Efficient Parallel Adaptive Partitioning for Load-balancing in Spatial Join

Jie Yang  
Computer Science Department  
Marquette University  
Milwaukee, USA  
jie.yang@marquette.edu

Satish Puri  
Computer Science Department  
Marquette University  
Milwaukee, USA  
sathish.puri@marquette.edu

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 and skewed 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.

In this work, we present Adaptive Partitioning (ADP) technique, which is based on Quadtree partitioning. Unlike most recent partitioning techniques, ADP partitions the workload instead of partitioning the individual datasets separately. According to the experiments, ADP partitions spatial data in a more balanced way than Quadtree partitioning and Uniform grid partitioning. ADP uses OpenMP tasks to accelerate the Quadtree partitioning process. ADP takes an output-sensitive duplication avoidance technique which minimizes duplication of geometries that do not participate in spatial join queries. 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 data sets, one with 717 million polylines and another with 10 millions polygons, is partitioned into 65536 grid cells within 7 seconds. ParADP performs well with both good weak scaling up to 4032 CPU cores and good strong scaling up to 4032 CPU cores.

Index Terms—spatial join, spatial data partitioning, load-balancing, Adaptive Partitioning, Parallel Partitioning, MPI-GIS, HPC

I. INTRODUCTION

With the increasing volume and complexity of spatial data, there is an increasing demand for efficient geospatial techniques for parallelizing spatial computations [10]. 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 geometry 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 [8]. A spatial join can be divided into two phases: 1) filter phase, which removes irrelevant data and 2) refinement phase, which computes join results [8]. The irrelevant data can be found if the minimum bounding rectangles (MBR) of geometries from two layers do not intersect with each other.

In our prior work, we developed MPI-GIS for partition-based polygon overlay and spatial join computation [13]. MPI-Vector-IO is a component of MPI-GIS that performs parallel I/O of spatial data stored in parallel filesystem [12]. 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.

Many existing spatial data partitioning techniques are based on one layer, which ignore the distribution of data in another layer. As shown in Figure 1, traditional partitioning algorithms 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. As the workload distribution is shown in Output C, the maximum workload is present in the fourth cell of grid C, even though the fourth cell in grid A and B do not have the maximum geometries in their respective grids. 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 fewer variations in workload. We compare our new algorithm’s performance in terms of partition quality with Quadtree-based partitioning. Together with ADP, an OpenMP based Quadtree partitioning approach is introduced.

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.

Source code and more results for this paper can be found at https://www.mscs.mu.edu/~satish/mpiacgis.html
Moreover, we propose a parallel adaptive partitioning algorithm (ParADP) for High-Performance Computing (HPC) environment using MPI and C++ threads. 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 it belongs. 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.

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 three classic partitioning techniques, Quadtree partitioning, and Uniform partitioning. Partition quality of ParADP algorithm is examined by comparing it with ADP and Quadtree partitioning. Several weak scaling and strong scaling experiments are designed for ParADP and it shows good scalability.

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 geometric data. Message Passing Interface (MPI) is also used for communication between computing nodes.

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.

II. Background and Related Work

Partitioning spatial data has been well-studied in literature. Equi-Partitioning, Min-Skew [1], 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 [4]. 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 [7], [12], [14], [18]. SpatialHadoop supports different partitioning schemes using techniques based on Quadtree, R-tree, grid, etc in a MapReduce environment [6]. Our parallel partitioning algorithm uses MPI.

MPI-based GIS system: In our prior work, we experimented with MPI-based approaches [2], [11], [13] and developed parallel I/O and partitioning framework called MPI-Vector-IO as an HPC system [12]. 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 and Hadoop-GIS use dynamic load balancing present in Apache Hadoop framework [3], [6], [20]. 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 [19].

Data skew results in load imbalance. To mitigate the effects of load imbalance, SPINOJA system [14] 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 [14], [15]. The experimental evaluation in [14] was done on a single processor.

\(^1\)http://trac.osgeo.org/geos

Fig. 1: #Geometries shown in each cell of grids A and B. The workload of a cell in grid C is the product of the #geometries present in corresponding cells in A and B.
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.

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.

Reference point method: As described earlier, geometry spanning multiple cells of a grid is duplicated in all the cells it passes through. A candidate pair (e.g., \((r_2, s_3)\) and \((r_1, s_1)\)) can be mapped to more than one cell. To avoid redundant computation on the same candidate pair by two different processors in more than one cell, reference point method is used [5]. As shown in Figure 2, this method calculates the intersection of MBRs \(r_2\) and \(s_3\) of a candidate pair and assigns the pair to the processor owning the cell where the center point of the MBR intersection belongs. 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 [3], [8], [12].

Output-sensitive Duplication Avoidance technique: We employ reference method in our implementation. This method can be further improved. Our new method takes advantage of the fact that since all candidate pairs are known, a geometry need not be stored in all the grid cells it passes through. We only need to partition the candidates. 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

\[\text{Algorithm 1 Algorithm for finding candidate pairs}\]

1. **Input**: Two sets of spatial objects \(R\) and \(S\).
2. **Output**: Candidate pairs denoted by \(C\), and their intersections \(I\).
3. Build Rtree index \(RI\) using MBRs of \(R\).
4. for MBR \(s_j\) in \(S\) do
   5. \(\text{results} \leftarrow \text{RI}.query(s_j.MBR)\)
   6. for \(r_k\) in \(\text{results}\) do
      7. \(\text{Find the intersection of } r_k.MBR \text{ and } s_j.MBR\)
      8. Calculate center point of intersection denoted by \(p_{jk}\).
      9. Calculate weight \(w_{jk}\) using Equation 1.
     10. \(C \leftarrow C \cup \text{pair}(p_{jk}, w_{jk})\)
     11. \(I \leftarrow I \cup p_{jk}\)
   12. end for
5. end for

Algorithm 1 describes the procedure used to find the pairs of geometries which potentially intersect each other, i.e. their MBRs have overlap. The inputs to the algorithm are two lists of geometries denoted by \(R\) and \(S\). In the algorithm, weight refers to the number of vertices in each geometry. An Rtree index is built from MBRs of \(R\) which helps reducing searching time [9]. Then an R-tree query is performed using MBRs of \(S\). Algorithm 1 can be parallized by using OpenMP in the inner loop from line 6 to line 12.
C is a list of pairs of objects from layer 1 whose MBRs intersect with MBRs of objects in layer 2. For each pair from this list, we find an actual MBR intersection, which can be a rectangle, line or point. We also calculate its 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. The central points of those MBR intersections are stored in I, which is used for Output-sensitive Duplication Avoidance.

Modeling complexity of spatial computations in the refine phase: Based on our previous research [19], 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.

After the filter step, we know all the candidate pairs and their intersections. 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.

Goal of parallel partitioning: Standard Quadtree partitioning divides a cell recursively if the number of points 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 sorted in descending order of their weights. The weights of those new sub-cells are recalculated by summing the weights of all candidates within 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.

B. Adaptive Partitioning (OpenMP)

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 \( N \).

To enforce this constraint, we have incorporated a heuristic in our OpenMP algorithm which compares the weight of a cell against the cell with the maximum weight \( w_{\text{max}} \). A cell other than the cell with \( w_{\text{max}} \) can be partitioned if its weight is \( \geq w_{\text{max}}/\kappa \), where \( \kappa \geq 1 \). The value of \( \kappa \) can be customized, but with lower value of \( \kappa \), the output of a parallel Quadtree partitioning is closer to a sequential Quadtree 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 \( \text{GlobalMBR} \) 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 \( \geq \kappa * \text{GlobalMBR}.weight \) are sub-divided concurrently via OpenMP tasks.

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 results from tasks, as directly writing results to \( G \) will cause a race condition. After all tasks are completed, all cells from \( G[0] \) to \( G[\text{numTasks} - 1] \) that were partitioned in the current iteration are deleted from \( G \). Next iteration begins when \( G \) is sorted. A threshold \( T \) is needed to avoid generating more sub-cells than required. \( T \) can be set to \( 4 * \text{numTasks} \) or more.

Algorithm 2 OpenMP based Quadtree Partitioning Algorithm

<table>
<thead>
<tr>
<th>Line</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Input: ( C ), target number of cells ( N ), ( \text{GlobalMBR} ), maximum OpenMP tasks ( P ), number of OpenMP tasks ( \text{numTasks} \leq P ), threshold ( \kappa )</td>
</tr>
<tr>
<td>2</td>
<td>Output: A list of grid cells ( G )</td>
</tr>
<tr>
<td>3</td>
<td>Initialize ( G \leftarrow \text{GlobalMBR} )</td>
</tr>
<tr>
<td>4</td>
<td>Initialize ( R \leftarrow \emptyset )</td>
</tr>
<tr>
<td>5</td>
<td>while number of cells less than ( N - T ) do</td>
</tr>
<tr>
<td>6</td>
<td>for ( i = 0; i &lt; P; i++ ) do</td>
</tr>
<tr>
<td>7</td>
<td>if ( G[i].\text{weight} \geq G[0].\text{weight} / \kappa ) then</td>
</tr>
<tr>
<td>8</td>
<td>#pragma omp task</td>
</tr>
<tr>
<td>9</td>
<td>( R[i] \leftarrow \text{Quadtree partition on } G[i] )</td>
</tr>
<tr>
<td>10</td>
<td>end if</td>
</tr>
<tr>
<td>11</td>
<td>end for</td>
</tr>
<tr>
<td>12</td>
<td>#pragma omp taskwait</td>
</tr>
<tr>
<td>13</td>
<td>( G . \text{delete}(0, \text{numTasks} - 1) )</td>
</tr>
<tr>
<td>14</td>
<td>// In each iteration ( \text{numTasks} ) may vary</td>
</tr>
<tr>
<td>15</td>
<td>( G \leftarrow \text{all elements in } R )</td>
</tr>
<tr>
<td>16</td>
<td>( R \leftarrow \emptyset )</td>
</tr>
<tr>
<td>17</td>
<td>( G . \text{sort()} ) // in descending order</td>
</tr>
<tr>
<td>18</td>
<td>end while</td>
</tr>
<tr>
<td>19</td>
<td>// After loop, ( G )’s size is around ( N - T + 4P )</td>
</tr>
<tr>
<td>20</td>
<td>Sequential Quadtree partition ( G ) to the size of ( n )</td>
</tr>
</tbody>
</table>
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 T.

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 [16] 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 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_Absoft 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 re-arranges its MBRs to the respective cells. Assign each CPU core one cell.

4) Partitioning Phase: Here we use ADP algorithm as discussed earlier. Each cell cij calculates its total weight wi j by adding all candidates weights within it. Each node gathers the total weight wi of its stripe and uses MPI_Reduce to get the total weight W for the whole dataset. Each cell generates wij ÷ W × T number of sub-cells using Quadtree partitioning, where T is the number of cells in the target 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.

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); 2)
in the communication phase, each node gets approximately \((m + n)/p\) MBRs and sends \((m + n)/(p-1)/p\) MBRs, which takes \(O(m + n)\); 3) in the in-memory work distribution phase, each node sorts MBRs from dataset II and divides two datasets into \(q\) subsets, which takes \(O(m/p)\log (m/p)) + O(q\log (m/p)) + O(q\log (n/p)) = O(nlog m + log n)\); 4) the partition phase takes \(O(m_{ij}n_{ij})\), 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 [16]. 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., \(O(mn/\lceil pq \rceil)\). 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 \(n/pq\) from the two datasets respectively. For the best case, ParADP is \((pq)^2\) times faster than ADP.

**V. EXPERIMENTAL RESULTS**

All of our experiments use various real world data sets, sports, lakes, parks, and roads, which are taken from SpatialHadoop website\(^4\) and extracted from https://www.openstreetmap.org. The attributes of those five data sets are shown in Table I. ADP in this section refers to the sequential version.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Geometry #</th>
<th>File size</th>
</tr>
</thead>
<tbody>
<tr>
<td>lakes</td>
<td>Polygons</td>
<td>8.4 M</td>
<td>9 GB</td>
</tr>
<tr>
<td>roads</td>
<td>Polygons</td>
<td>10 M</td>
<td>9.3 GB</td>
</tr>
<tr>
<td>Roads</td>
<td>Polygons</td>
<td>72 M</td>
<td>24 GB</td>
</tr>
<tr>
<td>Sports</td>
<td>Polygons</td>
<td>1.8 M</td>
<td>590 MB</td>
</tr>
</tbody>
</table>

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.

To evaluate the performance of the OpenMP based Quadtree partitioning, the number of available processors were changed when running the partition algorithm on 3.3 million points. The points were found by using Algorithm 1 on sports and lakes. 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.

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.

We performed experiments on ADP’s execution time. 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.

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 was stored separately in a file and written to hard disk in WKT format.

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 experiments of using sports and lakes, ADP only generated 10% of the data generated by Quadtree and Uniform partitioning.

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 lots of candidate pairs are found, such as 188 million candidate pairs for lakes and roads, ADP may use more disk space than the original files. However, ADP still uses less space than Quadtree Partitioning as ADP’s output can be treated as a Quadtree partitioning output which removed all unnecessary data. As Quadtree partitioning generates more cells in areas with high density than Uniform, there is a higher chance of geometries being duplicated in Quadtree partitioning, which results in higher disk space consumption.

B. OpenMP Quadtree Partitioning Speedup

We used 3.3 million points which are the centre points of a candidate’s MBR intersections from lakes and sports. Several experiments were done by different limits of available CPU cores. The value of \(k\) is set to be 2 and the threshold is set to be \(4\times p\), where \(p\) is the number of available CPU cores.

As shown in Table II, when the target number for partitioning is small, OpenMP based Quadtree partitioning gains nothing than the sequential Quadtree partitioning - when the available CPU cores are limited to 1, OpenMP based Quadtree partitioning works as a sequential Quadtree partitioning. When the target number for partitioning increased to a large number like 32 thousand, OpenMP based Quadtree partitioning used

\(^4\)http://spatialhadoop.cs.umn.edu/datasets.html
\(^5\)bridges.psc.edu
the number of intersections for MBRs being found. As can be seen, ParADP has good weak scaling.

R-tree code is not thread-safe in GEOS which impacts the scalability of concurrent R-tree queries by OpenMP tasks in Line 9 of Algorithm 2. R-tree data structure is used internally to find the Quadtree cell(s) overlapping with a candidate’s MBR. Though the performance is limited, OpenMP based Quadtree partitioning is still useful for multi-core CPUs.

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 number, the gap between maximum and minimum MPI process execution times narrows for GEOS Intersects method. This demonstrates load-balancing improves for a higher number of partitions for ADP.

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 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. Partition based spatial join is affected by data partitioning [8] 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.

In Table IV, $T_{total}$ stands for the total time and Candidates are the number of intersections for MBRs being found. As can be seen, ParADP has good weak scaling.
TABLE IV: ParADP execution time for weak scaling

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

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).

E. Strong scaling for ParADP

For strong scaling experiments, roads and parks are used. Six experiments are performed with 16, 32, 48, 64, 80, 90 nodes on Bridges. Each node on Bridges has 28 cores.

TABLE V: ParADP execution time for strong scaling

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

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. Except for the instance with 96 nodes, ParADP has high efficiency which ranges from 0.84 to 1.02, where the highest efficiency 1.02 is achieved for the instance 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.

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 \(\text{Intersects}\) method and 2) GEOS \(\text{Intersection}\). \(\text{Intersection}\) method takes more time than \(\text{Intersects}\) method because the output geometry needs to be computed for \(\text{Intersection}\). 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 \(\text{Intersection}\) 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 \(\text{Intersection}\) 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 applied \(\text{Intersects}\) on the partitioned parks and sports. ParADP shows improvement over Quadtree partitioning in both experiments.

The \(\text{Intersects}\) 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 \(\text{Intersects}\) execution times for processing ParADP partitioned cells are within a narrow range and none of them exceed 0.8 second. On the other hand, the \(\text{Intersects}\) execution times for processing Quadtree-based partitioned cells have higher variation and the longest execution time takes 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.
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.

(a) Applying GEOS Intersection on two datasets generated by ADP, Quadtree partitioning, and uniform partitioning.

(b) Applying GEOS Intersects method on two datasets generated by ADP, ParADP, and Quadtree partitioning.

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 11. The parks and sports are partitioned to 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.

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.

REFERENCES


