7-14-2020

Efficient Parallel and Adaptive Partitioning for Load-balancing in Spatial Join

Jie Yang

Satish Puri

Efficient Parallel and Adaptive Partitioning for Load-balancing in Spatial Join

Jie Yang
Computer Science Department, Marquette University, Milwaukee, WI
Satish Puri
Computer Science Department, Marquette University, Milwaukee, WI

Abstract:

Due to the developments of topographic techniques, clear satellite imagery, and various means for collecting information, geospatial datasets are growing in volume, complexity, and heterogeneity. For efficient execution of spatial computations and analytics on large spatial data sets, parallel processing is required. To exploit fine-grained parallel processing in large scale compute clusters, partitioning in a load-balanced way is necessary for skewed datasets. In this work, we focus on spatial join operation where the inputs are two layers of geospatial data. Our partitioning method for spatial join uses

Adaptive Partitioning (ADP) technique, which is based on Quadtree partitioning. Unlike existing partitioning techniques, ADP partitions the spatial join workload instead of partitioning the individual datasets separately to provide better load-balancing. Based on our experimental evaluation, ADP partitions spatial data in a more balanced way than Quadtree partitioning and Uniform grid partitioning. ADP uses an output-sensitive duplication avoidance technique which minimizes duplication of geometries that are not part of spatial join output. In a distributed memory environment, this technique can reduce data communication and storage requirements compared to traditional methods.To improve the performance of ADP, an MPI+Threads based parallelization is presented. With ParADP, a pair of real world datasets, one with 717 million polylines and another with 10 million polygons, is partitioned into 65,536 grid cells within 7 seconds. ParADP performs well with both good weak scaling up to 4,032 CPU cores and good strong scaling up to 4,032 CPU cores.

SECTION I. Introduction

With the increasing volume and complexity of spatial data, there is an increasing demand for efficient geospatial techniques for parallelizing spatial computations [11]. Spatial join and map overlay are important in many scenarios like disaster prediction and rescue, urban planning and so on. Parallel processing can be used to speed up the compute- and data-intensive spatial computations. Spatial data partitioning is an efficient method for data-parallel applications. However, spatial data is often skewed and contains a variety of geometric shapes, leading to a load-balancing problem in the parallelization of spatial computations.

Spatial join involves two spatial layers, namely, *R* and *S.* Performing spatial join queries with predicate *Intersects*, *Contains*, *Overlap*, etc, on *R* and *S* generates a collection of pairs (*r, s*), where $r \in R$, $s \in S$ that satisfy the join predicate. For example, "find all roads that cross a river" is an *Intersects* query [9]. A spatial join can be performed in two phases: 1) filter phase and 2) refinement phase. In the filter phase, the minimum bounding rectangles (MBR) of geometries are utilized to generate a collection of candidate pairs where each pair consists of cross-layer geometries whose MBRs have spatial overlap. These candidate pairs are further refined in the second phase using the actual geometric representations.

Many existing spatial data partitioning techniques are based on one layer, which ignore the distribution of data in another layer. In our prior work, we developed MPI-GIS for partition-based polygon overlay and spatial join computation [14]. MPI-Vector-IO is a component of MPI-GIS that performs parallel I/O of spatial data stored in parallel filesystem [13]. We experimented with uniform grid partitioning which does not perform well for skewed data. Adaptive spatial partitioning, as described in this paper, is designed to improve the load balancing in MPI-GIS.

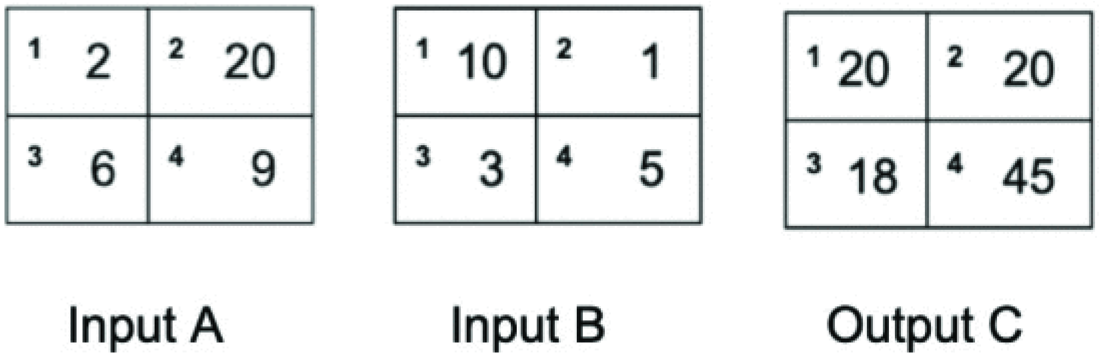


Fig. 1. Number of geometries shown in each grid cell. The workload of a cell in grid C is the product of the number of geometries present in corresponding cells in A and B (e.g., workload in the fourth cell is 9*5).

To illustrate workload partitioning for spatial join, an example is provided in Figure 1 using A and B as the two input layers. A and B have different data distribution and the data is partitioned among four grid cells. The output is layer C which assumes the worst case scenario where a geometry in A needs to be compared against all geometries in B. In output grid *C*, the maximum workload is present in the fourth cell, even though the corresponding fourth cell in grid A and B do not have the maximum geometries in their respective grids. This is in contrast to the traditional partitioning algorithms which will prioritize dividing the second cell in grid A for example. However, neither partitioning on *A* nor *B* alone will focus on partitioning the actual workload. In this paper, we propose a Quadtree-based algorithm (ADP) based on both layers. ADP takes the distribution of geometries in both layers into consideration which can improve spatial partitioning by producing grid cells with similar workload.

Since we use a filtering-based approach to find the potentially overlapping geometries, we can minimize duplication of geometries that do not take part in spatial computations in the refine phase. We refer to this technique as output-sensitive duplication avoidance. This is not possible in a single layer partitioning approach.

Moreover, we propose a parallel adaptive partitioning algorithm (ParADP) for High-Performance Computing (HPC) environment using MPI and C++ threads. Our distributed-memory algorithm with *p* multi-core processors scans the MBRs of an entire dataset by choosing *p* sample MBRs by each processor to get a global view of the data distribution. This is accomplished using the sample sort algorithm. For an HPC cluster with *p* multi-core processors with *q* cores each, *p* vertical stripes and *q* horizontal stripes within a vertical stripe are created. Further partitioning is carried out by each CPU core in parallel to meet the user-defined number of partitions. This method is designed to keep the processors busy and minimize the overall data movement during spatial partitioning. Once the grid partitions are created and geometries are mapped to the grid cells, actual geometries can be finally moved to the corresponding cell(s) where they belong. Therefore, the actual geometries need to move only once from source processor to destination processor. This is in contrast to dynamic load-balancing approach where repartitioning is used after initial partitioning to distribute workload to processors with lighter workload [9].

Various experiments are designed to inspect the performance of ADP and Parallel Adaptive Partitioning (ParADP). As a sequential partitioning technique, ADP's and ParADP's partition qualities are compared with Quadtree partitioning and Uniform partitioning. Our implementations use Geometry Engine

OpenSource[1] (GEOS) library which provides 1) spatial data indices such as Rtree, 2) geometry-based algorithms, and 3) parsing of spatial data.

The main contributions of this paper are as follows:

1. A load-balancing focused partitioning algorithm together with an improved Duplication Avoidance technique and an OpenMP tasks based in-memory parallel Quadtree partitioning implementation.
2. A fast adaptive parallel partitioning algorithm for load-balancing compute-intensive spatial operations implemented using Message Passing Interface (MPI) and C++ threads for spatial datasets containing geometries like polyline and polygon.
3. Experimental evaluation of the algorithm on a large compute cluster containing up to 4032 CPU cores with real-world datasets. Partitioning two layers 1) *roads* (24 GB) and 2) *parks* (9 GB) containing 75 million candidate pairs is completed within 7 seconds on a cluster of 4032 cores.

This paper is organized as follows. Section II introduces background information and related work. Section III describes Adaptive Partitioning and its OpenMP based version. Section IV presents a parallel partitioning algorithm for ADP. Section V evaluates the performance of ADP and ParADP. Finally, Section VI concludes this paper.

SECTION II. Background and Related Work

Partitioning spatial data has been well-studied in literature. Equi-Partitioning, Min-Skew [2], Uniform grid, R-tree, Quadtree, and binary space partitioning are some classic examples of space partitioning. The choice of partitioning scheme depends on the application where it is used. Multijagged is a scalable spatial data partitioning algorithm [5]. However, it is applicable for point data only. In our work, we consider polyline and polygon data as input.

Parallel and Distributed partitioning:

Parallel data partitioning has been studied in the context of spatial query processing, spatial join operation, and polygon overlay [8], [12], [13], [15], [19], [24][25]–[26]. SpatialHadoop supports different partitioning schemes using techniques based on Quadtree, R-tree, grid, etc in a MapReduce environment [7]. PolySketch is a tile-based partitioning of polygons and polylines for GPU-based computations [1], [23]. Our parallel partitioning algorithm uses MPI.

MPI-based GIS system:

In our prior work, we experimented with MPI-based approaches [3], [14] and developed parallel I/O and partitioning framework called MPI-Vector-IO as an HPC system [13]. MPI-Vector-IO partitions WKT files stored in parallel filesystems like Lustre and GPFS into file splits. After data partitioning, a uniform grid is used for spatial partitioning. However, it suffers from load-imbalance for skewed data due to the lack of adaptive grid partitioning. This motivated the research into load-balancing spatial partitioning techniques.

Load-balancing:

The output for an ideal spatial partitioning algorithm is to produce partitions that can be assigned to processors in a load-balanced fashion so that the total execution time is minimized. Ideally, processors should have equal amounts of work and a processor is not waiting for other processors to finish their computation. Both SpatialHadoop (SH) and Hadoop-GIS use dynamic load balancing present in Apache Hadoop framework [4], [7], [21]. In [7], Quadtree method performed well relatively to other methods. To avoid the cost of reading and shuffle-exchange of the entire data, SH reads only a small percentage (e.g. 1%) of the data randomly to generate a global space partitioning. Our methods presented here read all the MBR data. In our prior work, we studied load-balancing using Asynchronous Dynamic Load Balancing (ADLB) library, but the scalability was limited due to the high cost of moving polygonal data across MPI processes [20].

Data skew results in load imbalance. To mitigate the effects of load imbalance, SPINOJA system [15] partitions the spatial dataset such that the amount of computation demanded by each partition is equalized and the processing skew is minimized. Heuristics like declustering skewed distribution of geometries and round-robin assignment of partitions to processors has been shown to be effective for loadbalancing [15], [16]. The experimental evaluation in [15] was done on a single processor with 8 cores. In our current work, we have evaluated the performance using thousands of CPU cores in a distributed memory environment.

In this paper, ST_intersection[2] and ST_intersects[3] operations on two datasets are considered. ST_intersection is used to find the intersection region of two geometries and ST_intersects is used to find whether two geometries intersect with each other.

SECTION III. Adaptive Partitioning

In our adaptive partitioning (ADP) method, we consider both layers to capture the data skew inherent in spatial join and overlay operations. For each geometry in a layer, we take its minimum bounding rectangle (MBR) and the number of points it contains as input. The output is an adaptive grid consisting of cells and a mapping from candidate pairs to grid cells. The goal is to generate a spatial partition that minimizes the load imbalance when spatial computations are carried out in the refine phase in each cell.

The Adaptive Partitioning contains two steps: 1) find pairs of geometries from two spatial data layers whose MBRs overlap with each other, and 2) generate a grid using Quadtree partitioning and map those pairs to the grid cells. In partition-based spatial join (PBSJ), for a given number of partitions (cells), geometries from each layer is stored in all the partitions where it belongs. Spatial join is then carried out in each partition. Instead of partitioning data from each layer, in this paper, we propose to first find all the candidate pairs, and then partition the candidate pairs on a grid. The advantages of this approach is workload-aware partitioning as well as reducing the inter-process communication.

Algorithm 1 Algorithm for finding candidates

```
1: Input: Two collections of spatial objects $R$ and $S$.
2: Output: Candidate set denoted by $C$,
3: Build Rtree index $RI$ using MBRs of $R$
4: for MBR $s_j$ in $S$ do
```

```
        5: $results \leftarrow RI.query(s_j.MBR)$
        6: **for** $r_k$ in *results* **do**
                7: Find the intersection of $r_k$.*MBR* and $s_j$.*MBR*
                8: Calculate center point of intersection denoted by $p_{jk}$
                9: Calculate weight $w_{jk}$ using weight equation.
                10: $C \leftarrow C \cup tuple(r_k, s_j, p_{jk}, w_{jk})$
        11: **end for**
12: **end for**
```

A. Finding candidates for partitioning

Algorithm 1 describes the procedure used to find the pairs of cross-layer geometries which potentially intersect each other, i.e. their MBRs have overlap. The inputs to the algorithm are two collections (layers) of geometries denoted by *R* and *S.* An Rtree index is built from MBRs of *R* which helps in reducing searching time [10]. Then an R-tree query is performed using MBRs of *S* which generates a collection of candidates denoted by *C.*

Candidate set *C* is a collection of objects from layer 1 whose MBRs intersect with MBRs of objects in layer 2. For each element from this collection, we find actual MBR intersection, which can be a rectangle, line or point. We also calculate a candidate's weight based on the numbers of vertices in the two objects. The weight can be calculated based on the time complexity of the computational geometry algorithm involved in the refinement phase. Based on our previous research [20], assuming *m* and *n* are the number of vertices in two geometries, we use weight, $w = ((n + m)log(n + m))$ for finding geometric intersection. Weight and center points are attributes of each candidate as shown in Line 10 of Algorithm 1. The center points of MBR intersections are used for reference point method and output-sensitive duplication avoidance.

Reference point method:

As described earlier, geometry spanning multiple cells of a grid is duplicated in all the cells it passes through. As shown in Figure 2, a candidate (r_2, s_3) gets mapped to two cells (C,IV) and (C,V). To avoid redundant computation on the same candidate pair by two different processors in adjacent cells, reference point method is used [6]. As shown in Figure 2, this method calculates the intersection of MBRs r_2 and s_3 of a candidate and assigns it to the processor owning the cell where the center point of the MBR intersection belongs. For (r_2, s_3), the owner cell is (C,V). Since, this method works in the refinement phase, in a distributed memory implementation of PBSJ, mapping these geometries to their corresponding partitions requires data communication [4], [9], [13].

Output-sensitive Duplication Avoidance technique:

We employ the reference point method in our implementation. This method can be applied to reduce redundant storage of geometries in spatial join. Our new method takes advantage of the fact that since all candidates are known, a geometry need not be stored in all the grid cells it passes through. The storage of geometries in grid cells can be determined by the location of the candidates in the grid. We illustrate this observation using geometry r_1 that spans through multiple cells in Figure 2. The space saving is one of the advantages of considering both layers during spatial partitioning. In a filter and refine based join processing, there can be thousands/millions of candidates. Less redundancy resulting from avoiding unnecessary storage of geometries leads to reduced storage requirements as the grid cells become finer in resolution. This also applies to minimizing the communication required to move the large geometries to their corresponding cells in a distributed memory version of PBSJ.

After the filter step, we know all the candidates. Geometries from a layer that do not participate in any spatial join operations are considered to have zero-weight and thus do not impact the weight associated with a grid partition. Moreover, these geometries are not considered while mapping the pairs to the grid cells. Some of these geometries have thousands of vertices and span multiple grid cells. As such, in a distributed GIS system, where data partitioning is used, this improves the effectiveness of weight calculation and thus load balancing when grid dimensions become fine-grained.

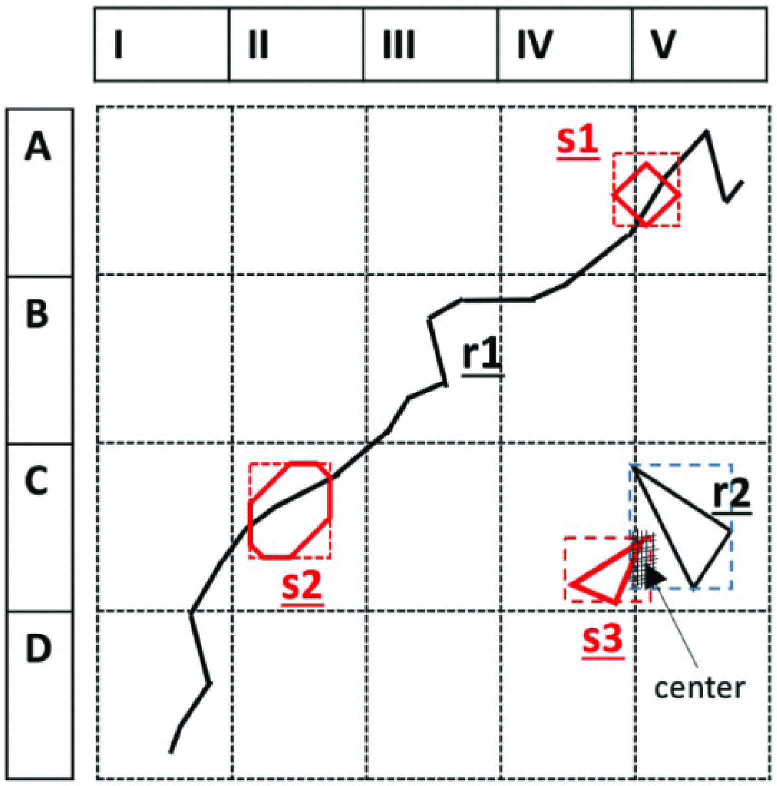


Fig. 2. Mapping of candidates to grid cells. (r_1, s_1), (r_1, s_2), (r_2, s_3) are candidates. Due to our output-sensitive method, geometry r_1 is not stored in cell ids (D, I), (C, I), (B, III), (B, IV), and (A, IV) even though it passes through these grid cells. Instead, r_1 is stored in cells (C,II) and (A,V) because it is part of two candidates (r_1, s_1) and (r_1, s_2) only.

B. Multi-threaded Partitioning of Candidates

For grid partitioning of candidates, we use each candidate's center point and weight attributes. Standard Quadtree partitioning divides a cell recursively if the number of objects in it is more than a threshold value. The goal of our parallel partitioning method is different. For a user-specified target number *N* of grid cells, the main goal is to generate *N* cells with roughly equivalent weight. Our sequential implementation of this method uses a greedy approach of selecting the cell with highest weight and generating four sub-cells. This can be implemented using a max-heap where cells are accessed in descending order of their weights. The weights of those new sub-cells are recalculated by summing the weights of all candidates within those sub-cell area. The greedy approach of first partitioning the cell with highest weight limits the concurrency to four tasks per step. Therefore, we relax this constraint by allowing multiple cells to be partitioned in parallel. However, we still want to generate grid cells that are closer to sequential implementation.

To illustrate an issue with parallelization of our Quadtree partitioning approach, here is an example. Let us consider 4 cells with weights given by an array A = {20, 15, 6, 4}. Let us assume that the first cell (A[0]) got divided into four sub-cells with weights {19, 1, 0, 0} by a thread. If another thread picks cell with weight 6 for division instead of picking cell with weight 19 and we need only eight cells as output, it is clear that we may not end up with desired output. A single-threaded execution would pick 19 before 6 because of its descending order priority. As, we can see, we do not want a lower weight cell to be considered for sub-division by a thread, if there are relatively higher weight cells still undergoing division by another thread. This decision making also depends on how many cells have already been partitioned w.r.t the target N.

Theorem 1 is used for guiding the parallelism of Quadtree partitioning in ADP and ParADP.

Theorem 1. *Assume that A is an array of cells arranged in descending order of its cell-weights. A sequential algorithm partitions cells in A by selecting a cell with the largest weight, and splitting the cell into sub-cells by dividing its weight. If $w_i \geq w_0/\kappa$, a sub-set $B = w_0, w_1, ..., w_{i-2}, w_{i-1}$ can be partitioned into at most $i*\kappa$ sub-cells whose values are $\geq w_i$, where split factor $\kappa \geq 1$.*

Proof. If $(i * \kappa) + q$ sub-cells were generated by the partitioning algorithm, whose weights $\geq w_i$, where $q > 0$, then $\sum_{n=0}^{i-1} w_n \geq ((i * \kappa) + q) * w_i \geq i * w_0 + q * w_i$. This leads to contradiction since $\sum_{n=0}^{i-1} w_n$ can at most be $i * w_0$. ■

Based on Theorem 1, we can simultaneously split cells with weights w_0 to w_i when $w_i \geq w_0/\kappa$. An algorithm can control the degree of concurrency by choosing κ. Moreover, by adjusting the value of κ, we can trade-off speed-up vs accuracy of a parallel partitioning method. Here the accuracy means the similarity of the partitioning grid produced by parallel method compared to the sequential method.

By applying Theorem 1, we have incorporated a heuristic in our OpenMP algorithm which compares the weight of a cell against the cell with the maximum weight w_{max}. A cell other than the cell with w_{max} can be partitioned if its weight is $\geq w_{max}/\kappa$, where $\kappa \geq 1$. The value of κ can be customized. With lower value of κ, the output of parallel partitioning is closer to a sequential partitioning.

Algorithm 2 describes the OpenMP based Parallel Quadtree partitioning algorithm. A user provides the desired number of partitions in the grid. C is the set of candidate pairs. Elements in C contains points with weights. The weight of a cell is the summation of the weights of candidates in that cell. Initially, G only contains an MBR, which is denoted by *GlobalMBR* in the algorithm, that covers all objects in R and S. During the execution of the algorithm, G will contain sub-cells generated at a given time and elements of G are sorted by their weights in descending order. Cells whose weights are $\geq \kappa *$ $G[0]$.*weight* are sub-divided concurrently via OpenMP tasks. The computations in Step 9 include distribution of the candidates in a cell among its sub-cells and calculating the weights of the new sub-cells.

The task number is bounded by the number of CPU cores P to achieve a balance between concurrency and partitioning quality. A list R is used to retrieve grid cells from tasks, as directly writing to G will cause a race condition. After all tasks are completed, all cells which were selected for partitioning will be removed from *G.* Next iteration begins when *G* is sorted. A threshold *T* is needed to avoid generating more sub-cells than required. Taking parallel Quadtree partitioning as an example, *T* can be set to $4 * P$ or more.

Algorithm 2 OpenMP based Quadtree Partitioning Algorithm

```
1: **Input**: Candidate collection $C$, target number of cells $N$, $GlobalMBR$, maximum OpenMP tasks $P$, number of OpenMP tasks $counter$ in queue $R$, threshold $T$
2: **Output**: A list of grid cells $G$
3: Initialize $G \leftarrow GlobalMBR$
4: Initialize $R \leftarrow \emptyset$
5: **while** number of cells less than $N - T$ **do**
    6: $counter \leftarrow 0$
    7: #pragma omp parallel num_threads($P$ )
    8: {
    9: #pragma omp single
        10: {
    11: //Only the main thread adds tasks
    12: **for** $i = 0; i < P ; i++$ **do**
        13: **if** $G[i].weight \geq G[0].weight/\kappa$ **then**
            14: $counter++$
            15: #pragma omp task
            16: $R[i] \leftarrow$ Quadtree partition on $G[i]$
        17: **end if**
    18: **end for**
        19: }//End omp single
    20: #pragma omp taskwait
    21: }// End omp parallel
    22: $G.delete(0, counter\ 1)$
    23: //In each iteration the number of tasks may vary
    24: $G \leftarrow$ all elements in $R$
    25: $R \leftarrow \emptyset$
    26: $G.sort()$ // in descending order of cell-weights
27: **end while**
28: //After loop terminates, G's size is around $N\ T + 4P$
29: Sequential Quadtree partitioning of $G$ to the size of $N$
```

SECTION IV. **Parallel Adaptive Partitioning**

In this section, we will discuss a parallel partitioning system to accelerate ADP using MPI+Threads approach. In short, we first split the candidate pairs along x-axis among compute nodes and then split those pairs along the y-axis among threads in a compute node. Finally, each thread employs ADP to further partition the grid into a user-defined number of partitions denoted by *N.*

First, we will describe how to partition a single layer of geometries using their MBRs as input in the next subsection. Then, we will discuss how to use both layers to guide parallel partitioning.

A. Parallel ADP for Distributed Memory

To speed up the partitioning algorithms, we designed a parallel partitioning algorithm, called ParADP, by using a hybrid MPI and multi-threaded implementation. MPI is only used to facilitate data communication among the compute nodes. C++ Threads are used within each multi-core node. ParADP consists of a *parallel MBR sorting phase*, a *data communication phase*, a *work distribution phase*, and a *partition phase.* Parallel Sorting by Regular Sampling (PSRS) technique [17] is used for sorting regular samples of MBRs taken from different compute nodes.

ADP for two datasets of size m and n using p nodes is shown next:

1. **Parallel Sorting Phase**: Each compute node reads m/p and n/p MBRs from the two datasets respectively. Then each node sorts the MBRs from **dataset I** by the maximum x-coordinate values. Each node chooses p regular samples and node 0 gathers all samples. Node 0 sorts all samples and chooses $p-1$ pivot values from the sorted sample list. Node 0 broadcasts all pivot values.
2. **Communication Phase**: Partition the whole world into p vertical stripes based on the pivot values and assign each node a stripe. Each node marks the MBRs meant for other $p - 1$ nodes for communication. Since each node has a fraction of the entire data, it contains MBRs that do not belong to the stripe it is assigned. So, each node sends MBRs to their corresponding nodes based on whether a given MBR overlaps with a stripe. Also, MBRs belonging to the local stripe are received from other nodes. These steps can be performed by using MPI_Send and MPI_Recv functions or using MPI_Alltoall function.
3. **In-memory Work Distribution Phase**: After communication, each node sorts the data it received by the maximum y-coordinate values of the MBRs from **dataset II.** For creating horizontal stripes, each node chooses $q - 1$ pivot values from the sorted maximum y values, where q is the number of cores in each compute node. Each node partitions its stripe into q horizontal cells and redistributes its MBRs among the respective cells. Each CPU core is assigned one cell.
4. **Partitioning Phase**: Here we use ADP algorithm as discussed earlier. Each cell c_{ij} calculates its total weight w_{ij} by adding all candidates' weights within it. Each node gathers the total weight w_i of its stripe and uses MPI_Reduce to get the total weight W for the whole dataset. Each cell generates $w_{ij} \div W \times N$ number of sub-cells using Quadtree partitioning, where N is the target number of cells required in the grid.

The grid generated by the parallel partitioning system is different from a normal Quadtree grid. An example is given in Figure 3 to show how ParADP works. Each processor in ParADP gets a unique stripe and divides the stripe further among its cores. To reach the user-defined target number of partitions, each core partitions the space within its horizontal stripe independently.

Figure 4 shows a grid with 8192 cells generated by ParADP for the *roads* and the *parks* using 32 nodes on *Bridges.* There are 32 stripes which can be distinguished by different coloring scheme.

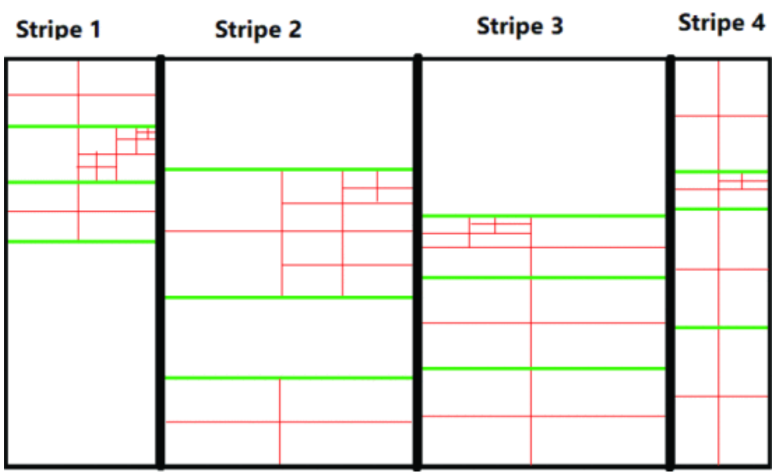


Fig. 3. ParADP using 4 compute nodes with 4 cores in each node. Longitudinal thick black lines are generated first for rearranging data based on its stripe boundary in each node. The green lines are generated by each node independently. Every CPU core/thread is assigned one cell. The thin red lines are partitioning boundaries generated by each CPU core individually.

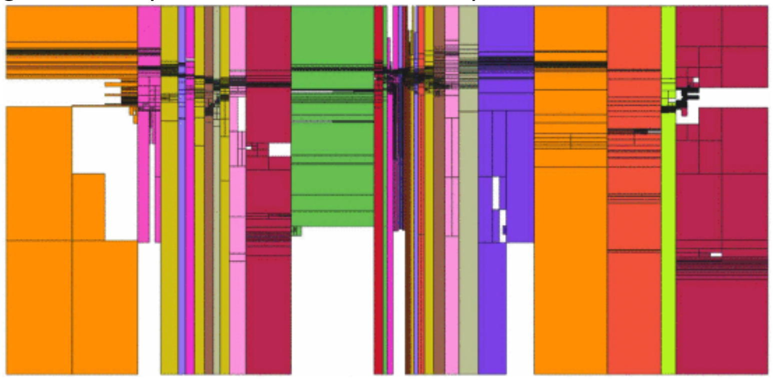


Fig. 4. Parallel partitioning of the *roads* and the *parks* into 8192 grid cells using ParADP

B. Time Complexity

Execution time breakdown:

1) In the parallel sorting phase, sorting m/p MBRs on each node takes $O((m/p) \log*(m/p))$ and communicating pivot values takes $O(p^2)$ time. 2) In the communication phase, each node gets approximately $(m+n)/p$ MBRs and sends $(m+n)*(p-1)/p$ MBRs, which takes $O(m + n)$ time. 3) In the work distribution phase, each node sorts MBRs from dataset II and divides two datasets into q subsets, which takes $O(m/p*\log (m/p)) + O(q*\log (m/p)) + O(q*\log (n/p))$ time. 4) The partition phase takes $O(m_{ij} * n_{ij})$ time, where m_{ij} and n_{ij} represent the number of MBRs in a cell from dataset I and II respectively.

Best and worst case:

ADP takes $O(mn)$ time because an MBR in dataset I can potentially overlap with all the MBRs in dataset II. However, in ParADP, MBRs from dataset I is roughly equally divided among p nodes. PSRS algorithm ensures that a processor ends up with at most $2m/p$ objects [17]. If we assume that MBRs are drawn from uniform distribution, each compute node roughly gets m/p MBRs.

In the worst case, m/p MBRs from dataset I and n/q MBRs from dataset II, are clustered in one cell (owned by a CPU core), while other cells only have MBRs from one layer only. The time complexity in the worst case is the product of the number of MBRs from I and II, i.e., $O(mn/(pq))$. Even in the worst case, ParADP is pq times faster than ADP, which is $O(mn)$.

The best case is when both datasets are uniformly distributed. In this case, each CPU core gets m/pq and n/pq MBRs from the two datasets respectively. For the best case, ParADP is $(pq)^2$ times faster than ADP.

SECTION V. Experimental Results

All of our experiments used various real world data sets, *sports*, *lakes*, *parks*, and *roads*, which are taken from Spatial-Hadoop website[4]. The attributes of the datasets are shown in Table I. ADP in this section refers to the sequential version.

TABLE I Attributes of the data sets

Name	Type	#Geometries	File size
sports	Polygons	1.8 M	590 MB
lakes	Polygons	8.4 M	9 GB
parks	Polygons	10 M	9.3 GB
Roads	Polylines	72 M	24 GB

Most of the experiments are done on a supercomputer named *Bridges*[5] at the Pittsburgh Supercomputing Center. *Bridges* has 752 regular nodes and each node has 2 Intel Haswell (E5-2695 v3, 14 cores each processor) processors running at 2.3 - 3.3 GHz with 128 GBs of memory.

In the subsections below, we have provided the storage space savings due to our duplicate avoidance technique. We performed experiments to analyze ADP's execution time. The weak scaling and strong scaling experiments are designed for testing the scalability of ParADP. The partition qualities of ADP, ParADP, Quadtree Partitioning, and Uniform Partitioning were compared.

A. Performance of Output-sensitive Duplication Avoidance

Three pairs of real world data sets were used: 1) *lakes* and *sports*, 2) *roads* and *sports*, and 3) *roads* and *lakes.* By storing in Well Known Text (*WKT*) format, *sports*, *lakes* and *roads* take 24 GB, 9 GB, 590 MB disk space respectively. They were partitioned into 1024, 2048, 4096, 8192, 16384 parts using three techniques: ADP, Quadtree, and Uniform grid. Then the geometries in each grid cell were stored separately in a file and written to hard disk in *WKT* format.

To evaluate the performance of our new duplication avoidance technique on the pre-processing stage of spatial join, ADP, Quadtree, and Uniform partitioning were used on different pairs of datasets. Their outputs were stored in a hard disk. We applied output-sensitive duplication avoidance technique in ADP only to compare the improvement in space complexity. In Figure 5, in all situations, ADP generates less data than Quadtree and Uniform partitioning. In the case of partitioning *sports* and *lakes* into 1024 cells, the total size of files generated by ADP only use around 10% of the disk space used by data generated by Quadtree and Uniform partitioning.

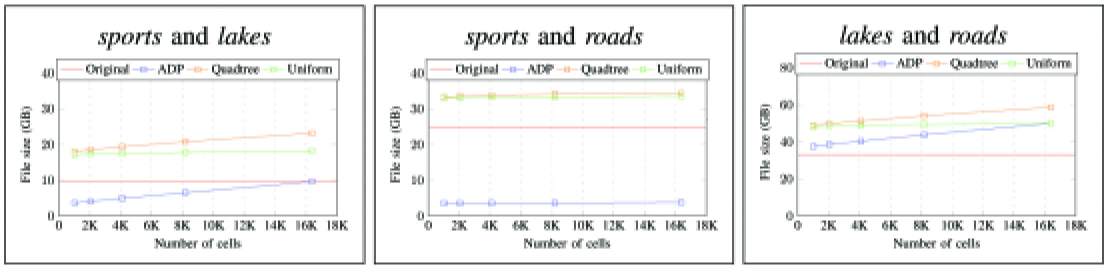


Fig. 5. Storage space needed using different partitioning techniques

When one dataset is much smaller than the other one, the number of candidate pairs may be smaller than the number of geometries in the two datasets. ADP can take this advantage and save disk space when the partitions are written to disk. This also means less communication for an in-memory distributed PBSJ algorithm. When the number of candidate pairs generated is high, such as 188 million, for *lakes* and *roads*, ADP may use more disk space than the original files. However, ADP still uses less space than Quadtree Partitioning because ADP stores geometries that are part of the candidate pairs only. As Quadtree partitioning generates more cells in areas with high density than Uniform grid, there is a higher chance of geometries being duplicated in Quadtree partitioning, which results in higher disk space consumption.

B. OpenMP Quadtree Partitioning Speedup

To evaluate the performance of the OpenMP based Quadtree partitioning, we used 3.3 million points which are the center points of a candidate's MBR intersections from *lakes* and *sports* generated by Algorithm 1. Several experiments were done by using different number of CPU cores. The value of κ is set to 2 and the threshold is set to $4P$, where P is the number of available CPU cores.

As shown in Table II, the benefits of OpenMP parallelization is realized for higher number of cells. For 32K cells, OpenMP based Quadtree partitioning achieved its highest speedup of 2.63 compared to the sequential Quadtree partitioning. Performance is impacted by the lack of scalable multi-threaded R-tree library that we use internally in Line 9 of Algorithm 2.

TABLE II OpenMP based Quadtree Partitioning time

Cores/Cells	1024	2048	4096	8192	16384	32768
1	16.03	18.33	21.09	27.10	42.43	99.39
4	16.17	18.43	21.06	26.18	35.21	62.09
8	16.06	18.31	21.57	26.54	33.42	51.87
16	15.99	18.26	20.54	24.66	30.84	50.22
32	15.41	17.79	19.93	24.04	28.89	38.05

C. Computing cost for ADP

We designed several experiments to test the impact on ADP quality using different partition numbers. *Parks* and *sports* data are partitioned into 1024, 2048, 4096, 8192, 16384 cells for five experiments respectively. For all cases, three nodes (84 cores) on *Bridges* are used. As shown in Figure 6, with a higher number of partitions, the gap between maximum and minimum MPI process execution times narrows for GEOS *Intersects* method. This demonstrates that load-balancing improves for a higher number of partitions for ADP.

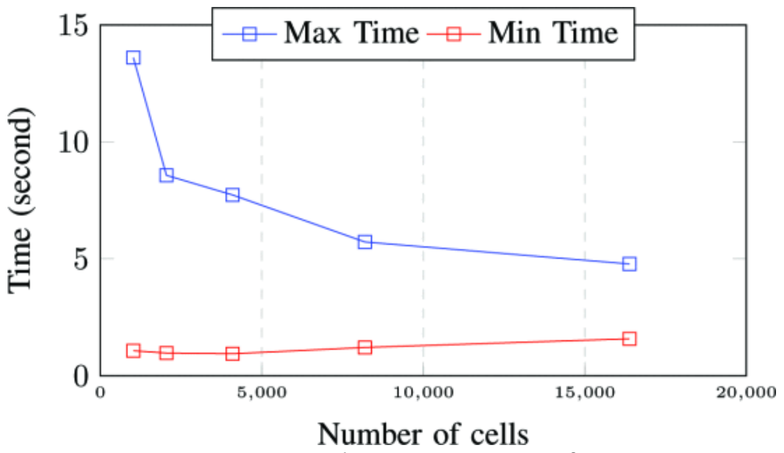


Fig. 6. Max process time and min process time for GEOS *Intersects* method using *parks* and *sports* using increasing number of grid cells generated by ADP. 84 cores are used for all cases.

Partitioning cost is determined not only by the number of objects in the two layers but also by the number of candidate pairs found during the filtering phase. The number of candidates found in the filtering phase using Algorithm 1 are as follows:

1. *parks* and *sports* is about 2.7 millions.
2. *roads* and *parks* is about 8.1 millions.

TABLE III ADP execution time for different pairs of datasets

R	*S*	Partition Number	$T_{alg_1}(s)$	$T_{quad}(s)$
quad	*sports*	8192	835.69	28.43
parks	*sports*	16384	828.93	43.10
parks	*sports*	32768	853.57	99.32
parks	*sports*	65536	859.63	346.73
roads	*parks*	8192	19714.0	1222.69
roads	*parks*	16384	19193.3	1562.82
roads	*parks*	32768	19972.4	1975.74
roads	*parks*	65536	19829.0	2757.72

Table III shows the time it takes for ADP on different pairs of datasets to generate a given number of partitions. T_{alg1} is the time used for finding the candidates. For the same pair of datasets, T_{alg1} doesn't change as the number of candidates doesn't change much when the partition number changes. T_{quad} is the time for implementing Quadtree partitioning.

When the size of the two datasets increases, the time for finding candidates pairs grows faster than the Quadtree partitioning time. This is shown in Table III. Partition based spatial join is affected by data partitioning [9] cost. From Table III, we can see that partitioning can take a lot of time when the target partition number is large for bigger datasets. Even though ADP is effective as we saw earlier, it is time-consuming. This motivates the need for parallel partitioning.

D. Weak scaling for ParADP

Here we discuss weak scaling experiments for ParADP. We generate new pairs of datasets by duplicating geometries in *parks.* When the number of compute nodes increases by 16, one duplication of *parks* is added to the workload and the number of cells in the target grid increases by 8192. As shown in Table IV, ParADP has good weak scaling.

TABLE IV ParADP execution time for weak scaling

R	S	Candidates	Nodes	Grid cells	$T_{total}(s)$
roads	parks	75 M	16	8192	49.32
roads	2*parks	150 M	32	16384	38.63
roads	3*parks	225 M	48	24576	36.89
roads	4*parks	300 M	64	32768	41.45
roads	5*parks	375 M	80	40960	40.13
roads	6*parks	450 M	96	49152	32.26
roads	7*parks	525 M	112	57344	30.77
roads	8*parks	600 M	128	65536	31.70
roads	9*parks	675 M	144	73728	30.56

E. Strong scaling for ParADP

For strong scaling experiments, *roads* and *parks* are used. Nine experiments are performed with 16, 32, 48, 64, 80, 96, 112, 128, 144 nodes on *Bridges.* Each node on *Bridges* has 28 cores.

TABLE V ParaDP execution time for strong scaling

R	S	Partition Number	Nodes	$T_{total}(s)$	$T_s(s)$
roads	parks	65536	16	55.56	1.82
roads	parks	65536	32	24.69	0.73
roads	parks	65536	48	19.32	0.64
roads	parks	65536	64	13.80	0.38
roads	parks	65536	80	11.98	0.32
roads	parks	65536	96	10.13	0.15
roads	parks	65536	112	9.64	0.15
roads	parks	65536	128	7.46	0.14
roads	parks	65536	144	6.84	0.14

In Table V, T_{total} stands for the total time for ParADP and T_s stands for the time for parallel sorting step. In all instances, ParADP has high speedups as shown in Figure 7. ParADP has high efficiency which ranges from 0.84 to 1.02, where the highest efficiency of 1.02 is achieved with 32 nodes. The reasons that the efficiency is greater than 1 are that 1) with data decomposition for both layers, the query range for every geometry is sharply reduced; 2) within a certain number of nodes, the parallel sorting time decreases with more nodes as *R* doesn't change.

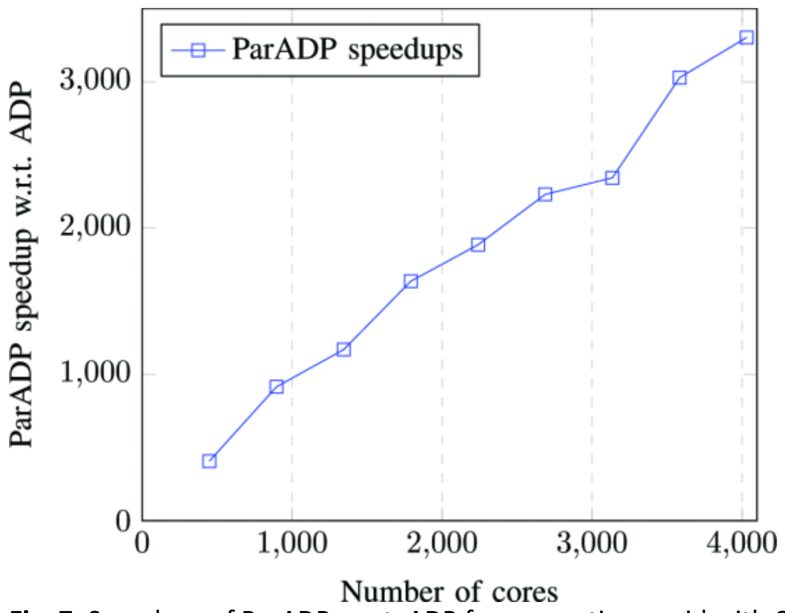


Fig. 7. Speedups of ParADP w.r.t. ADP for generating a grid with 65536 cells using two datasets - 1) *roads* (72 million polylines) and 2) *parks* (10 million polygons).

F. Partition Quality

We compare the partition qualities between ADP, Quadtree partitioning, and Uniform partitioning. For the partitioned data, we have implemented refinement phase using 1) GEOS *Intersects* method and 2) GEOS *Intersection. Intersection* method takes more time than *Intersects* method because the output geometry needs to be computed for *Intersection.* Round-robin scheduling of partitions/cells to MPI processes is carried out. Static scheduling captures the partition quality for a given partitioning technique. Maximum and minimum execution time is reported. The maximum time taken by a thread/process determines the end-to-end time.

Figure 8 shows the comparison of execution time for ADP, Quadtree Partitioning, and Uniform Partitioning. As shown in the figure, with more MPI processes involved, the average execution times decrease. However, using Uniform partitioning, the maximum MPI process execution time doesn't change much; using Quadtree partitioning, the overall maximum MPI process execution time changes slightly but in some cases it increases. Since the execution time of a parallel application is decided by the thread taking the longest time (straggler effect), using ADP minimizes the overall execution time.

Figure 9 shows the timing for the refinement phase using GEOS *Intersection* method. We compare the partition quality using ADP, Quadtree partitioning, and ParADP. Figure 9(a) shows the maximum (max) and minimum (min) MPI process times when a MPI_GIS implementation applied *Intersection* on the partitioned *parks* and *sports.* Both data are partitioned into 8192 parts. As shown in the figure, the max process times for ParADP are much lower than the max process times for Quadtree partitioning. The min process times for ParADP are higher than the min process times for Quadtree partitioning. The MPI process times for ParADP-based partitioned data are in a much narrower range than the MPI process times for Quadtree-based partitioned data. Figure 9(b) shows the maximum (max) and minimum (min) MPI process times when *Intersects* operation is applied on the partitioned *parks* and *sports.* ParADP shows improvement over Quadtree partitioning in both experiments.

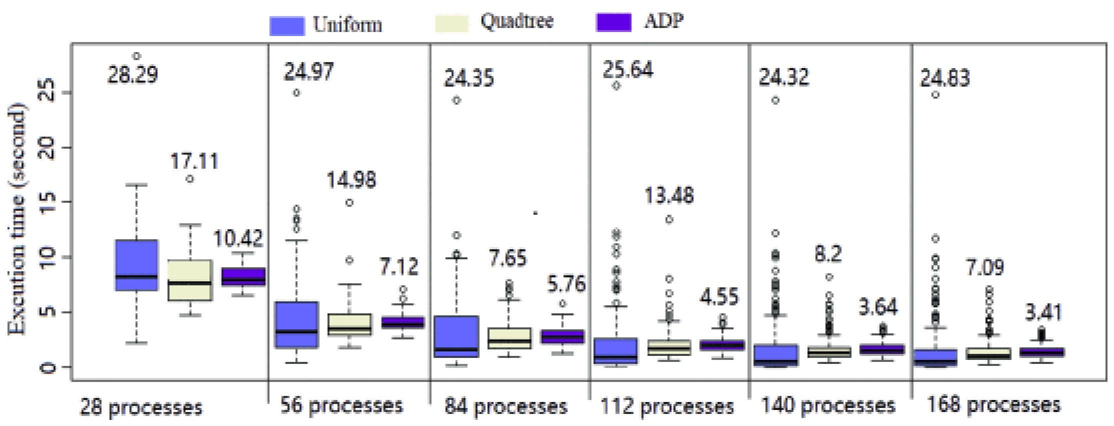
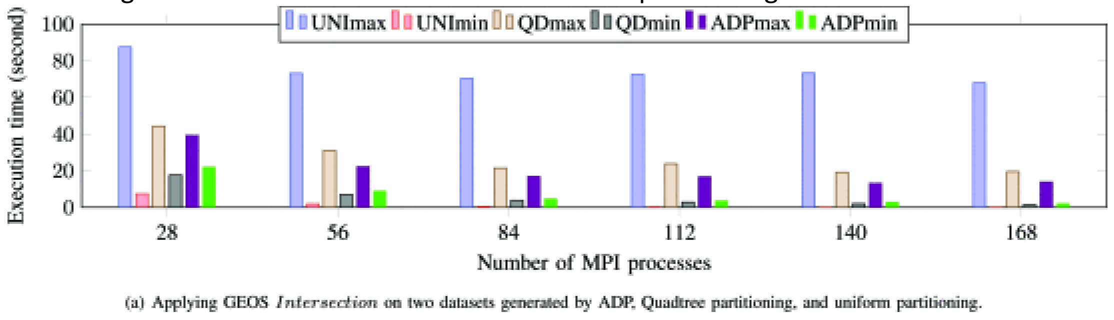


Fig. 8. Box-plot showing distribution of execution time by different MPI processes running GEOS *Intersects* query using *parks* and the *sports* data. Each time the data sets are partitioned into 8192 parts. Max process execution time along with few outliers are also shown for each partitioning scheme.



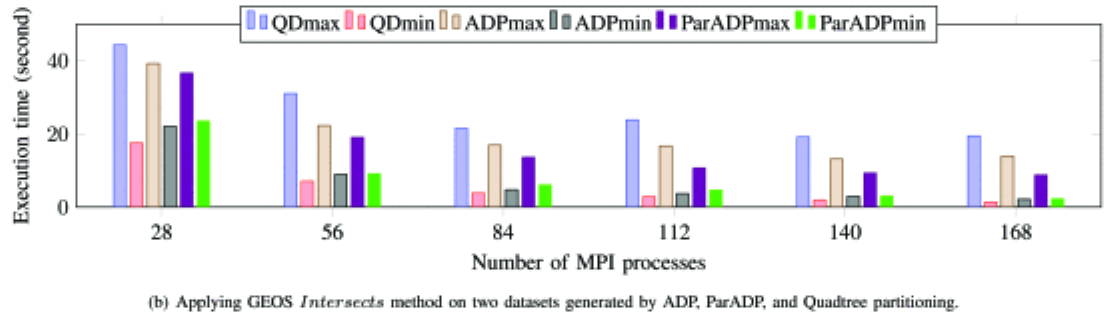


Fig. 9. Max and min process execution times. Datasets *parks* and *sports* were used and partitioned into 8192 parts.

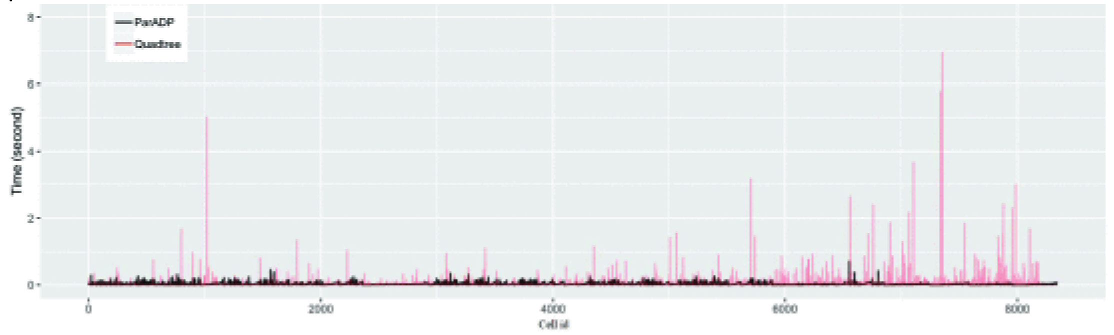


Fig. 10. Execution time of applying *Intersects* on different cells of the partitioned *parks* and *sports.* The data sets are partitioned into 8192 cells by ParADP and Quadtree partitioning.

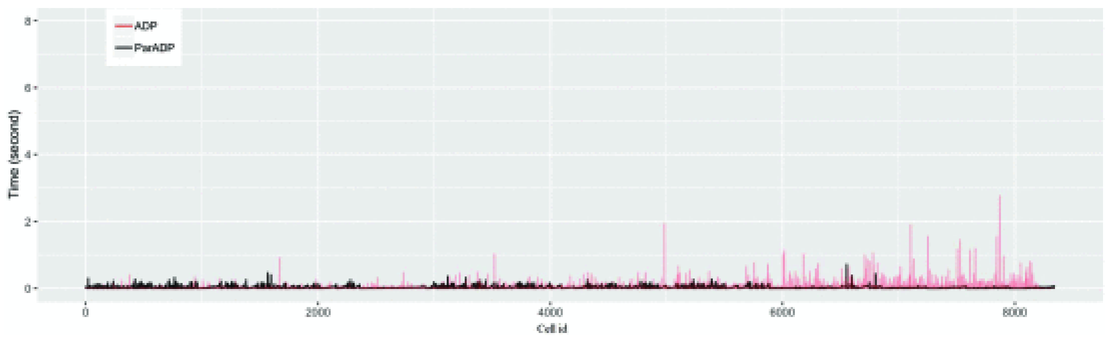


Fig. 11. Execution time of applying *Intersects* on different cells of the partitioned *parks* and *sports* data. The data sets are partitioned into 8192 cells by ParADP and ADP.

The *Intersects* execution times for processing each cell of partitioned *parks* and *sports* are shown in Figure 10. The *parks* and *sports* are partitioned to 8192 cells by ParADP and Quadtree partitioning. As we can see, the *Intersects* execution times for processing ParADP-based partitioned cells are within a narrow range and none of them exceed 0.8 second. On the other hand, the *Intersects* execution times for processing Quadtree-based partitioned cells have higher variation and the longest execution time taken is 7 seconds. If we consider a large scale HPC system with as many CPU cores as the number of partitions and each CPU core is assigned data in a pair of cells, spatial join can be done within 0.8 second using ParADP partitioned cells while 7 seconds are needed to process Quadtree-based partitioned cells.

The *Intersects* execution times for processing each cell of partitioned *parks* and *sports* are shown in Figure 11. The *parks* and *sports* are partitioned into 8192 cells by ADP and ParADP. Cells with higher cell id take a longer time in ADP. This is because cells with higher id have larger weight. Compared to ADP, ParADP shows a narrower process execution time range and a lower value for maximum MPI process execution time.

ParADP shows better partition quality than Quadtree partitioning. Once the candidate pairs are partitioned among the CPU cores, ParADP internally calls ADP. In this way, ParADP method can exploit parallelism in adaptively partitioning the workload. The above-mentioned experimental results prove the benefit of partitioning workload by considering both layers versus partitioning data in a layer by layer basis.

For load balancing spatial computations, an alternative approach is to start with a grid that is based on a single layer (dataset) and dynamically rebalance the workload in cells that have higher workload. In a distributed memory environment, this leads to the movement of complex geometries from an MPI process with higher workload to another MPI process with lower workload. Moreover, there is overhead involved in serializing, deserializing and parsing the geometries due to communication. This is based on our prior experience of parallelizing spatial join with ADLB library for load balancing. The size of individual geometries varies from few KB to 10 MB. Therefore, the cost of dynamic load balancing while running partition-based spatial join is quite high. Thus, we explored the feasibility of generating a grid with user-specified number of partitions in this paper.

SECTION VI. **Conclusion**

In this paper, we proposed Adaptive Partitioning techniques. ADP can partition spatial data like polygons and polylines in a load-balanced fashion. We have presented experiments on various real-world data sets and evaluated the partition quality between ADP and two classic partitioning techniques, Quadtree partitioning, and Uniform partitioning. A new duplication avoidance technique is introduced by which unnecessary duplication of geometries spanning multiple grid cells is reduced. An OpenMP version of ADP was also presented.

We have also designed a parallel partitioning system. Parallel ADP can partition large real-world spatial datasets with data skew in a shorter time. ParADP algorithm has been shown to be scalable on thousands of CPU cores. ParADP shows better partition quality than ADP and Quadtree-based partitioning. The weak scaling and strong scaling experiments prove that ParADP has good scalability and improves performance with increase in the size of compute cluster up to 4032 CPU cores.

ACKNOWLEDGMENT

This work is partly supported by the National Science Foundation CRII Grant No.1756000 and the Northwestern Mutual Data Science Institute. This work used the NSF Extreme Science and Engineering Discovery Environment (XSEDE), which is supported by ACI-1548562.

References

1.Y. Liu, J. Yang and S. Puri, "Hierarchical Filter and Refinement System Over Large Polygonal Datasets on CPU-GPU", *2019 IEEE 26th International Conference on High Performance Computing Data and Analytics (HiPC)*, pp. 141-151, 2019.

2.Swarup Acharya, Viswanath Poosala and Sridhar Ramaswamy, "Selectivity estimation in spatial databases", *Proceedings of the 1999 ACM SIGMOD international conference on Management of data (SIGMOD '99)*, pp. 13-24, 1999.

3.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", *2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum*, pp. 1433-1439, 2012.

4.Ablimit Aji, Fusheng Wang, Hoang Vo, Rubao Lee, Qiaoling Liu, Xiaodong Zhang, et al., "Hadoop GIS: a high performance spatial data warehousing system over mapreduce", *Proc. VLDB Endow*, vol. 6, no. 11, pp. 1009-1020, August 2013.

5.M. Deveci, S. Rajamanickam, K. D. Devine and U. V. Catalyurek, "Multi-Jagged: A Scalable Parallel Spatial Partitioning Algorithm", *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, no. 3, pp. 803-817, March 2016.

6.J. Dittrich and B. Seeger, "Data redundancy and duplicate detection in spatial join processing", *Proceedings of 16th International Conference on Data Engineering (Cat. No.00CB37073)*, pp. 535-546, 2000.

7.Ahmed Eldawy, Louai Alarabi and Mohamed F. Mokbel, "Spatial partitioning techniques in SpatialHadoop", *Proc. VLDB Endow*, vol. 8, no. 12, pp. 1602-1605, August 2015.

8.Wm Randolph Franklin, Chandrasekhar Narayanaswaml, Mohan Kankanhalll, David Sun, Meng-Chu Zhou and Peter YF Wu, "Uniform grids: A technique for intersection detection on serial and parallel machines", *Proceedings of Auto-Carto*, vol. 9, 1989.

9.Edwin H. Jacox and Hanan Samet, "Spatial join techniques", *ACM Trans. Database Syst*, vol. 32, no. 1, March 2007.

10. Ming-Ling Lo and Chinya V. Ravishankar, "Spatial joins using seeded trees", *Proceedings of the 1994 ACM SIGMOD international conference on Management of data (SIGMOD '94)*, pp. 209-220.

11. Sushil K Prasad, Danial Aghajarian, Michael McDermott, Dhara Shah, Mohamed Mokbel, Satish Puri, Sergio J Rey, Shashi Shekhar, Yiqun Xe, Ranga Raju Vat-savai et al., "Parallel processing over spatial-temporal datasets from geo bio climate and social science communities: A research roadmap", *2017 IEEE International Congress on Big Data (BigData Congress)*, pp. 232-250, 2017.

12. Satish Puri, D. Agarwal, X. He and S. K. Prasad, "MapReduce Algorithms for GIS Polygonal Overlay Processing", *2013 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum*, pp. 1009-1016, 2013.

13. Satish Puri, Anmol Paudel and Sushil K Prasad, "MPI-Vector-IO: Parallel I/O and partitioning for geospatial vector data", *Proceedings of the 47th International Conference on Parallel Processing ICPP*, pp. 13, 2018.

14. Satish Puri and Sushil K Prasad, "A parallel algorithm for clipping polygons with improved bounds and a distributed overlay processing system usingmpi", *2015 15th IEEE/ACM International Symposium on Cluster Cloud and Grid Computing*, pp. 576-585, 2015.

15. Suprio Ray, Bogdan Simion, Angela Demke Brown and Ryan Johnson, "Skew-resistant parallel in-memory spatial join", *Proceedings of the 26th International Conference on Scientific and Statistical Database Management*, pp. 6, 2014.

16. Shashi Shekhar, Sivakumar Ravada, Vipin Kumar, Douglas Chubb and Greg Turner, "Load-balancing in high performance GIS: Declustering polygonal maps" in International Symposium on Spatial Databases, Springer, pp. 196-215, 1995.

17. Hanmao Shi and Jonathan Schaeffer, "Parallel sorting by regular sampling", *Journal of parallel and distributed computing*, vol. 4, no. 1992, pp. 361-372, 1992.

18. Edgar Solomonik and Laxmikant V Kale, "Highly scalable parallel sorting", *2010 IEEE International Symposium on Parallel & Distributed Processing (IPDPS)*, pp. 1-12, 2010.

19. Xiaofang Zhou, David J Abel and David Truffet, "Data partitioning for parallel spatial join processing", *Geoinformatica2*, vol. 2, no. 1998, pp. 175-204, 1998.

20. Jie Yang, Anmol Paudel and Satish Puri, "Spatial Data Decomposition and Load Balancing on HPC Platforms", *Proceedings of the Practice and Experience in Advanced Research Computing on Rise of the Machines (learning) (PEARC '19)*, pp. 4, 2019.

21.S. Shohdy, Y. Su and G. Agrawal, "Load Balancing and Accelerating Parallel Spatial Join Operations Using Bitmap Indexing", *2015 IEEE 22nd International Conference on High Performance Computing (HiPC)*, pp. 396-405, 2015.

22. Raphael A. Finkel and Jon Louis Bentley, "Quad trees a data structure for retrieval on composite keys", pp. 1-9, 1974.

23. Anmol Paudel and Satish Puri, "OpenACC Based GPU Parallelization of Plane Sweep Algorithm for Geometric Intersection. Accelerator Programming Using Directives 2018" in Lecture Notes in Computer Science, Springer, vol. 11381.

24. Satish Puri and Sushil K. Prasad, "Efficient Parallel and Distributed Algorithms for GIS Polygonal Overlay Processing", *2013 IEEE International Symposium on Parallel & Distributed Processing Workshops and Phd Forum*, vol. 2013, pp. 2238-2241, 2013.

25. Satish Puri and Sushil K. Prasad, "Output-Sensitive Parallel Algorithm for Polygon Clipping", *2014 43rd International Conference on Parallel Processing*, pp. 241-250, 2014.

26. Danial Aghajarian, Satish Puri and Sushil 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*, pp. 1-10, 2016.