructed during the sweep, and it allows for comparing newly encountered rectangles with active ones in logarithmic time, by performing a search on that tree. As this search requires $logL$ steps, the new upper bound for this version of the sweep-line algorithm becomes $O(N + K)logN$, which is the state-of-the-art for non parallelized versions of the sweep-line algorithm.

D. SCAN-LIST *Algorithm*

A critical limitation of the basic Bentley-Ottman sweep-line algorithm is that it must iterate through the entire dataset in series. The need to maintain a record of active objects introduces some invariants, which are violated when trying to unravel the main sweep loop and distributing the work. A straight-forward approach to side-step these invariants and distribute the work to many cores in the same machine is the SCAN-LIST algorithm [12].

In SCAN-LIST, a sweep is performed across a dimension in an arbitrary direction; however, only left edges (represented by the x^0 coordinates) are considered while scanning. The scan step picks one rectangle (R_1) at a time and checks for intersection with the ones to the right of this test rectangle. The scan continues as long as the right-edge coordinate of the selected rectangle $x^1_{R_1}$ is greater than the left-edge coordinate of the newly encountered rectangle $x^1_{R_2}$. At this point, the selected rectangle is discarded and the scan proceeds to the next one. An illustrative example of this process can be seen in Fig. 2b.

This method is actually much less sophisticated than sweep-line, and would be outperformed by it in a sequential execution as it has a worst-case serial complexity of $O(N^2)$. However, it has the benefit of being effortlessly parallelizable, at least within a single machine. All processing for each selected object is independent of the others, and they can be performed simultaneously, thus outperforming the serial sweep-line. However, there are still limitations with this algorithm. The stopping condition for each selected object isn't specific and pre-determined, but instead it depends on the length of the object in the scan dimension (in worst-case scenarios, this could even span the entire dataset!). As such, it is impossible to directly partition the data into smaller batches and distribute them across a cluster environment.

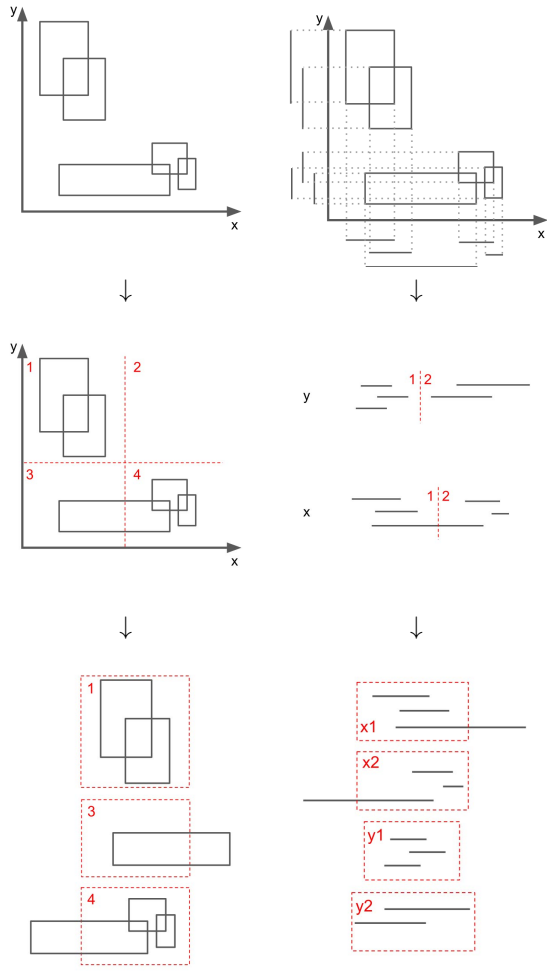
In our proposed method, we utilize elements from both sweep-line and SCAN-LIST, as well as sophisticated techniques for input data partitioning, to achieve a highly parallelized and truly scalable and distributed solution to the axis-aligned object intersection problem.

III. METHODOLOGY

In this section, we introduce the MRSWEEP and MRSWEEP-D methods and all procedures associated with their evaluation. We first describe the steps involved in the MRSWEEP algorithm, including the sampling for the dynamic partitioning. Afterwards, we present its variant MRSWEEP-D, with its own partitioning scheme and additional steps. Finally, we discuss the strengths and limitations of each approach.

A. MRSWEEP

For our proposed algorithm, we employ dynamic partitioning through sampling. As a pre-processing step, we wish to divide the entire observation space in all dimensions so that a roughly equal number of of input data objects is present in every partition. To do that, we try to approximately determine the underlying distribution in each dimension, with the help



(a) all dimensions (b) per dimension

Fig. 3: Illustrative example of data partitioning.

of a representative sample. Although this step adds a slight computational overhead, Apache Spark's architecture allows us to retrieve a sufficiently large sample of size S in negligible time.

We proceed by calculating the coordinates of the center points for each sampled object in all dimensions $(x^c_{R_1}, y^c_{R_1}, x^c_{R_2}, y^c_{R_2}, \ldots)$. Afterwards, we group together and sort the coordinates for each dimension $(\{x^c_{R_1}, x^c_{R_2}, \ldots\}, \{y^c_{R_1}, y^c_{R_2}, \ldots\})$. Finally, since these points should follow the same underlying distribution as the initial dataset, all we need in order to produce P partitions overall (i.e. $\frac{P}{D}$ partitions in each dimension) is to select every $\frac{Z \cdot D}{P}$th value in each dimension and use that as a partition point in that dimension. These partition points are used to define an uneven D-dimensional grid, with each cell receiving an arbitrary partition id.

Having determined the partitions, we can now proceed with

Procedure 1 GetPartitions

Input: Set S of objects in D dimensions
Output: Set of partitions $P_{d,i}$ with their bounds

```
C = sample(S, Z)                          // get sample
C.map(R_i: (R_i^0 + R_i^1)/2)             // get centers
for d ∈ D do
    C_d.sort()                            // sort centers
    for i ∈ [2, (Z · D)/P] do
        P_{d,i} = (C_{d,i−1}, C_{d,i})    // get partition points
    end for
end for
```

processing the dataset itself. As each input object is read, it is compared with the partition grid. For every cell it comes in contact with, a key-value pair is returned, with the value being the object and the key being the corresponding partition id. Effectively, each object is replicated across all the partitions it belongs in, and objects with the same partition id are grouped and processed independently. An illustration of this process can be seen in Fig. 3a.

The next step is sorting the objects in each partition along a sweep direction and simply applying the state-of-the art sweep-line algorithm independently on each of them. Any two objects that are intersecting in at least one point are guaranteed to both belong in the partition that includes that point. Therefore, since all intersecting pairs always appear in at least one partition together, we are guaranteed to correctly find all intersecting pairs in the dataset, with possibly some of the results duplicated. The final step, then, is to simply filter out all duplicate results. The pseudocode for the algorithm can be found in Alg. 1.

Note that in 2-D+ this method will only guarantee a roughly equal number of objects in each partition, as long as their distribution in space is close to either the gaussian or the uniform distributions. If that isn't true, there is no guarantee that the resulting partitions will be balanced. For example, in Fig. 3a, partition 2 has no objects at all, and is therefore not included. In cases like these, much more complex and computationally heavy methods are required to produce balanced partitions of D-dimensional space. We instead try to address this limitation through the algorithm itself, as shown in the next subsection.

B. MRSWEEP-D

Similar to MRSWEEP, MRSWEEP-D we dynamically determine a number of partition points using a sample. However, while before we divided the entire observation space into P partitions, each with the same number of dimensions as the original space, now we split the objects even further, down to their component factors in each dimension instead. This means that we create $\frac{P}{D}$ partitions for every dimension $(P_{x1}, P_{x2}, P_{y1}, P_{y2}, \ldots)$, and each one contains the projections of some objects in the corresponding dimension. An illustration of this process can be seen in Fig. 3b.

Algorithm 1 MRSWEEP

Input: Set S of objects in D dimensions
Output: Set of intersect pairs $(R_1R_2R_4R_6, \ldots)$

```
P_{d,i} = GetPartitions(S)                // get partition points

function PARTITION(R)
    for P_i ∈ P do                        // for every partition
        if R ∈ P_i then                   // check if object is in
            yield (P_i, R)                // emit and continue
        end if
    end for
end function
S.map(partition).groupByKey()             // perform partitioning

S.mapValues(SweepLine)                    // get intersecting pairs
S.groupByKey()
S.mapValues(x: x[0])   // remove duplicates, keep only 1st
```

After the separate factors of all objects have been sent to the corresponding partitions, once again a sorting step takes place. It is now possible to simply execute the classic sweep-line or SCAN-LIST algorithm for 1-D line segments, and find the pairs of objects that are partially intersecting, i.e. intersecting in at least one dimension. The SCAN-LIST was selected out of the two for the experiments in this work, as it lends itself better to parallel, multi-threaded calculation and, as such, performed better. Of course, this means that another step is necessary: we need to determine which of these are actually intersecting in all dimensions, and which are not. For this purpose, we use a simple key-based reduction, where the key is formed from the ids of the intersecting pair, and the value is the number of dimensions they overlap in. If, after this reduction step, a pair objects is not overlapping in all D dimensions, then it is filtered out, and only the complete intersections are returned. Alg. 2 outlines the exact process.

This time, since the partition points in each dimension approximately correspond to the underlying distribution in that dimension, the partitions produced by MRSWEEP-D should always have approximately equal numbers of objects (as can be seen in 3b. Thus, processing can be more effectively distributed, with the end result being a more scalable algorithm. There is, however, a trade-off: depending on the distribution of objects in space and the number of dimensions, the number of intermediate results (i.e. partial intersections) can be prohibitively large. Specifically, when the objects are uniformly distributed in space, the fact that they intersect in one dimension is in no way indicative about them intersecting in another; thus MRSWEEP-D will produce many intermediate results that will be later discarded. Furthermore, if D is high, and a pair of objects are intersecting in only 1 or 2 dimensions, MRSWEEP-D will still examine all dimensions for potential partial intersections, whereas MRSWEEP will stop after the first negative result.

Overall, MRSWEEP should be well-suited to problems

Algorithm 2 MRSWEEP-D

Input: Set S of objects in D dimensions
Output: Set of intersection pairs $(R_1\_R_2, R_4\_R_6, \ldots)$

```
P_{d,i} = GetPartitions(S)                    // get partition points

function PARTITION(R)
    for d ∈ D do                              // for every dimension
        for P_{d,i} ∈ P do                    // for every partition
            if R_d ∈ P_{d,i} then             // check if object is in
                yield (P_{d,i}, R)            // emit and continue
            end if
        end for
    end for
end function
S.map(partition).groupByKey()                 // perform partitioning

S.mapValues(ScanList)                         // get pairs

function MERGE(Factors of intersect. R_1R_2)
    for d ∈ D do
        set(R_1-R_{2d})                       // remove duplicates in same d
    end for
    if length(R_1-R_2)= d then                //all intersect
        yield R_1-R_2                         // emit and continue
    end if
end function
S.groupByKey().mapValues(merge)
```

where the objects are very high-dimensional or are uniformly/normally distributed in space. On the other hand, MRSWEEP-D is better geared towards cases where the objects are concentrated in several specific points ("hotspots") in space, and have a small number of dimensions. It should be possible to determine which one is more appropriate by examining the properties of the dataset itself.

IV. EXPERIMENTAL EVALUATION

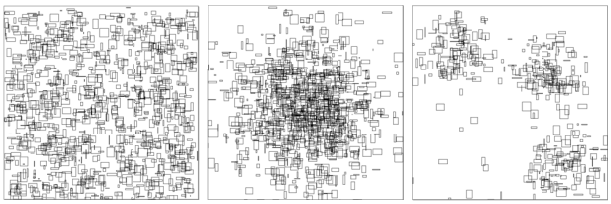
In this section, we experimentally evaluate the performance of our proposed distributed algorithms MRSWEEP and MRSWEEP-D. We also evaluate their behaviour assuming different levels of problem complexity. Before presenting the results, we provide details of the computational environment and the data sets employed.

A. Cluster Environment

All experiments are conducted using Apache Spark deployed on a Hadoop cluster with 7 nodes, each with 24 Intel® Xeon® E5-2620 CPU v3 @ 2.40GHz cores and 64 GB of RAM, at a total capacity of 168 cores (limited to 140 for Spark usage) and 448GB of RAM. For the algorithm implementation we are using Python 3.7 and PySpark 2.4.4. For each experiment that takes up to 2 hours, we execute any evaluated algorithm ten (10) independent times and report average values (of execution time or other quantities, accordingly).

TABLE II: Summary of Datasets

Name	N	$K(\approx)$	D	Distribution
XS-N	10^6	$2 \cdot 10^6$	2	*normal* ($\mu = 0.5T; \sigma = 0.1T$)
S-N	$5 \cdot 10^6$	10^7	2	*normal* ($\mu = 0.5T; \sigma = 0.1T$)
M-U	10^7	$2 \cdot 10^7$	2	*uniform*
M-N	10^7	$2 \cdot 10^7$	2	*normal* ($\mu = 0.5T; \sigma = 0.1T$)
M-H	10^7	$2 \cdot 10^7$	2	*hotspots* ($\#spots = 3$)
L-N	$5 \cdot 10^7$	10^8	2	*normal* ($\mu = 0.5T; \sigma = 0.1T$)
XL-N	10^8	$2 \cdot 10^8$	2	*normal* ($\mu = 0.5T; \sigma = 0.1T$)
M-N-1D	10^7	$2 \cdot 10^7$	1	*normal* ($\mu = 0.5T; \sigma = 0.1T$)
M-N-3D	10^7	$2 \cdot 10^7$	3	*normal* ($\mu = 0.5T; \sigma = 0.1T$)
M-N-4D	10^7	$2 \cdot 10^7$	4	*normal* ($\mu = 0.5T; \sigma = 0.1T$)



(a) *uniform* (b) *normal* (c) *hotspots*

Fig. 4: Small examples of the different distributions of the input data that are employed in the experiments. The *uniform* distributes objects equally in the Euclidean space; the *normal* or *gaussian* creates a hot spot in the Euclidean space, where a large number of objects is concentrated and results in large number of multiple overlaps; the *hotspots* replicates the behavior of the *gaussian* multiple times and generates multiple hot spots in the Euclidean space.

B. Data

In order to evaluate the behavior of the algorithms under certain conditions, we use synthetic data. A data generator was implemented that produces data sets of specific characteristics thanks to a controlled number of parameters. We define a Euclidean space in 1-D, 2-D or a higher dimension D and fix the size of each dimension to be T, effectively ranging in $[0, T]$; unless otherwise noted $T = 1000$. Within that space, we randomly generate N objects(e.g., line segments in 1-D or rectangles in 2-D, etc.). The size of each dimension (side) of an object is uniformly at random selected from the range $t : [0, t_{max}]$, where $t_{max} = r \cdot T$ and $r \in [0, 1]$ represents a ratio of the total length T. For example, if $r = 0.01$ and $T = 1000$, then the size for each dimension of a region would be bound by $0 \leq t \leq 10$. Randomly generated objects will result in a number of intersections. To control how objects are located in the Euclidean space (and therefore the likelihood of them intersecting) we rely in three different probability distributions, *uniform*, *gaussian or normal* and *hotspots*. These distributions and their properties were selected to reflect a wide variety of possible real-world conditions and different levels of problem complexity; the *gaussian* or *normal* distribution has mean (μ) of $\mu = 0.5T$ and standard deviation (σ) of $\sigma = 0.1T$, while the *hotspots* one is a combination of a specified number of gaussians with random mean (μ) values and sigma (σ) values in the range [0,1T]. In

practice, the distribution is specified by the user and when an object is generated the position of its starting coordinate for each dimension is dictated by one of the three possible distributions. Illustrative examples of these distributions for 2-D can be seen in Figure 4. Therefore, the configurable parameters of the synthetic data generator are *the number of objects* N, *a ratio* r, *a dimension* D and *a spatial distribution, with related parameters*. For experimental evaluation purposes, various datasets were created, ranging from 10^6 to 10^9 regions and resulting in 10^7 to 10^{10} intersections. The reference name and details of all the generated datasets used in the experiments can be found in Table II.

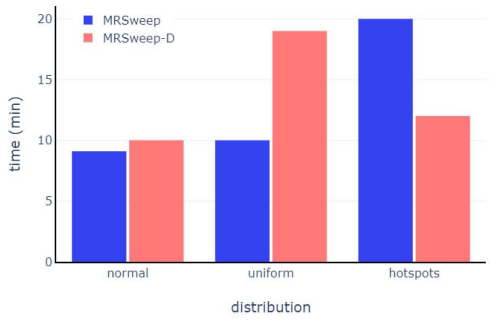
C. Experiments

We aim to evaluate the following aspects:

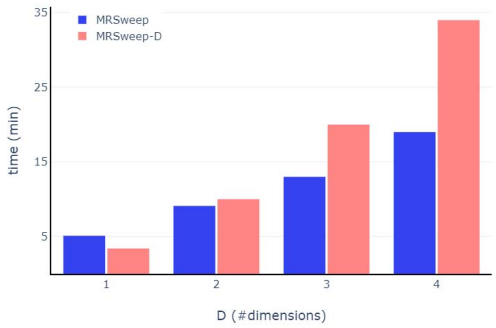
- **Effect of Object Distribution on Performance**. How does the distribution of objects in Euclidean space affect the performance of MRSWEEP and MRSWEEP-D? How the performance is affected if objects are defined in higher-dimensions?
- **Effect of Partitioning Parameters on Performance**. How do MRSWEEP and MRSWEEP-D behave for different numbers of data partitions?
- **Scalability**. How does our proposed MRSWEEP and MRSWEEP-D methods scale with increasing numbers of cluster cores, for various data sizes?

1) Effect of Object Distribution on Performance: First, we examine the performance of MRSWEEP and MRSWEEP-D for datasets with different spatial distributions. To that end, we use the datasets M-N, M-U and M-H, that contain objects following a *normal* (or *gaussian*), *uniform* and *hotspots* spatial distribution, respectively. The results can be seen in Figure 5a. As expected, MRSWEEP outperforms MRSWEEP-D for uniformly distributed objects; the opposite is true for the *hotspots* distribution. A major cause of this is *partial intersections*: when two objects intersect in some dimension(s) but not in all of them, MRSWEEP-D will examine all dimensions, whereas MRSWEEP will examine the sweep dimension first and won't proceed to the next if they don't intersect. This results in more calculation steps and a slower execution time for MRSWEEP-D in the *uniform* distribution, where partial intersections are more likely to occur. However, for the *hotspots* distribution, MRSWEEP-D performs better, despite this additional cost. The main reason is the imbalance in partition sizes: MRSWEEP attempts to partition the entire space using an uneven grid, which produces unbalanced partitions, as the the objects are not symmetrically distributed. When the objects follow a *normal* distribution, the objects can be evenly partitioned and partial intersections are less likely to occur; therefore, both algorithms perform similarly well.

Secondly, we explore the behavior of the two algorithms as a function of higher dimensions. For that task, we use the datasets with that refer to higher dimensions M-N-1D, M-N, M-N-3D and M-N-4D. Figure 5b displays the results. In 1-D, both algorithms follow the exact same partitioning technique. However, MRSWEEP-D slightly outperforms MRSWEEP, as it



(a) distribution×time

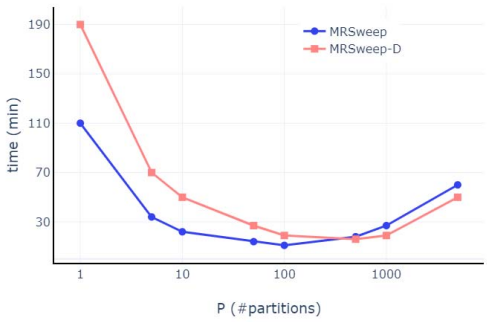


(b) D×time

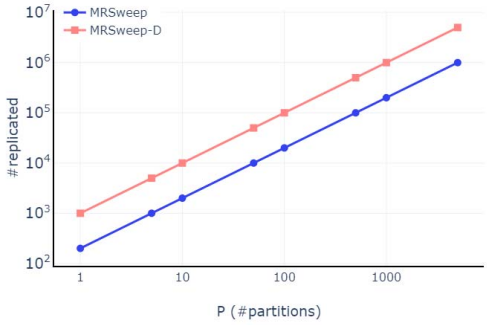
Fig. 5: Effect of spatial distribution and number of dimensions on execution time for MRSWEEP and MRSWEEP-D.

utilizes the SCAN-LIST algorithm, which as discussed in Section III, can be executed in parallel threads – something that Apache Spark handles automatically. However, as the number of dimensions increases, partial intersections are becoming more and more likely, while object intersections (where objects intersect in all dimensions) are less likely to occur. This results in more intermediate results produced for MRSWEEP-D, and it is quickly outperformed by MRSWEEP.

2) Effect of Partitioning Parameters on Performance: Subsequently, we explore the impact that the number of partitions has on the performance of MRSWEEP and MRSWEEP-D, in terms of execution time. This is particularly helpful in determining the optimal partitioning properties for each algorithm. We use the M-N dataset, setting the number of partitions P from 1 to 5000. Note that for a value of $P = 1$, the entire dataset consists of the single partition and the execution is effectively non-distributed. For this experiment we employ all available nodes in the cluster (effectively 140 cores). The results can be seen in Figure 6a. The first observation is that as the number of partitions increases, the execution time decreases. This is intuitive – as the data is divided into more partitions, it can be easier to distribute the computation to more workers/cores on the cluster. However, this downwards

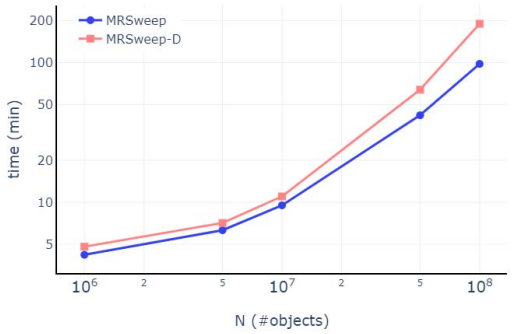


(a) $P\times$time

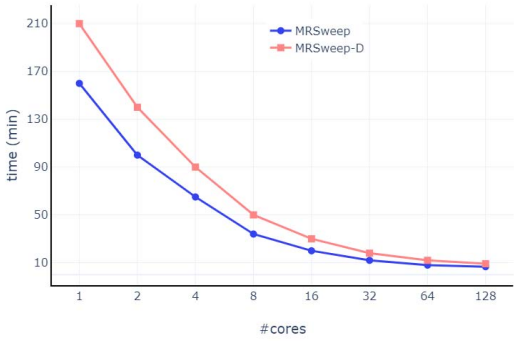


(b) $P\times$#replicated

Fig. 6: Effect of number of partitions on execution time and number of additional replicated objects for MRSWEEP and MRSWEEP-D.



(a) $N\times$time



(b) $\#cores\times$time

Fig. 7: Effect of number of objects and number of cores on execution time for MRSWEEP and MRSWEEP-D.

trend continues up to the point that the number of partitions is similar to the number of cores (140), at which point the trend switches to upwards, indicating that the execution time starts increasing, instead of continuing to decrease.

There are a few reasons for this behavior. First, as the number of partitions increases, the amount of intermediate results produced in the MapReduce operations increase as well, creating more cluster communication costs (shuffling over the network). But, more importantly, this behavior is explained by the effect that the partitioning has to the data and the need to merge results that have been replicated in multiple partitions. Recall that every time an object is spanning two (or more) partitions, we replicate that object in all partitions and merge it at a later stage. To further study this behavior we perform an additional experiment where the number of replicated objects ($\#replicated$) is monitored as a function of the number of partitions ($\#partitions$). Figure 6b shows the results. Note that the plot is in log-log scale and appears as a straight line. The slope of a log-log plot gives the exponent in the relationship between $\#replicated$ and $\#partitions$:

$$\#replicated \propto \#partitions^{100}$$

The exponent (i.e., 100) is an artifact of the synthetic generator parameter that controls the size of each object. So, as the number of partitions increases above $\sim$ 140, the number of replicated objects increases faster, and leads to the upwards trend of the execution time that we observe in Figure 6a.

In all the previous cases, MRSWEEP outperforms MRSWEEP-D, as the dataset follows a gaussian distribution, for which the former is better suited for. Other than that, the trends remain the same.

3) Scalability: Finally, we evaluate the time performance of MRSWEEP and MRSWEEP-D. In the following experiments, we fix the number of partitions to $P = 300$.

First, we evaluate the performance of the methods as a function of different data size (i.e., number of objects). For this experiment, we make use of the XS-N, S-N, M-N, L-N, XL-N datasets and employ all available nodes in the cluster (effectively 140 cores). Figure 7a shows the results. It can be seen that both MRSWEEP and MRSWEEP-D can detect the intersections in very large instances of the problem (10^8 objects, having $2 \cdot 10^8$ intersections) within a few hours; in slightly smaller datasets within minutes. The increase on the time for larger datasets can be explained by the number

of replicates that are created during the execution of the algorithm. As we explained in paragraph IV-C2 and since in this experiment the number of partitions is fixed to $P = 300$, a larger number of objects will increase the number of object replicates; therefore the method will have to carry on the overhead of merging replicates found in different partitions.

More importantly, we evaluate the performance of the methods as a function of the numbers of available cores. For this experiment, we make use of the M-N dataset. The results can be seen in Figure 7b. We observe that as the number of cores is increasing (from 1 to 128) the execution time of both MRSWEEP and MRSWEEP-D is steadily decreasing. The decrease is not linear, since by increasing the number of cores we carry on the overhead of shuffling and coordinating the different cores (communication cost). We have effectively provided evidence that our proposed distributed methods can scale gracefully to very large data sizes and make efficient use of the available cores of a large cluster. More importantly, they can yield significant performance improvements to current state-of-the-art implementations. For instance, we were able to find the intersections of 10^7 objects, having $2{\cdot}10^7$ intersections using 128 cores in less than 10 minutes – the same operation would require approximately 200 minutes when executed on a single core.

V. Related Work

Our research is related to the *object intersection* problem in computational geometry, *spatial data structures* that commonly arise in the context of spatial databases and spatial data mining, as well as the problem of *algorithm distribution* and *parallelization* in *MapReduce*. Several key ideas on these topics have already been cited throughout the paper; here we present a more comprehensive coverage of each topic.

A. Sweep-line Algorithm

A great number of data structures and algorithms have been developed that deal with finding and performing queries on intersecting objects [6], [13]. One of the most common related problems to that is the axis-aligned (or iso-oriented) rectangles in $\mathbb{R}^d$ [14], [15], or the very similar orthogonal range search problem [16]. A critical step in various different applications such as collision detection in computer simulations [17] or for object placement problems [18], this problem has seen a lot of usage in data mining research [19], [20].

The state-of-the-art techniques used in related research belong to one of two families of algorithms: either a *sweep-line* (also known as *plane sweep*) or a *divide-and-conquer* algorithm, which have been shown to be equivalent in computation cost [21]. These algorithms are commonly tasked with the purpose of constructing data structures that can accommodate spatial queries by identifying pair-wise [22], [23] or multiple [24] intersections of objects.

There have been some extensive solutions on optimizing these techniques and methodologies for processing large amounts of data in a single processor/single disk environment [25], as well as many proposed parallelization methods for different versions of the sweep-line algorithm. In [11], a parallel algorithm for the general, non-axis-aligned interval intersection problem is proposed, that relies on statically partitioning the observation space into strips and performing the sweep-line algorithm on these. In our research, we focus on a parallel and distributed implementation of the axis-aligned objects case instead. Although it's more specific, it is of great importance to numerous problems and it allows for better optimized and efficient calculation techniques. In other similar cases, the partitioning used introduces synchronization points that aren't feasible in a distributed environment [10]. Furthermore, in [12], the highly parallel scan-list algorithm is proposed. However, it is best suited for single-node, multi-threaded parallelization, as every separate process running in parallel needs to be able to access the entire dataset locally. In a distributed environment, this introduces the highly expensive requirement of replicating the entire dataset to every node. Such a distributed sweep-line in external memory algorithm has been proposed to address very few computational geometry problems, such as the skyline merge one [26], but not the rectangle intersection problem.

B. MapReduce and Apache Spark

MapReduce [27] is a distributed parallel processing model and execution environment for processing large data sets running on large clusters of commodity machines. Apache Spark [28] is an open-source, distributed, in-memory computing framework and architecture that follows a similar approach but overcomes some of the inefficiencies of MapReduce. MapReduce and Spark have been utilized for various spatial data mining and distributed processing problems, such as spatial [29] and spatiotemporal joins [30]. A high-level approach for parallelization and distribution of spatial data processing has been proposed in [31], however it is not specifically directed towards the axis-aligned object intersection problem. To the best of our knowledge, this is the first work that presents methods to distribute the sweep-line paradigm over a cluster with multiple nodes, using Apache Spark.

VI. Conclusions

The axis-aligned object intersection problem is an essential computational geometry problem and a key component in numerous applications, such as data mining and machine learning, spatial databases, hardware design, and computer vision. Current state-of-the-art solutions to address the problem rely on an algorithmic paradigm known as sweep-line or sweep plane. While this paradigm offers significant advantages to naive approaches, it presents limitations that prevent it to scale to very large instances of the problem (i.e., instances of large number of objects, large number of intersections or both).

Motivated by the scalability limitation of such a popular and useful method, we presented MRSWEEP and MRSWEEP-D, two efficient distributed algorithms that implement the sweep-line paradigm. Our comprehensive experiments provide evidence that parallelization on a cluster using MapReduce on Apache Spark yields significant performance improvements.

For instance, we were able to find the intersections of 10^7 million objects, having $2 \cdot 10^7$ intersections using 128 cores in less than 10 minutes – the same operation would require approximately 200 minutes when executed on a single core. To the best of our knowledge, this is the first work that deals with distributing the sweep-line paradigm over a cluster with multiple nodes, with distributed storage and distributed memory. An important characteristic of the distribution strategy is that the proposed algorithms offer a high level of *versatility*:

- they can adapt or be adapted to instances of the problem of variable levels of complexity (i.e., different distributions of the input data and number of intersections);
- they can gracefully scale to objects of higher dimensions. That is noteworthy, since previous attempts provide customized implementations of the sweep-line that focus on objects of a specific dimension, usually 1-D or 2-D.

The designed algorithms offered also a number of lessons and insights about the complexity of distributing the problem. While, the parallelization of the sweep-line itself is conceptually intuitive, how one should partition the data is not as straightforward as it might seem. While joining partial results that span boundaries of the data partitions is not difficult, an increase on the number of partitions might carry substantial overhead that cannot be ignored. Our proposed methods reflect on these considerations and provide a sophisticated way to scale out the computation of already existing implementations of the popular sweep-line algorithm.

Overall, MRSWEEP and MRSWEEP-D offer a practical solution to a challenging real-world problem. As such, we anticipate their impact to be significant in various domains.

Reproducibility: To encourage reproducibility of the results and broader adoption of the methods, we make our synthetic data generator, source code, and data sets publicly available at: https://github.com/tipech/mrsweep.

REFERENCES

[1] J. M. Patel and D. J. DeWitt, "Partition based spatial-merge join," *SIGMOD Rec.*, vol. 25, no. 2, pp. 259–270, Jun. 1996.

[2] T. Pechlivanoglou and M. Papagelis, "Fast and accurate mining of node importance in trajectory networks," *2018 IEEE International Conference on Big Data (Big Data)*, 2018.

[3] J. Fang, J. Wong, K. Zhang, and P. Tang, "A new fast constraint graph generation algorithm for vlsi layout compaction," *1991., IEEE International Symposium on Circuits and Systems*, 1991.

[4] T. Tang, E. L. Bohez, and P. Koomsap, "The sweep plane algorithm for global collision detection with workpiece geometry update for five-axis nc machining," *Computer-Aided Design*, vol. 39, no. 11, p. 1012, 2007.

[5] Bentley and Ottmann, "Algorithms for reporting and counting geometric intersections," *IEEE Transactions on Computers*, vol. C-28, no. 9, pp. 643–647, Sep. 1979.

[6] Bentley and Wood, "An optimal worst case algorithm for reporting intersections of rectangles," *IEEE Transactions on Computers*, vol. C-29, no. 7, pp. 571–577, July 1980.

[7] H. W. Six and D. Wood, "The rectangle intersection problem revisited," *Bit*, vol. 20, no. 4, p. 426–433, 1980.

[8] E. M. McCreight, *Efficient algorithms for enumerating intersecting intervals and rectangles*. Xerox, Palo Alto Research Center, 1980.

[9] M. T. Goodrich, "Intersecting line segments in parallel with an output-sensitive number of processors," *Proceedings of the first annual ACM symposium on Parallel algorithms and architectures - SPAA 89*, 1989.

[10] H.-P. Kriegel, T. Brinkhoff, and R. Schneider, "The combination of spatial access methods and computational geometry in geographic database systems," *Advances in Spatial Databases Lecture Notes in Computer Science*, p. 5–21, 1991.

[11] M. Mckenney and T. Mcguire, "A parallel plane sweep algorithm for multi-core systems," *Proceedings of the 17th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems - GIS 09*, 2009.

[12] 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, p. 966–970, 2013.

[13] F. Dévai and L. Neumann, "A rectangle-intersection algorithm with limited resource requirements," in *2010 10th IEEE International Conference on Computer and Information Technology*. Berlin, Germany: IEEE, June 2010, pp. 2335–2340.

[14] H. Edelsbrunner, "A new approach to rectangle intersections," *International Journal of Computer Mathematics*, vol. 13, no. 3-4, 1983.

[15] T. M. Chan, "A note on maximum independent sets in rectangle intersection graphs," *Information Processing Letters*, vol. 89, no. 1, pp. 19–23, 2004.

[16] B. Chazelle, J. E. Goodman, and R. Pollack, *Advances in discrete and computational geometry: proceedings of the 1996 AMS-IMS-SIAM Joint Summer Research Conference, Discrete and Computational Geometry*. Princeton, NJ: American Mathematical Society, 1999.

[17] Y. Zhou and S. Suri, "Collision detection using bounding boxes: Convexity helps," *Algorithms - ESA 2000 Lecture Notes in Computer Science*, pp. 437–448, 2000.

[18] P. K. Agarwal, M. V. Kreveld, and S. Suri, "Label placement by maximum independent set in rectangles," *Computational Geometry*, vol. 11, no. 3-4, pp. 209–218, 1998.

[19] T. Fukuda, Y. Morimoto, S. Morishita, and T. Tokuyama, "Data mining with optimized two-dimensional association rules," *ACM Transactions on Database Systems*, vol. 26, no. 2, p. 179–213, Jan 2001.

[20] A. Giacometti and A. Soulet, "Dense neighborhood pattern sampling in numerical data," *Proceedings of the 2018 SIAM International Conference on Data Mining*, p. 756–764, Jul 2018.

[21] R. H. Güting and W. Schilling, "A practical divide-and-conquer algorithm for the rectangle intersection problem," *Information Sciences*, vol. 42, no. 2, pp. 95–112, 1987.

[22] L. Arge, O. Procopiuc, S. Ramaswamy, T. Suel, and J. S. Vitter, "Scalable sweeping-based spatial join," in *Proc. of the 24rd International Conference on Very Large Data Bases*, ser. VLDB '98. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1998, pp. 570–581.

[23] F. Zhang, X.-Z. Qiao, and Z.-Y. Liu, "A parallel smith-waterman algorithm based on divide and conquer," in *Fifth International Conference on Algorithms and Architectures for Parallel Processing*. IEEE, Oct 2002, pp. 162–169.

[24] T. Pechlivanoglou, V. Chu, and M. Papagelis, "Efficient mining and exploration of multiple axis-aligned intersecting objects," *2019 IEEE International Conference on Data Mining (ICDM)*, 2019.

[25] M. T. Goodrich, Jyh-Jong Tsay, D. E. Vengroff, and J. S. Vitter, "External-memory computational geometry," in *Proceedings of 1993 IEEE 34th Annual Foundations of Computer Science*, 1993, pp. 714–723.

[26] C. Sheng and Y. Tao, "On finding skylines in external memory," in *Proceedings of the thirtieth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, 2011, pp. 107–116.

[27] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, p. 107, Jan 2008.

[28] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster computing with working sets," in *Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing*, ser. HotCloud'10. USA: USENIX Association, 2010, p. 10.

[29] E. H. Jacox and H. Samet, "Spatial join techniques," *ACM Transactions on Database Systems*, vol. 32, no. 1, Jan 2007.

[30] R. T. Whitman, M. B. Park, B. G. Marsh, and E. G. Hoel, "Spatio-temporal join on apache spark," *Proceedings of the 25th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems - SIGSPATIAL17*, 2017.

[31] K. Wang, J. Han, B. Tu, J. Dai, W. Zhou, and X. Song, "Accelerating spatial data processing with mapreduce," *2010 IEEE 16th International Conference on Parallel and Distributed Systems*, 2010.