A Heterogeneous HPC Platform for Ill-Structured Spatial Join Processing

Danial Aghajarian
Advisor: Sushil Prasad
Computer Science Department, Georgia State University
daghajarian@cs.gsu.edu, sprasad@gsu.edu

ABSTRACT

Given two layers of large polygonal datasets, detecting those pairs of cross-layer polygons which satisfy a join predicate, such as intersection or contain, is one of the most computationally intensive primitive operations in the spatial domain applications. There are alternative solutions for this ill-structured problem in the literature. However, non of them is designed to take advantage of heterogeneous clusters equipped with GPU accelerators to process big spatial data efficiently. In this research, we propose a distributed heterogeneous HPC platform for spatial join processing based on two-step filter and refinement approach. This work includes two main parts. First, we introduce a set of novel GPU-suited data structures and algorithms. Proof of correctness and analysis is also provided for each algorithm. Second, by applying these new techniques, we propose several GPU-based spatial join systems for various operations such as ST_intersect and polygon overlay to improve the performance of current state of the art systems.

1 Introduction and Motivation

Spatial data comprising rectangles, polygons, lines, and points are wide-spread in Geographic Information Systems (GIS). Because of advanced remote sensing technologies, the volume of data generated in such applications has tremendously increased over the past decade. This shows ever-increasing demand for High Performance Computing (HPC) such as cloud computing, Graphics Processing Units (GPU) and Message Passing Interface (MPI) programming in GIS domains. There are several distributed systems to make HPC computing available for geospatial processing including cloud-based systems, Message Passing Interface (MPI) systems, and map-reduce systems to handle tremendous volume of spatial data. Even with such parallelism, employing only CPUs in modern heterogenous architectures, typically equipped also with GPU, one to two orders of speedup remains unharnessed. One effective way of reducing the number of nodes while keeping up with the required computing power is to accelerate the computations in Graphic Processing Units (GPU).

Spatial join is one of the most computationally intensive operations in spatial computing. Therefore, harnessing parallel processing capabilities of modern hardware platforms to perform join operation over big spatial datasets is essential.

Generally, spatial join algorithms over polygonal data follow a two-phase paradigm:

- Filtering phase: reduces all the possible cross-layer polygon pairs to a set of potentially intersecting candidate pairs based on minimum bounding rectangle overlap-test.
- Refinement phase: removes any results produced during the filtering phase that do not satisfy the join condition.

The filtering phase can be presented as an ill-structured problem since there are many alternative solution for this phase including but not limited to R-tree, Quad-tree, regular grid, plane sweeping and no technique has superior performance and depend on computing platform and properties of spatial datasets one method may outperform the others

The refinement phase is significantly time-consuming. For instance, an analysis of join operation on CPU over more than 10,000 spatial objects shows that refinement phase takes five times more than the rest of the operations including filtering and parsing datasets. While this study demonstrates the significance of refinement step, in the current literature, most GPU-related works have only addressed the filtering phase algorithms.

The goal of this reseach is to study current challenges of data structure and algorithms used in filtering and refinement phases of spatial join systems and address them based on state of the art hardware and software GPU-based solutions to make it possible to process spatial big data in a real time manner. The rest of this report is organized as follows: In next section, we provide a brief problem formulation. then, we explain three new algorithms we designed for filtering technique. Furthermore, spatial join processing systems built upon these algorithms are presented and finally we state our conclusion and future work.

2 Problem Definition

Given a polygon P, MBR_P is the minimum bounding rectangle of P. Also for two overlapping bounding rectangles, MBR_{P1} and MBR_{P2}, we define *Common MBR*, $MBR_{P1 \cap P2}$, as the minimum bounding rectangle of their overlapping area. Finally, for any polygon P, E_P is the list of edges and $E_P(i)$ denotes i-th edge.

Spatial join operation can be defined over two spatial objects and a predicate. In this paper, we define spatial join as follows: for any given pair of polygons, $P1$ and $P2$, $P1 \bowtie P2$ returns true, if and only if either there exists a pair of edges $E_{P1}(i)$ and $E_{P2}(j)$ such that they intersect, or if overlap or one of the polygons lies inside the other one.

3 Algorithms

We designed thee novel parallel algorithms for filtering phase that are briefly explained in this section.

3.1 Sort-based MBR Filtering (SMF)

SMF takes *MBR* sets **R** and **S** with $|\mathbf{R}| = m$ and $|\mathbf{S}| = n$ as input and generates cross-layer MBR-overlapping pairs as output set **C**. *SMF* describes the Sort-based MBR Filter suitable for GPUs. In order to achieve more speedup, we designed *CRadixSort*, a customized parallel radix sort algorithm to exploit GPU architecture with following features.

- *sortIndex*: In order to prevent swapping 64-bytes elements in **X** over GPU main memory which is not efficient, CRadixSort prepares sorted indices such that *sortIndex[i]* is the index of i-th smallest element in **X**.
- *rankIndex*: To have an efficient parallel algorithm, each MBR_i needs to know indices of its left and right coordinates in vector **X** in $O(1)$ (without searching through *sortIndex*). To provide this information, we introduce *rankIndex* which keeps track of MBRs in *sortIndex* vector. *rankIndex[i]* is the index of x_i at *sortIndex*.

The following properties are always held by these two vectors for any $0 \leq i \leq m + n - 1$:

$$rankIndx[\ sortIndx[\ i\ ]\ ] = sortIndx[\ rankIndx[\ i\ ]\ ] = i \tag{1}$$

We showed that given two sets of MBRs, **R** and **S**, SMF algorithm will generate all overlapping MBR pairs without any false positives or duplicates. We also proved that the algorithm's complexity is:

$$= O((n+m) \cdot b + \frac{\bar{w}}{W_a} \cdot (n+m)^2) \tag{2}$$

that it depends on $\frac{\bar{w}}{W_a}$ factor which is proportional to the number of output pairs and can change between $\frac{\bar{w}}{W_a} = O(\frac{1}{n+m})$ and $\frac{\bar{w}}{W_a} = O(1)$. We showed that SMF is a linear time algorithm for most of real datasets, however, the worse case could be $O((n+m)^2)$ in case $\bar{w} \approx W_a$ which means each MBR is almost as wide as the entire area and therefore has potential overlap with almost all the other MBRs.

We have used a sequential optimized *GEOS* library as baseline to compare with *SMF*. Table 1 shows our experimental results using all three pairs of real datasets that shows up to 57-fold speedup.

Table 1: Running time of *SMF* and GEOS for MBR filtering

Dataset	Running time (ms)		♯ of Outputs
	GEOS	SMF	
Urban	197	6	28,687
Telecom	2,683	82	747,086
Water	13,048	227	1,020,458

3.2 Common MBR (CMF) Filter

CMF is an additional level of filtering that is applied on polygon edges to reduce number of candidate polygon pairs as well as the number of edges to be considered in the refinement phase by eliminating those edges that do not intersect Common MBR. Given a pair $(P1, P2) \in \mathbf{C}$, with corresponding MBRs, MBR_{P1} and MBR_{P2}, Common MBR ($MBR_{P1 \cap P2}$) is defined as the area covered by both of them (See green rectangles in Figure 1).

Here, we provide two lemmas which are the basis of **CMF** algorithms.

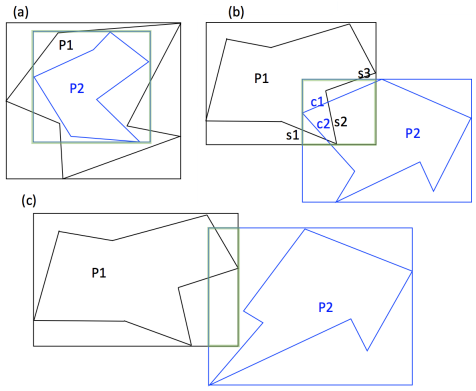

Figure 1: Examples for three *CMF* output classes: (a) $MBR_{P1 \cap P2} = MBR_{P2}$ and CMF will tag $(P1, P2)$ for "P1 contains P2" *point-in-polygon* test. (b) $MBR_{P1 \cap P2} \neq MBR_{P1}$ and $MBR_{P1 \cap P2} \neq MBR_{P2}$ and $P1 \cap MBR_{P1 \cap P2} \neq \emptyset$ and $P2 \cap MBR_{P1 \cap P2} \neq \emptyset$, therefore (P_1, P_2) is directly sent to *edge-intersection* test. (c) $P2 \cap MBR_{P1 \cap P2} = \emptyset$ which means the pair is disjoint.

LEMMA 1. ***CMF-Pre-PnP Test**: Given polygon pair $(P1, P2) \in \mathbf{C}$ with corresponding minimum bounding rectangles MBR_{P1} and MBR_{P2}, if P1 contains P2, then MBR_{P1} contains MBR_{P2}. In other words, we have $MBR_{P1 \cap P2} = MBR_{P2}$.*

Lemma 1 provides a necessary condition for *point-in-polygon* test. We can classify a given polygon pair of **C**, $(P1, P2)$, into one of the three following categories:1) Pairs with $MBR_{P1 \cap P2} = MBR_{P1}$, 2) Pairs with $MBR_{P1 \cap P2} = MBR_{P2}$, and 3) Pairs with partially-overlapping MBRs (See Figure 1). The first two classes can be added to *within candidate set* **W** for actual *point-in-polygon* test. By applying Lemma 1 before doing this test over all **C** elements, we gain performance due to the following reasons:

- For any given pair, verifying whether Lemma 1 holds true is only a constant-time operation while actual *point-in-polygon* test takes $O(n_e)$ where n_e is the number of edges.
- Join predicate requires testing for both "P1 contains P2" and "P2 contains P1" cases, but Lemma 1 identifies which polygon may contain the other one that eliminates one unnecessary test.

LEMMA 2. ***CMF-Pre-Edge-intersection Test**: Given two edges $E_{P1}(i)$ and $E_{P2}(j)$ from polygons P1 and P2, if the edges intersect, then they either completely lie inside $MBR_{P1 \cap P2}$ or intersect it. In either case, intersection point is not outside $MBR_{P1 \cap P2}$.*

Lemma 2 provides a necessary condition for *edge-intersection* test. It says that given $(P1, P2)$ pair, if any edge from P1 lies completely outside of $MBR_{P1 \cap P2}$, it will not intersect with P2. As a result, we can remove that edge from polygon edge list for the refinement phase.

The following Corollary is a direct result of Lemma 1 and 2 and it can be used to detect some disjoint pairs in **C** before refinement phase.

COROLLARY 1. *Given pair $(P1, P2) \in \mathbf{C}$, let $\hat{E}_{P1} = \{i | E_{P1}(i)$ either intersects $MBR_{P1 \cap P2}$ or lies inside it$\}$.*

Similarly, we can define $\hat{E}_{P2}$, intersecting-edge candidate set for P2. P1 and P2 are disjoint if $(P1, P2) \notin \mathbf{W}$, the within candidate set, and $\hat{E}_{P1} = \emptyset$ or $\hat{E}_{P2} = \emptyset$

To illustrate Lemma 1 and 2 and Colorrary 1, three different examples are shown in Figure 1.

CMF thus classifies elements of **C** as follows:

1. *Within candidate set* (**W**): set of all the polygon pairs $(P1, P2) \in \mathbf{C}$ such that $MBR_{P1 \cap P2}$ is either equal to MBR_{P1} or MBR_{P2}.

2. *Intersecting-edge candidate set* (**I**): set of all polygon pairs $(P1, P2) \in \mathbf{C}$ such that $(P1, P2) \notin \mathbf{W}$ and $\hat{E}_{P1}$ and $\hat{E}_{P2}$ are non-empty.

3. *Disjoint set*: Polygon pairs $(P1, P2) \in \mathbf{C}$ that are neither in **W** nor in **I**.

As *CMF* iterates through each edge once, the algorithm complexity is $O(n_e)$ which n_e is number of edges. In the worst-case, all edges may lie inside their *common MBR* or cross it, but based on our real data experimental results, shown in Table 2, applying *CMF* reduces the effective edges by factor of 40. This filter also eliminates almost two-third of pairs by applying Lemma 2.

Table 2: CMF effect on reducing workload of the refinement phase for Water datasets.

	No CMF	*With CMF*
Time (ms)	120,751	4,401
♯ of Edge-intersecting pairs	566,656	198,142
♯ of edges (layer1)	1,048,479,573	25,969,322
♯ of edges (layer2)	954,431,290	20,451,866

Following lemma proves that even in the worst-case scenario, *CMF* will reduce number of operations in our point-in-polygon test.

LEMMA 3. ***PnP analysis**: Given a candidate set **C** of potentially intersecting polygons, applying CMF filter will always reduce the overall work for point-in-polygon test.*

3.3 CMF-grid

As a followup work to CMF filter, we delved deeper into the *Common MBR* concept. The overall idea is to partition each *Common MBR* into grid-cells (uniform for each pair but variable across different pairs, suitably determined). *CMF-Grid* algorithm then discards those pairs that none of whose grid-cells contain edges from both polygons. Figure 2 illustrates the differences between traditional grid techniques and our adaptive approach. *CMF-Grid* can be distinguished from all the other grid-based techniques in the geospatial processing literature by following two features:

1. In *CMF-Grid*, grid-cells do not necessarily cover the whole universe (Figure 2-b).

2. In *CMF-Grid*, grid-cells may overlap with each other (in cases that *Common MBRs* of overlapping pairs overlap).

We proved that given a polygon pair (P_{i_1}, P_{i_2}) in C and $CMBR_i$, the average number of cross-polygonal in-cell all-to-all edge-intersection tests (workload) after applying *CMF-Grid* can be formulated as a quadratic function of w_{g_i} as:

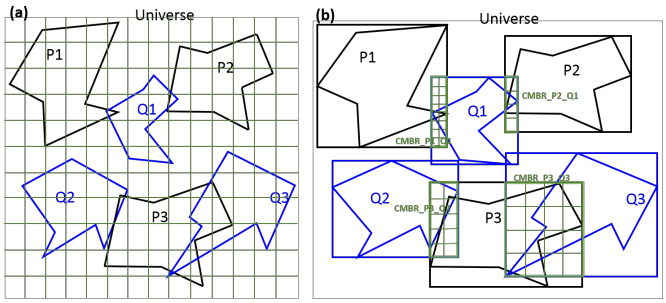

Figure 2: (a) uniform grid technique, and (b) *CMF-Grid*: In *CMF-Grid*, grid-cells are not of the same size and may not cover the whole universe.

$$W_i^E = Function(w_{g_i}) = \frac{|\hat{E}_{i_1}|.|\hat{E}_{i_2}|}{\pi^2 k_i h_i w_i} \times \Big[\pi^2 k_i^2 w_{g_i}^2 + 2\pi k_i (k_i + 1)(\hat{\hat{E}}_{i_1} + \hat{\hat{E}}_{i_2}) w_{g_i} + 4(k_i + 1)^2 \hat{\hat{E}}_{i_1} \hat{\hat{E}}_{i_2}\Big] \quad (3)$$

where $0 < w_{g_i} \leq w_i$

The key achieved results in CMF-grid technique are:

1. ***CMF-Grid***: A GPU-based non-uniform grid technique over *Common MBR* reduces the refinement phase workload more than $30,000$ times compared to the naive all-to-all algorithm. It improves over its predecessor, *CMF* filter, by reducing workload by 700 times.

2. We showed that in order to achieve the fastest end-to-end running time, *CMF-Grid* does not need to minimize the workload by applying a very fine grid; 4 to 6 times coarser grid-cell sizes can lead to best performance.

3. We also showed the impact of the shape of grid-cells on workload of refinement phase for square and CMBR-proportioned shapes.

Table 3 compares running time of refinement phase for all-to-all, CMF and CMF-grid filtering techniques.

Table 3: Refinement workload reduction of Water dataset for all-to-all, *CMF* and *CMF-Grid*.

	All-to-all	*CMF*	***CMF-grid***
Time (ms)	120,751	981	**89**
Pairs $\in I$	566,656	198,142	198,125
Edges(all)	2,002,910,863	46,421,188	**147,002,513**
Edges(active)	2,002,910,863	46,421,188	**15,043,183**
Workload	$535,108,085,968$	12,794,606,592	**17,370,352**

4 GPU-based spatial join processing systems

4.1 GCMF

While using GPU accelerators can achieve 1 to 2 orders of magnitude speedup in processing spatial data, to the best of our knowledge there is no GPU-based spatial join system based on ST_intersect in the literature. Therefore, we decide to design and develop such a system using the algorithms and data structures proposed in the previous part.

4.1.1 System Design

ST_Intersect predicate requires both *edge-intersection* and *point-in-polygon* tests. R-trees are used to index polygons and then R-tree query is used to detect potentially overlapping polygons. Finally, *point-in-polygon* and *edge-intersection* tests are applied in the refinement phase. Overall running time of the traditional system is heavily dominated by the refinement phase which we try to address by introducing our new system design.

GCMF has two subsystems. The first subsystem includes two filtering components, *SMF* and *CMF*, explained in previous sections. The refinement subsystem comprises two components: *point-in-polygon* test (*PnP_Test*) and *edge-intersection* test (*EI_Test*). The first component takes **W** as input and performs the *point-in-polygon* test. If a pair passes the test successfully, it goes to output directly, otherwise it is sent to *edge-intersection* test for further processing. As shown in the Figure 3, the input of the *EI_Test* comes from **I** as well as those pairs from **W** which failed *point-in-polygon* test. Finally, *EI_Test* adds a pair to output, if it can detect at least one cross-layer edge-intersection/overlap in that pair.

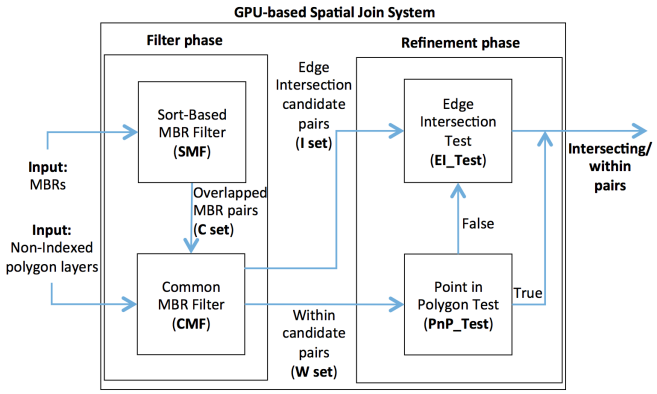

Figure 3: GCMF system design overview

The filter phase components (SMF and CMF) are already explained in the algorithm sections. The following are the properties of the refinement phase of GCMF system:

- Parallel Point-in-Polygon Test (PnP): Ray crossing test is implemented in parallel.
- Parallel Edge-Intersection Test (EI): a parallel optimized edge-intersection test over GPU.

We compared PnPTest performance versus two other algorithms 1) sequential version of crossing test and 2) naive *point-in-polygon* test over GPU in which each thread is responsible for a given test point. *PnPTest* gains 28 to 30-fold and 8 to 9 fold speedup compared to sequential and naive GPU algorithms respectively.

4.2 GCMF+

GCMF introduced *CMF* filter to reduce *edge-intersection* refinement phase workload by filtering out those edges that lie outside of their *Common MBR* for pairs of polygons.

Building on our previous work in *GCMF*, the improved system has two upgraded components (Figure 3-green boxes) compared to *GCMF* as follows:

- We replace *CMF* filter with *CMF-Grid* technique.

Table 4: End-to-end running time of spatial join with ST_intersect operation for four different systems.

Dataset	Running Time (ms)			
	Sequential		Parallel	
	PostGIS	GEOS	GCMF	**GCMF+**
Urban	3,120	5,770	52	31
Water	232,122	148,040	1,663	739

- A new grid-based *edge-intersection* test based on in-cell all-to-all *Edge-intersection* test.

Table 4 shows running time of GCMF and GCMF+ with PostGIS and optimized GEOS base systems. GCMF+ by taking advantage of CMF-grid shows 225% improvement over GCMF.

- GCMF+ with CMF-grid filter shows up to 200-fold speedup versus optimized GEOS sequential library.
- GCMF+ also shows a 225% improvement over GCMF by replacing CMF with CMF-grid filter.

4.3 MPI-CUDA-GIS: A Distributed System over GPU-equipped Clusters (In progress)

MPI-GIS is a distributed system based on Message Passing Interface (MPI) for overlay operation of two large layers of polygons. The system has four components as follows:

- Input component: It reads data and organize them into R-tree data structure.
- Task Creation: It performs a query into R-tree to identify potentially overlapping cross-layer pairs.
- Process component: Each task (pair) is sent to a MPI process for additional processing using edge-intersection or containment test.
- Output component: It collects all the output polygons from different processes and generates output file.

Building on MPI-GIS, we plan to incorporate our new filter and refinement algorithms into this system to make our system scalable. By integrated system will benefit from GPU-acceleration and distributed computing that leads to a system that:

- The achieved speedup is expected to be 10-50 times better than current MPI-GIS system.
- The system is expected as scalable as MPI-GIS.

Designing a distributed system that can handle Terabyte-size spatial data has some challenges that we want to address in the last part of our work. These challenges are summarized as follow:

- **Load balancing and partitioning**: Partitioning data based on resources available in each node to make sure the computation is distributed as evenly as possible such that minimizes data duplication.
- **Scalability**: We need to make sure that distributed system holds weak scaling condition that it by increasing data size and adding resources to the system, the total running time remains the same.
- **IO**: Reading Terabyte-size data from disk can take much more time than the time that is required for processing such a data. This makes IO a bottleneck for our system. Therefore, applying efficient parallel IO methods such as MPI-IO library is necessary.