8-7-2018

A Heterogeneous High Performance Computing Framework For Ill-Structured Spatial Join Processing

Danial Aghajarian
Georgia State University

A HETEROGENEOUS HIGH PERFORMANCE COMPUTING FRAMEWORK FOR ILL-STRUCTURED SPATIAL JOIN PROCESSING

by

DANIAL AGHAJARIAN

Under the Direction of Sushil K. Prasad, Ph.D.

ABSTRACT

The frequently employed spatial join processing over two large layers of polygonal datasets to detect cross-layer polygon pairs (CPP) satisfying a join-predicate faces challenges common to ill-structured sparse problems, namely, that of identifying the few intersecting cross-layer edges out of the quadratic universe. The algorithmic engineering challenge is compounded by GPGPU SIMT architecture. Spatial join involves lightweight *filter* phase typically using overlap test over minimum bounding rectangles (MBRs) to discard majority of CPPs, followed by *refinement* phase to rigorously test the join predicate over the edges of the surviving CPPs. In this dissertation, we develop new techniques - algorithms, data structure, i/o, load balancing and system implementation - to accelerate the two-phase spatial-join processing. We present a new filtering technique, called Common MBR Filter (*CMF*), which changes the overall characteristic of the spatial join algorithms wherein the *refinement* phase is no longer the computational bottleneck. *CMF* is designed based on the insight that intersecting cross-layer edges must lie within the rectangular intersection of the MBRs of CPPs, their common MBRs (*CMBR*). We also address a key limitation of *CMF* for class of spatial datasets with either large or dense *active* *CMBR*s by extended *CMF*, called *CMF-grid*, that effectively employs both *CMBR* and grid techniques by embedding a uniform grid over *CMBR* of each CPP, but of suitably engineered sizes for different CPPs. To show efficiency of CMF-based filters, extensive mathematical and experimental analysis is provided. Then, two GPU-based spatial join systems are proposed based on two *CMF* versions including four components: 1) sort-based MBR filter, 2) *CMF/CMF-grid*, 3) point-in-polygon test, and, 4) edge-intersection test. The systems show two orders of magnitude speedup over the optimized sequential GEOS C++ library. Furthermore, we present a distributed system of heterogeneous compute nodes to exploit GPU-CPU computing in order to scale up the computation. A load balancing model based on Integer Linear Programming (ILP) is formulated for this system. We also provide three heuristic algorithms to approximate the ILP. Finally, we develop *MPI-cuda-GIS* system based on this heterogeneous computing model by integrating our CUDA-based GPU system into a newly designed distributed platform designed based on Message Passing Interface (MPI). Experimental results show good scalability and performance of MPI-cuda-GIS system.

INDEX WORDS: Spatial data, Spatial join, GPU computing, Parallel algorithm, Colocation pattern mining, Distributed systems, MPI, Heterogeneous systems, Load balancing, ILP optimization

A HETEROGENEOUS HIGH PERFORMANCE COMPUTING FRAMEWORK FOR ILL-STRUCTURED SPATIAL JOIN PROCESSING

by

DANIAL AGHAJARIAN

A Dissertation Submitted in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

in the College of Arts and Sciences

Georgia State University

2018

A HETEROGENEOUS HIGH PERFORMANCE COMPUTING FRAMEWORK FOR ILL-STRUCTURED SPATIAL JOIN PROCESSING

by

DANIAL AGHAJARIAN

DEDICATION

I would like to dedicate this work to my parents who have always supported and loved me unconditionally.

ACKNOWLEDGEMENTS

This work would not have been possible without the support of many people. I want to express my gratitude to my advisor Dr. Sushil K. Prasad who has completely supported me during my PhD. I am also thankful to other committee members Dr. Rajshekhar Sunderraman, Dr. Shamkant B. Navathe and Dr. Rafal Angryk for their great comments and guidance. Finally, I appreciate Dr. Satish Puri for his directions and helps especially in the first two years of my research.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

- CMF: Common MBR Filter
- EI_Test: Edge Intersection Test
- GPU: Graphic Processing Unit
- SIMT: Single Instruction Multiple Thread
- HPC: High Performance Computing
- MBR: Minimum Bounding Rectangle
- MPI: Message Passing Interface
- PnP_Test: Point-in-Polygon Test
- SMF: Sort-based MBR Filter
- ILP: Integer Linear Programming

Chapter 1

INTRODUCTION

1.1 Introduction

Given two large layers of spatial data representing spatial location of features and boundaries, medical images, traffic, electronic circuit and etc, finding cross layer objects satisfying a predicate such as intersect, overlay, point in polygon, etc., is called spatial join. This operation is significantly important in many applications including but not limited to Geographic Information System (GIS), Weather forecasting systems, Biology, VLSI and Social sciences [1, 2]. The ever increasing volume of these spatial data on the one hand and the crucial need of real-time processing of them in order to extract helpful information on the other hand makes it necessary to exploit High Performance Computing (HPC) such domains. For instance, NASA satellite data archives exceeded 500TB and it is still growing rapidly [3]. Also, the primitive-like overlay operation over two layers of spatial objects, including more than *700,000* polygons using the state of art ArcGIS software can take more than 13 minutes on a single compute node [4]. Another example is spatial join of a polyline table with 73M records representing the contiguous USA with itself that takes roughly 20 hours to complete on an Amazon EC2 instance [2]

To address these challenges, researchers have designed several distributed architectures to make HPC computing available for geospatial processing including cloud-based systems [5, 6], Message Passing Interface (MPI) systems [7], and map-reduce systems [8]. Most of these systems make use of powerful and expensive computing clusters to break down the computations over several distributed nodes. Some of these applications handle tremendous volume of spatial data which requires using many nodes. While those works mainly have focused on the system design aspects, there are still some unexplored theoretical and algorithmic areas in the field of spatial join processing that make it possible to efficiently reduce search space and as a result computing time decreases without using any extra computing resources.

Furthermore, even with such a parallelism, employing only CPUs in modern heterogeneous architectures, typically equipped also with Graphic Processing Units (GPU), one to two orders of speedup remains unharnessed [9]. One effective way of reducing the number of nodes while keeping up with the required computing power is to accelerate the computations using GPU. Therefore, effective employment of CPU-GPU pair is critical for real-time spatial processing [1] [2]. GPUs are very popular among the HPC technologies as they are widely available at low prices yet with powerful features. The state of art GPUs are very powerful and can reduce CPU-GPU communications by directly transferring some data between each other. For example, *Tesla P100* GPU from *NVIDIA* with Pascal$^{\text{TM}}$architecture, has 16 *GB* of the main memory and it provides $3,584$ *Cuda cores* operating at 1480 MHz base clock that provides 5.3 *TFLOPS* of double precision floating point calculations. Pascal$^{\text{TM}}$architecture also introduces *NVLink*, the new high speed interconnect technology for GPU-to-GPU and GPU-to-system communications that is up to 5 times faster than traditional *PCIe* bus. Each *P100* GPU is equipped with four *NVLink* connections that can be used in various topologies. One popular architecture is a mesh network of 8 GPUs connected by these high-speed links. In this topology GPU-CPU communications use the *PCIe* bus. Figure 1.1 illustrates a traditional computing node versus GPU-enabled computing node. Theoretically, these GPU nodes can process data orders of magnitude faster than traditional nodes depend on the problem. Upgrading current nodes is a more feasible and less costly solution than adding more nodes, moreover most supercomputing clusters are already equipped with GPUs and, because of this, algorithms and data structures based on GPU-architecture are a good fit for geospatial big data processing.

In general, spatial join is defined as follows: given two spatial datasets R and S and a spatial join predicate $\bowtie$ (e.g., overlap, contain, intersect) as input, spatial join returns the set of all pairs (r, s) where $r \in R,\ s \in S$, and $\bowtie$ is true for (r, s) [10]. A typical application of a spatial join is “Find all pairs of rivers and cities that intersect.” Another polygonal spatial

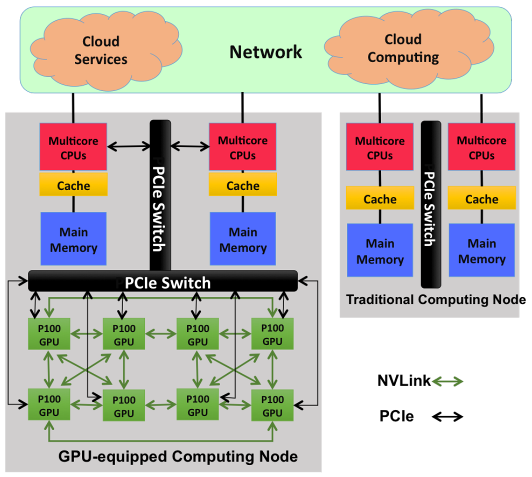


Figure (1.1) CPU-only computing node vs CPU-GPU computing node

join operation is *ST_Intersect* in which for a given pair of polygons, it returns true if and only if polygons share any portion of space [11].

Generally, spatial join algorithms over polygonal data follow a two-phase paradigm [10]:

- Filtering phase: reduces all the possible cross-layer polygon pairs to a set of potentially intersecting candidate pairs based on Minimum Bounding Rectangle (MBR) 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. A problem is ill-structured if it has well-defined beginning state but usually undefined end state, actions, goals and constraints [12] [13]. We can represent ill-structured filtering problem as follows:

- **Beginning State**: We always start with two sets of spatial objects (P and Q where $|P| = m$ and $|Q| = n$)(**well-defined**)

- **Goals**: The general goal of filtering is to reduce search space as much as possible for the refinement phase but we cannot further elaborate any objective goals for this operations. (**almost undefined**)

- **Actions**: As we will see in literature review section, there are several approaches that have been applied for filtering spatial data including but not limited to R-tree, Quad-tree, regular grid, plane sweeping and etc. It is also clear that no technique has superior performance and depend on computing platform and properties of spatial datasets one method may outperform the others. (**undefined**)

- **End State**: Depend on efficiency of applied actions and spatial distribution of objects End State can vary from empty set to all possible spatial pairs ($\{(p_i, q_j) \mid 1 \leq i \leq m\ ,\ 1 \leq j \leq n\}$). (**undefined**)

The refinement phase is significantly time-consuming. For instance, an analysis of join operation on CPU over more than 10,000 spatial objects in [14] 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.

1.2 Motivation

Based on what is stated so far, there are considerable issues in the current literature of spatial join processing. The following summarizes these challenges:

1. **Lack of efficient parallel data structures and algorithms**: Tree-based data structures such as R-trees and interval trees have been used in spatial data processing. While these data structures are designed for sequential algorithms, they are not currently suitable for large datasets over GPUs mostly because of their hierarchical structures and memory usage in their implementations. Furthermore, building up tree structures in cases which spatial objects are skewed such as tall or wide MBRs is inefficient [15].

2. **Unexplored efficient filtering techniques**: MBR overlap test is the standard method for filtering out disjoint pairs of objects. However, due to its insufficient filtering capabilities, output of filtering phase is still too large. There are some un-

explored aspects of filtering algorithms that can make room for further elimination of potentially interesting pairs before computationally-extensive refinement phase.

3. **Inadequate GPU-based spatial join primitive libraries**: Although CudaGIS [16] has introduced some spatial operations over GPUs, this library is not comprehensive and some primitives such as edge-intersection are not covered. Also, some of the operations such as point-in-polygon test are not designed to fully leverage GPU architecture capabilities.

4. **Lack of efficient GPU-based spatial join processing systems**: People have used GPUs for partially processing spatial data but to the best of our knowledge there are not GPU-based systems for polygonal overlay and ST_intersect.

5. **Lack of heterogenous distributed systems for processing spatial data**: Currently, all the distributed systems are designed to work on powerful clusters to take advantage of couple hundreds CPU cores. While almost all the recently built clusters are also equipped with GPUs, developing a system for heterogenous computing model to use CPU-GPU power at the same time can significantly increase the speedup and lead to more scalability.

The goal of this dissertation is to study current challenges of data structure and algorithms used in filtering and refinement phases of spatial join systems and address some of them based on the state of art heterogeneous models (CPU-GPU-based) to make it possible to process spatial big data in a near-real-time manner. The rest of this manuscript is organized as follows: In Chapter 2, first, we provide the basic definitions and concepts. Then, various aspects of spatial join problem in the current literature are explored in details. Chapter 3 formulates *CMF*, a novel space reduction filter based on Common MBR (*CMBR*) of two MBR-overlapping spatial objects, and proposes a GPU-based system for spatial join processing of large polygonal datasets. In Chapter 4, we extend the concept of *CMF* filter to *CMF-Grid* by applying a uniform grid technique over the *CMBR* area. *CMF-Grid* effectively achieves more performance over *CMF* in cases where spatial objects are dense in their

CMBRs. Chapter 5 provides an example of spatial join in colocation mining over GPU. Our proposed GPU optimization shows a significant improvement over recently grid-based technique. Then, in Chapter 6, optimization of load balancing and processing of spatial data in distributed heterogeneous computing environments is formulated as an Integer Linear Programming (ILP) problem. Then, we propose *MPI-cuda-GIS* system, a distributed system that integrates CPU and GPU computing into a HPC framework based on MPI and cuda architectures. Finally, we provide conclusion of future works of our research in Chapter 7.

Chapter 2

LITERATURE REVIEW

2.1 Introduction

Spatial vector data is a general term that can be applied to any data that presents geographical objects and locations in any form, referred as spatial data types [17], such as lines, rectangles, points, or in general polygons. These objects can be one, two or three dimensional and are used in the same way as attribute data types such as integers, floats and strings [18]. In other words, every part of a map (cities, rivers, highways, intersections, forests, electronic objects, etc.) can be presented as a spatial object in the form of geometric shapes. There are wide ranges of applications from Computer Aided Design (CAD) to Geographical Information Systems that use these spatial data to process their data [19, 20, 21]. In general, these processes can be any operation on any number of operands representing the relation between one (etc. self-spatial join), two or more (multiway join) sets of spatial objects [22]. In the literature, general problem sometimes is reffered as overlay problem [23] where operations, as well as the number and the type of operands can be of any arbitrary form; However in practice, binary intersection is more popular. In this chapter, we study different algorithms used for spatial join. In particular, hierarchical data structures such as various kind of trees and their applications in spatial join processing and indexing will be explored in details. We classify these techniques and point out their issues such as memory limitations and managements. The main goals of this chapter can be summarized as follows:

- We state the basic definitions that are used for comparison and evaluation of different spatial data structures.
- We provide an introduction of indexing data structures and digging into their basic concepts. We also state pros and cons of each structure in different applications and

datasets.

- We review the spatial join techniques in the literature in the following subcategories:
  - Hierarchical tree-based and plane sweep algorithms for spatial join processing.
  - Grid-based techniques for spatial join processing.
  - Distributed systems for spatial join processing.

The rest of this chapter is organized as follow. In the next section, we describe general spatial join problem and the basic concepts. Then, we state the classification of data structures and algorithms. The hierarchical indexing data structures based on classification is presented in Section 2.3. Grid-based techniques are explained in Section 2.4. Finally, we point out spatial join systems in Section 2.5.

2.2 Basic Definitions

We presents some basic definitions from the literature that may be used for comparing the space or performance efficiency of different indexing methods. For some definitions, there is more than one definition and we try to cover all of them.

2.2.1 Overlap

Overlap is the amount of volume is covered with more than one spatial object. Overlap has a direct impact on query efficiency. The more the overlap, the less efficient queries we will have. More overlap means more paths to the search for finding a given query. Efficient data structures tend to minimize overlap parameter. For example in R-tree, if an interested area is covered with k intermediate nodes, then k paths must be searched in the worse-case scenario that increases the processing time by factor of $O(k)$. In [24] two definitions is presented for overlap as follow:

Simple overlap: The portion of the space covered with more than one hyper-rectangle in a node. Assume $\{R_1, R_2, ..., R_n\}$ is the set of hyper-rectangles in a given node. Then the overlap is defined as:

$$simple\ overlap = \frac{\| \bigcup_{i \neq j} R_i \cap R_j \|}{\| \bigcup_{i=1}^{n} R_i \|} \tag{2.1}$$

Where $\|A\|$ is the volume covered by hyper-rectangle A. This is a simple definition of overlap that does not take the number of overlapping objects into account. To address this issue, the weighted overlap is defined as follows:

Weighted overlap: Weighted overlap is the portion of objects lying in the overlapping area defined as:

$$simple\ overlap = \frac{|\{p \mid p \in \bigcup_{i \neq j} R_i \cap R_j\}|}{|\{p \mid p \in \bigcup_{i=1}^{n} R_i\}|} \tag{2.2}$$

where $|A|$ denotes number of elements in A. If the distribution of data is not uniform, weighted overlap is more efficient than simple overlap.

2.2.2 Coverage

The overall area covered of all the leaf nodes in a tree is defined as coverage [25] formulated as follows:

$$coverage = \sum_{i=1}^{n} \sum_{j=1}^{|LF_i|} A(LF_i(j)) \tag{2.3}$$

where n is the number of leaf nodes in the data structure, LF_i is the set of all objects in the i-th leaf node and $A(X)$ is the area covered by object X. Coverage can also be defined over other levels of a tree in the same way as leaf level. Efficient data structures tend to minimize this parameter because the smaller coverage the less dead space (the area covered in the tree while it contains no object) will be in the data structure.

2.2.3 Range queries

There are different types of range queries for spatial data; however, generally speaking, range queries can be expressed by determining a region (region query). The formal definition of range query is presented as follow: given a k dimensional space of the form $D_0, D_1, ..., D_{k-1}$, we define an interval over this domain as $I_0 \times I_1 \times ... \times I_{k-1}$ such that $I_i \in D_i$. Then, various kinds of range queries are defined as below [26]:

Partial range queries: if some of the intervals in a range query are full domains, the query is a partial range query, more precisely:

$\{I_0 \times I_1 \times ... \times I_{k-1} \mid \exists i \; such \; that \; I_i = D_i\}$

Partial match query: If some of the intervals are points and the rest are full domains. In the other words:

$\{I_0 \times I_1 \times ... \times I_{k-1} \mid \exists i \; such \; that \; I_i = a \; point \;\; \wedge (\forall i \; I_i \in \; \{point, D_i\})\}$

Exact match query: If all the intervals are points that is:

$\{I_0 \times I_1 \times ... \times I_{k-1} \mid \forall i \; I_i \in \;\; is \; a \; point)\}$

2.2.4 Window-based queries

The following queries are defined based on a window area [27]:

Exist queries: The query to determine whether or not a particular feature exists inside a window.

Report queries: It finds identity of all the features inside a window. This is the same as "exist" query with an additional step of accumulating features and report them as a set of features that overlap the window.

Select queries: It finds all the blocks that feature f entirely covers inside the window. In another classifications [28], stated below, "Exist" and "report" queries are considered the same:

Location-based queries: In this query, we are interested in features associated with a particular location. For example, "What are Interstate highways passing through Atlanta area?".

Feature-based queries: In this case, we are looking for all areas having a specific feature. For example, "Which cities does I85 highway pass through?" is a feature-based query.

2.2.5 q-edge

A q-edge is part of a line that passes through a block. If both endpoints of an q-edge are of the degree 1 (no connection), it is called isolated q-edge. In the Figure 2.1 the red and blue sections of lines, a and b, are examples of q-edge while line b is an isolated q-edge.

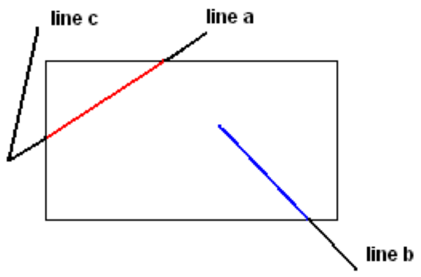


Figure (2.1) Two q-edge examples. The blue section of line b and red section of line a are q-edge. Line b is called an isolated q-edge as its two ends have no connection.

2.3 Hierarchical Indexing Spatial Data Structures

In the following subsections, we explain each hierarchical indexing data structure with examples and their differences and applications.

2.3.1 R-tree

This data structure is one of the most basic and early tree structures [29]. The key idea of R-tree is to represent nearby objects together using a MBR covering them. Spatial information is stored in leaf nodes using joint rectangles as containers. Leaves contain pointers to actual objects that usually are stored in disk pages. All leaf nodes in the R-tree contain index records of the form $(I, tuple_i d)$ which $I = \{I_0, I_1, ..., I_{n-1}\}$ and n is dimension and I_i is closed bounded interval $[a, b]$ describing the extension of the object in that dimension. On the other hand, non-leaf nodes are in the form of $(I, child_{pntr})$ where $child_{pntr}$ is the address of a child node in the subtree and I is the rectangle that covers all the children of that node [29]. Intermediate nodes are associated with some MBRs enclosing all the MBRs corresponding to its children nodes.

The degree of a R-tree is (m, M) pair where $m \leq \frac{M}{2}$ and each node except the root must have between m and M index records and root must have at least two children unless it is a leaf node. Furthermore, each R-tree like B-tree is a height-balance tree [30] and all the leaves appear in the same level. Height of R-tree with N index records is at most $\lceil \log_m N \rceil - 1$. Performance tuning factors of R-tree are parameters m and M. Simulation results have shown that linear node-split algorithm is as good as other more complicated algorithm in R-trees [29].

R-trees have some drawbacks. First, the MBRs in the leaves are overlapping which can degrade searching performance of R-trees drastically as shown in Figure 2.2. Another problem with R-trees is that they just consider area for their splitting algorithms [31]. In the section related to R*-tree has shown that this heuristic criterion does not necessarily end up to best possible result.

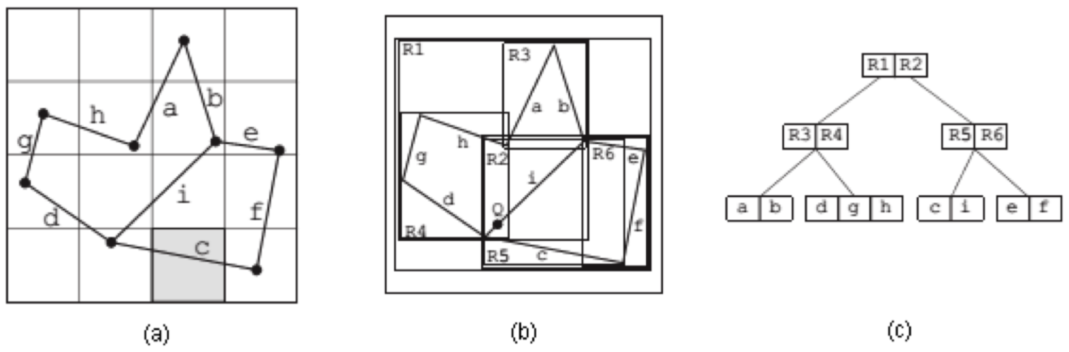


Figure (2.2) R-tree example of the degree (2,3), (a) line data. (b) MBR splits of the region with regards to the line data.(c) Corresponding R-tree.

Figure 2.2 is an example of R-tree with degree (2, 3). As shown, MBRs have overlap with each other and this degrades the efficiency of R-tree structure. In Fig 2.2, in order to find associated line to point Q, search must be started from root. Both R1 and R2 in the root contain point Q. Therefore, we have to search both of them. Searching R1 will not lead to any result. Finally, by searching R2 and then R5, we are able to find line i.

2.3.2 R^*-tree

The R-tree optimization algorithms try to minimize the area of each bounding rectangle in the inner nodes heuristically. There is no proof to show that this heuristic can always achieve the best possible insertion path. On the other words, the R-tree structure takes only area into consideration while all the other parameters have impacts on each other and it is not possible to optimize one without considering the influences of others on the overall performance. The main idea of R* tree is to address this problem by taking more parameters into account such as area, margin, and overlap and providing the following rules [31]:

1. **Minimizing the area covered by a directory rectangle**: this improves the performance by making it possible to choose traversed path in higher levels of tree.

2. **Minimizing the overlap between directory rectangles**: it decreases the number of paths to be traversed.

3. **Minimizing the margin of a directory rectangle**: it leads to more quadratic rectangles that ultimately improves the structure.

4. **Optimizing storage utilization**: the higher the storage cost, the lower the query cost as tree height will be kept low.

In the table 3, the some examples of criteria combinations and their effects have been presented: R^* tree algorithms outperforms their R-tree counterparts by considering the area, margin, and overlap of the directory rectangles and optimizing all of them at the same time. Thus, this data structure is robust against unusual data distribution. Also, R^*-tree improves its insertion cost by using Forced Reinsert operation which changes entries between adjacent nodes to decrease the overlap leading to higher data utilization, less split and margin by using more CPU cost [31].

The experimental results in [24] shows that R^*-tree does not efficiently support indexing of more than five dimensions. Figure 2.3 plots the overlap of a R^*-tree versus the dimension. As the number of dimensions increases there are more overlap in the data structure. In fact, Table (2.1) Effect of different criteria combinations on R^*-tree performance

Criteria combination	Description	Impacs
#1 and #2	Requires More freedom in number of rect-angles in a node	Lower storage utilization, Higher margin
#1	Covering of data space is reduced	Higher margin, Potentially less overlap
#3	Better packing	May either reduce or increase storage utilization
#4	Sufficiently large query rectangles, will be effected more by #4 than #1 to #3	

for dimension of 5, the overlap reaches to 90%. Finally, the main drawbacks of R* trees are listed below:

- Allocation process of entries onto the nodes is non-deterministic and highly data dependence (The same as R-tree).
- Implementation cost of R^*-tree is more than R-tree.

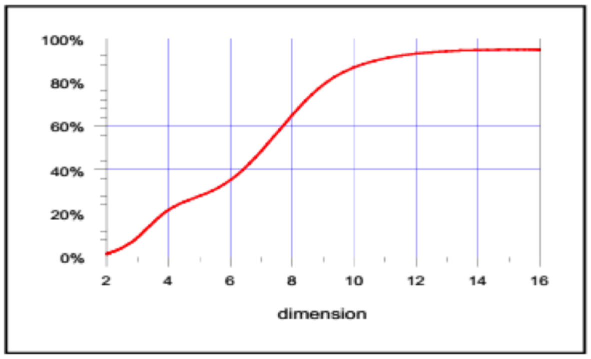


Figure (2.3) Overlap of R^*-tree vs. the dimension of space.

2.3.3 R^+-tree

R^+-tree is an extension of k-d-B-tree [32] that is built based on disjoint areas to avoid overlap. In the R-tree, each actual object belongs to one MBR in the leaf level while in the R^+-tree each object is associated to all rectangles that it intersects. On the other words, it implies that rectangles in the leaves are not necessarily minimum bounding rectangles. As a result each object may appear in more than leaf node [23].

In the R^+-tree, each leaf node is of the form $(o_{id}, Rect)$ where o_{id} is a pointer refers to a object in the database and *Rect* is used to describe the bounds of that object. On the other hand, an intermediate node is of the form $(p, Rect)$ where p points to lower level of the tree and *Rect* is a bounding rectangle which encloses all the children rectangles. Despite leaf-level rectangles, middle-level rectangles are minimum bounding. Similar to R-tree, leaves in R^+-tree are in the same level (tree is balanced). Finally, as leaf-level rectangles do not have any overlap, R^+-tree has search performance advantage over R-tree especially in point queries [32]. Another important difference between R-tree and R^+-tree is that although in R-tree number of entries in each node is between m and M/2, this property does not necessarily hold in R^+-tree. However, [26] showed that in practice storage utilization in k-d-B-trees, as the basic type of R^+-tree, remains in acceptable levels close to B-trees.

Finally, to clarify difference between R-tree and R^+-tree, the same spatial data in R-tree example is indexed by R^+-tree in the Figure 2.4. As shown, given a point, there is only one path from root to leaf and this makes search algorithm fast and efficient. As mentioned above, the problem with this data structure, is that each object may appear in more than one leaf node that may cause too much overhead.

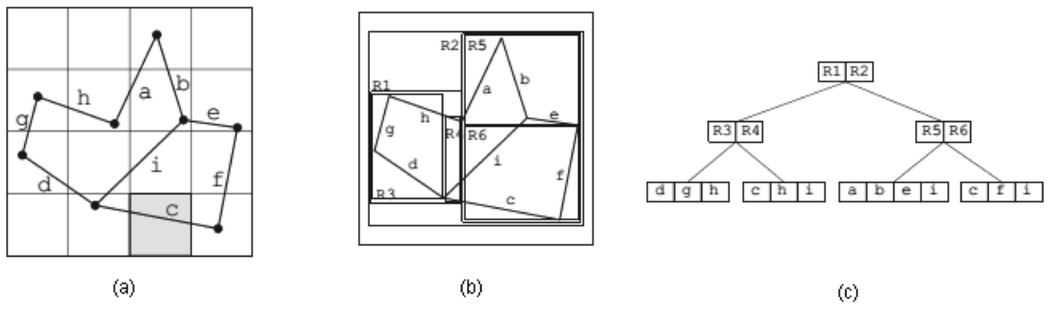


Figure (2.4) R^+-tree example, (a) line data. (b) MBR splits of the region with regards to the line data.(c) Corresponding R^+-tree.

2.3.4 X-tree

R^*-tree is not efficient enough to index high-dimensional data because of its higher overlap. X-tree (eXtended node tree) [24] is introduced to address this issue. Although, the structure still uses joint rectangles in the directory, the X-tree, as its name implies, tries to minimize the overlap by introducing super nodes.

Before introducing X-tree, there were two main approaches to deal with high-dimensional data [24]. The first approach assumes that high-dimensional data are highly correlated and as a result informative data are just in some subspaces of that high-dimensional space. Thus, they can transform high-dimensional data to lower dimensions without any data loss where data can efficiently be indexed using traditional methods. The second approach also assumes that in majority of high-dimensional data, a small number of dimensions have the most of information. Although, reducing dimension is an efficient technique and must be applied wherever possible. Practical results of this approach show that the dimension of reduced-space data is still too large.

The main idea of X-tree data structure is to minimize overlap of the directory by using a new organization of tree that is optimized for high-dimensional space [24]. Generally speaking, the main reason of overlap is splitting and X-tree tends to reduce the overlap by avoiding splitting. Instead, it extends directory nodes over normal block size by introducing super node. By increasing the size of a node, we take advantage of the fact that sequential search of entries in a super node is much faster than sequential search of all children nodes (overlap area) with additional overhead of many random page accesses.

The X-tree structure can be viewed as a hybrid approach that combines linear array with hierarchical R-tree-like structure [24]. In a very low dimensional space that typically there is no overlap or overlap rarely happens, most efficient organization is a hierarchical data structure such as R-tree. This is because selectivity in the directory is very high and the number of required page accesses is directly related to height of the tree. On the other hand, there are so many overlaps in high dimensional spaces and most of directories must be searched in order to answer queries. In the case of full overlap in which the whole directory search is needed, the linear directory structure requires less space and performs faster. But in medium dimensionality cases, a hybrid solution such as X-tree performs well.

Figure 2.5 [24]. illustrates the general structure of X-tree. The leaf nodes contain rectilinear MBRs together with pointers to the actual objects. The nodes in the middle or directory nodes contain MBRs together with pointers to their children. X-tree includes three different types of nodes: 1) data nodes or leaf nodes, 2) directory nodes, and 3) super nodes. Super nodes, as their names implies, have large variable sizes to avoid splitting to improve efficiency. In fact, super nodes are alternative to have highly overlapped children that results in inefficient directories. Despite R-tree with large node sizes, X-tree provides a heterogeneous structure that makes large nodes (super nodes) wherever it is required.

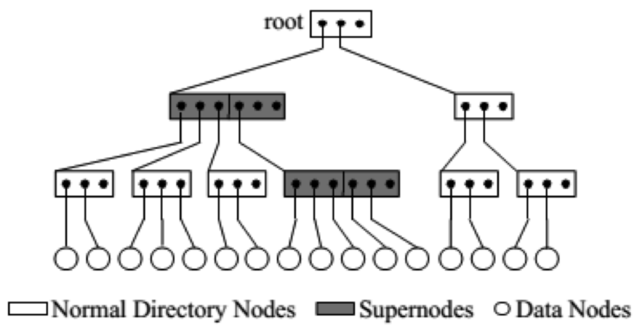


Figure (2.5) An example of X-tree data structure

Super nodes are created during insertion whenever there is no other way to avoid overlap and due to the fact that overlap increases as dimensionality goes higher, the internal structure of X-tree also changes with increasing dimension. Figure 2.6 [24] distinguishes these cases. Finally, two extreme cases of X-tree structure is explained in [24]. First, no directory has super node that the structure is the same as R-tree and happens in low dimensionality. Second, only there is one large super node (root) that happens in high dimensional space or highly-overlapped data. Experimental results presented in this paper shows that both page access and CPU time in X-tree has improved over R-tree and this improvement increases in higher dimensionality.

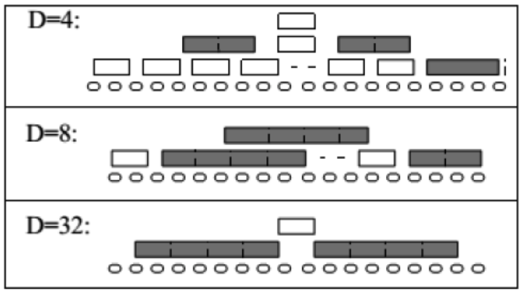


Figure (2.6) X-tree with different dimensionality.

2.3.5 KDB-tree

In some applications we want to answer range queries from a large dynamic indexed data stored in external memory given a k-dimensional key as $\{key_0, ..., key_{k-1}\}$. [26] addressed this problem using K-D-B-trees. A very special case of K=1 (one dimensional key) can be efficiently solved by B-trees or one of its variants. Also, in the case of static data (data which do not change frequently), one can use static data structures such as K-D tree or pseudo-dynamic techniques such as overflow areas for inserting data. K-D-B-trees are balanced trees with fixed-size nodes like B-trees, however unlike B-trees, 50% utilization cannot be guaranteed theoretically. Basically, the combination of some B-tree and K-D-tree properties makes this data structure and one can expect I/O efficiency of B-tree and multi-dimensional search efficiency of KD-tree in K-D-B-tree [26].

As mentioned above, each K-D-B-tree, in general has K dimensional key domains (point) as elements of $domain_0 \times domain_1 \times ... \times domain_{k-1}$ and each region can be defined as the set of points $(x_0, x_1, ..., x_{k-1})$ satisfying the following criteria [26]:

$min_i \leq x_i \leq max_i$, $0 \leq i \leq k-1$ where $min_i, max_i \in domain_i$

Also there are two types of pages in a K-D-B-tree:

- **Region pages**: a collection of (region, page ID) pairs.
- **Point pages**: a collection of (point, location) pairs where location is the address of the database record.

Every K-D-B-tree can be defined with the following set of properties [26]:

- Region pages can be neither empty nor contain null pointers.
- Point pages are the leaf nodes of the tree.
- The regions in each page must be disjoint.
- If both node and children are region pages, then, the union of the children regions must be equals to region page. If child page is a point page, then all the points must be in region.

The principle operations such as various queries, insertion, deletion, and reorganization are facilitated by introducing the concept of splitting of regions. Despite B-trees, insertion algorithm in K-D-B-tree forces pages at lower levels to split even though they are not overfull. Experimental results verify efficient search structure for large multi-dimensional indexes ($k \geq 2$. Moreover, full rage queries are quite efficient in this data structure, however, if objectives are partial match or partial range queries, other search structures such as K-D-tree might perform better [26].

2.3.6 Quad-tree

The major problem with all variations of R-tree data structures is that they are data-dependent and it is difficult to find the best composition for some data. Quad-tree address this dependency issue while they still provide disjoint adaptive decomposition of the space based on data distribution. The block sizes in the quad-tree are quadratic and also each side is a power of two. Furthermore, the block positions are predefined.

There are variants of quad-tree. They can be differentiated based on the type of data they represent (rectangles, points, regions, curves, surfaces and volumes), decomposition rules (regular decomposition or data-governed rules) and resolution (fixed or data-dependent) [23]. In the following, we explain each variant and classify them based on their properties and applications in the literature.

2.3.6.1 Region quad-tree Region quad-trees [33] can represent a region by its interior data based on recursive decomposition. The main idea of quad-tree is successive subdivision of the whole area to find the maximal blocks. Each block has a maximum capacity and as long as its size is not below the upper bound, it will be broken into smaller blocks. For example in 2D space, the square area is recursively divided into four equal-size areas. Quad-trees can store different data types. For example, Figure 2.7 [33] illusterates a binary data representation in a quad-tree. The recursive subdivision continues in each sub-square to reach to an entirely uniform area (consist of entirely $1's$ or $0's$) [33]. As shown, block sizes are adaptive based on data distribution, but their sizes are always a power of 2 as a result of consecutive halving of each block.

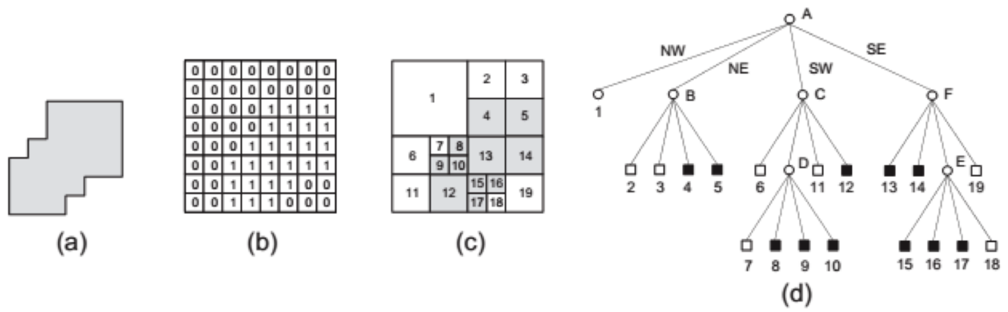


Figure (2.7) Region quad-tree example, (a) Simple region. (b) Binary representation of region. (c) Maximal blocks. (d) Corresponding quad-tree.

2.3.6.2 PR quad-tree PR quad-tree (Point Region) uses the same structure as region quad-tree to store point data [34]. Points are only stored at leaf level (one point in each node). The drawback of this data structure is that the height of the tree depends on the minimum separation between points. Bucketing technique can improve the structure. The idea is the decompose a block only if it contains more than a given threshold. PR quad-trees are popular in search applications. For example, given a record (point), determine all the records in a specific distance from it [23].

2.3.6.3 PM quad-tree PM-quad-tree [28] can store polygonal maps (it is actually an adaptation of the PR quad-tree). They provide a reasonably compact representation that satisfies the following three requirements:

1. Storing polygon maps without information loss.
2. Positioning of the map does not affect the structure drastically. For example, shift and rotation do not significantly increase the required storage for the map.
3. Data structure manipulation is efficient.

To address these requirements, they propose some decomposition criteria as follow:

- C1: At most one vertex can lie in a quad-tree leaf.
- C2: At most one q-edge can be in each leaf.
- C2': A region that has a vertex, cannot contain any q-edge that includes that vertex.
- C3: A region contains no vertex, can have at most one q-edge.
- C3': A region contains no vertex, can have only q-edges that meet a common vertex exterior to the region.

Based on these criteria, three PM-trees [23] are proposed (PM1 to PM3) summarized in the Figure 2.2 [28]. Figure 2.7 shows that the depth of tree from PM1 to PM3 is decreasing. In the best case (PM3), the only criterion is C1 (Figure 2.8.c). While PR quad-tree and PM quad-tree are similar in some aspects, PM3 stores more information in the leaf level without any information loss. PM quad-tree is efficient in storing dynamic line segments and point data types as well as map overlay algorithms [23].

PR quad-tree and PM quad-tree are similar in some ways. For example, when a point is located on the border of some areas, it is inserted in all nodes on whose border , in the case of PM quad-tree, when a line or part of it, falls on the border between two area, it is inserted in all nodes associated to that border. Although decomposition rules in PM3 and

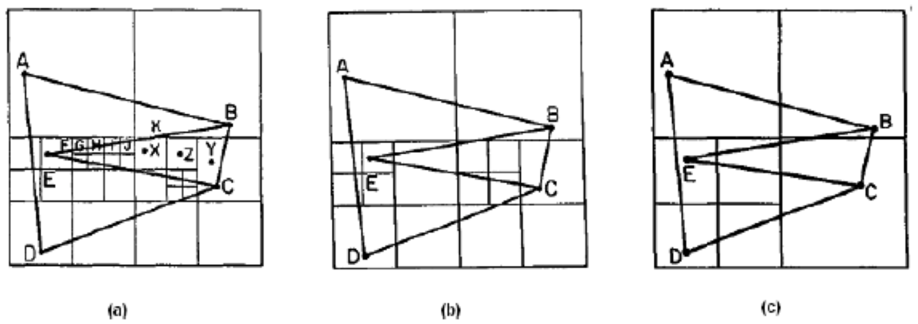


Figure (2.8) Different variants of PM quad-trees, (a) PM1 quad-tree meeting criteria C1, C2' and C3. (b) PM2 quadtree meeting criteria C1, C2' and C3'. (c) PM3 quadtree meeting criteria C1.

Table (2.2) The summary of PM1 to PM3 quad-tree data structures

Name	Criteria	Description
PM1	C1, C2', C3	1) More than one q-edge can be stored in a leaf nod. 2) There are different approaches to organize q-edges in a node. 3) Simplest organizing method is based on dictionary. 4) Depth of tree is maximum required. 5) Depth C3 depends on positioning of grid(may be too large).
PM2	C1, C2', C3'	1) The problem with C3 has been addressed. 2) Depth is bounded by maximum required by C1 and C2'. 3) Worst-case tree depth is less sensitive to depth of tree.
PM3	C1	Worse-case depth is less than two previous cases.

PR-quad-tree are almost same, in the case of PM3 more information is stored in the leaf nodes. Also, polygon maps can be stored without any information loss in the PM3. PM quad-tree is efficient in storing dynamic line segments and point data types as well as map overlay algorithms [23].

Finally, In the Table 2.3, R-tree and quad-tree are compared based on results presented in [35]. The creation time for the quad-tree is less than R-tree. As shown, R-trees are always better in terms of space requirements. Their efficiency also surpasses quad-tree for large polygon data, but for point data and small polygon data, quad-trees outweigh R-trees. Also, query time on large polygonal datasets is faster using R-trees than quad-tree.

Table (2.3) Efficiency of quad-tree and R-tree for different data types

	Faster Creation	Faster Update	Faster Insertion	Less Storage
Point	Quad-tree	Quad-tree	Quad-tree	R-tree
Polygon (small)	-	-	Quad-tree	R-tree
Polygon (large)	R-tree	R-tree	R-tree	R-tree

2.3.7 Pyramid-tree

Quad-tree is suitable to answer location-based queries because its data is indexed based on spatial information. On the other hand, there is no efficient way to answer feature-based queries (no feature indexing). The only approach is to traverse the whole tree to determine the regions satisfying a given feature value. This brute-force search algorithm is very time consuming and inefficient in practice. The pyramid data structure [23] addresses this problem.

Feature-based queries searching for answers without examining any particular location. Intermediate nodes in pyramid tree hold index of feature information of their sub-trees in a compact form. This makes the searching process faster and more efficient. For example, by looking at the root of every pyramid, one can easily find out whether a particular feature value presents in the whole tree or not. If value exists, the same process can be applied to sub-trees in a recursive manner until we find the given value or no more node is available to search. Figure 2.9 [23] shows a pyramid example over non-binary data. To find all the areas that are associated to feature value ”A”, we start processing from the root. Because ”A” exists in the feature list of root, the answer set is not empty. Then, we should process all the four children of the root. The only useful sub-tree to further processing is the last child. Doing so, we end up to two areas in this sub-tree as the answer of query. Pyramid data structure can eliminate so many unnecessary tree traverses because of feature indexes in the middle nodes.

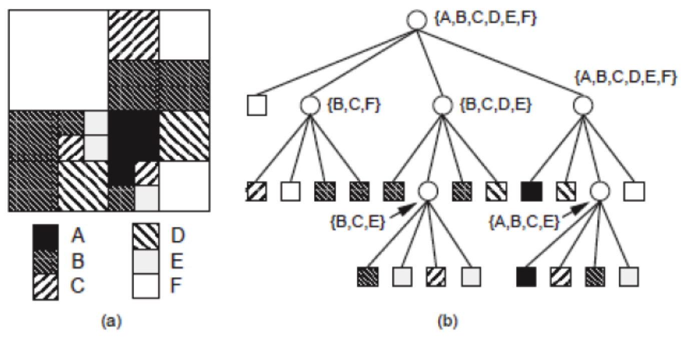


Figure (2.9) Pyramid example for non-binary data.

Table (2.4) Spatial indexed hierarchical data structures classification

Feature	Value	Hierarchical data structure
Overlap	Joint	R-tree, R^*-tree, KDB-tree, X-tree
	Disjoint	R^+-tree, PR quad-tree, PM quad-tree, Region quad-tree
Size	Fixed	PR quad-tree, PM quad-tree, Region quad-tree, Pyramid
	Adaptive	R-tree, R^*-tree, R^+-tree, KDB-tree, X-tree
Position	Predefined	PR quad-tree, Pyramid, PM quad-tree, Region quad-tree
	Random	R-tree, R^*-tree, R^+-tree, KDB-tree, X-tree

2.3.8 Classification of Hierarchical Indexing Data Structures

Table 2.4 classifies indexing data structures based on three parameters (overlap, size and position). In the following subsections, we explain each parameter.

2.3.8.1 Joint vs. disjoint data structures The datasets can be indexed based on joint or disjoint areas. For example, if a technique makes use of MBRs and they are overlapping in the leaf level the method is called joint (R-tree). Otherwise it is a disjoint technique (R^+-tree). In a disjoint method, the overlap parameter is zero, while in joint data structures some optimization techniques must be applied to minimize the overlap of bounding area in the leaf level.

2.3.8.2 Adaptive-size vs. fixed-size All the indexing hierarchical data structures decomposes the whole area based on some criteria. The splitting process can be either bottom-up or top-down, but all the data structures ultimately end up to the structure in which the data or their pointers is stored in the last level of the tree. In the splitting process, if the size of areas achieved from splitting is predefined based on some criteria, the method is called fixed-size, otherwise it is called adaptive-size. For example, R-tree (described later) is adaptive size while PM tree variants are fixed-size.

2.3.8.3 Adaptive-position vs. fixed-position The decomposition rules can be defined in a way that sub-areas resulted in each step are placed only in the predefined locations. In this case, the method is called fixed-position, otherwise, it is called adaptive position. The quad-tree is an example of a fixed-position data structure which bounding boxes can only be put in the locations with a power of two values.

2.4 Grid-based Spatial Join Techniques

Grid-based filters for spatial data has been deeply investigated over the past thirty years. Uniform grid methods for various spatial operations have been introduced in [36, 37, 38, 39, 40, 41] by Franklin. Although these works provide a strong theoretical and practical framework for applying non-hierarchical uniform grid approaches to reduce the workload of refinement phase of the most spatial join operations including but not limited to polyline intersection and polygon overlay, they have two major limitations. First, they are not specifically designed for many-core architectures such as GPUs mainly because these HPC technologies did not exist of that time. Second, they are built based on grid-cells of the same size that makes those algorithms less efficient in the presence of spatial data distributed non-uniformly over the whole area. In [42], an analytical model is presented to explore the effect of skewed data on parallel join algorithms. Their results confirm that data skew can have a significant impact on the performance of parallel algorithms. In the following paragraphs we summarize grid-based algorithms in the literature.

PBSM, Partition Based Spatial Merge spatial join operation [43] introduces a grid-based filter step. This method provides a good performance especially when neither of the input layers have an index. PBSM partitions MBRs data into non-overlapping uniform cells using a spatial partitioning function such as round robin or hash on tile number. Then it assigns each MBR to the cells and applies plane-sweep technique in each cell independently to find overlapping MBRs. Impact of various parameter changes on the performance of the algorithm is studied in this work. Although sequential implementation of PBSM shows a better performance than R-tree and Indexed Nested Loop-based algorithms, this work does not offer any improvement for the refinement phase that is the most computationally-intensive part of spatial join operation.

A parallel uniform-grid polygon overlay algorithm on CPU based on OpenMP and on GPU based on CUDA has been recently proposed in [44]. This algorithm works for any polygonal dataset including self-overlapping, degenerate, disjoint or objects with holes. This algorithm has space and time complexity of $O(N + K)$ where N and K are the number of edges and intersections respectively. They have reported up to 4 fold speedup on NVIDIA Tesla C2075 GPU over a 4-cores CPU implementation over synthesized and real datasets. They reported an average 16-fold speedup over ArcGIS software application. Their experimental results show that the best grid size must includes 10 to 20 edges per cell. They did not study this theoretically, beyond heuristic methods to achieve this optimized workload per cell. In general, determining the best cell size is one of the most challenging issues in all grid-based techniques. Experimental results presented in [45] show that optimal cell size is close to average edge sizes. They also state that a variation from this optimum value by a factor of 3 increases the running time only 20% to 50%. These statistics are provided based on their datasets and they did not provide any theoretical framework to generalize this assumption. Although [44] is specifically designed for GPU, the limitations of uniform grid, stated above, have not been addressed yet.

2.5 Spatial Join Systems

In this subsection, we have summarized related work focusing on the design and implementation of various types of join operations. Table 2.5 has summarized these techniques based on computing framework they are built upon. In the following subsections, we point out some of the centralized (CPU/GPU) techniques.

Table (2.5) Summary of Spatial Join Techniques.

	CPU-based	GPU-based
Centralized	Sequential / Multicore: [46, 15, 47, 48, 49]	CudaGIS: [50, 16] CMF-based: [51, 52] Others: [53, 54]
Distributed	MPI: MPI-GIS [55, 56, 57] Hadoop: SpatialHadoop [8, 58] Spark: SparkGIS [59, 60] Cloud (Microsoft Azure): Crayons [61, 62, 63]	MPI-cuda-GIS (Chapter 6)

2.5.1 Sequential and multi-core spatial joins algorithms

A sequential plane-sweep MBR filtering algorithm has been explained in [46]. In this paper, the basic idea is to sort lower boundaries of rectangles for plane sweeping in a recursive manner. An extensive performance evaluation on synthetic datasets with various ranges of properties including tall-shape or wide-shape rectangles is carried out in this paper. The results shows that the algorithm efficiency significantly depends on the width to height ratio of the rectangles. Some methods for dealing with tall/wide rectangles is described in [15]. GIPSY [48] is a novel approach for spatial join of two datasets with contrasting density to address space oriented coarse-grained partitioning challenges. GIPSY partitions the dense dataset using a method similar to STR [47] and then joins it with the non-indexed sparse dataset. Their extensive evaluation results using synthetic and real datasets yields up to 18-fold speedup. The main limitation of this algorithm is the assumption that one of the datasets is sparse.

A bottom-up spatial join approach based on CPU parallelism has been proposed in [49]. This algorithm does not rely on pre-existing spatial indices. The MBR join over Sequoia2000 dataset [64] (58411 by 20,974 size) takes more than 7 seconds on a platform with 40 processors.

2.5.2 GPU-based spatial joins algorithms / systems

A naive parallel implementation of spatial join using R-tree has been described in [53]. This top-down query search method runs less than 3 times faster on GPU than CPU on average and considering the CPU-GPU data transfer time, the performance is even worse than CPU implementation. Parallel spatial join using R-tree has been implemented in [50]. The GPU algorithm runs 8 times faster than multi core CPU implementation. A simple parallel r-tree query implementation on GPU is stated in [54] which runs 20 times faster than CPU. Another R-tree-based spatial join on GPU with less than 4-fold speedup has been reported in [53]. In [65], six spatial join queries has been implemented over GPU and they have achieved 6-10 fold speedup including transfer time from CPU to GPU. They have shown the significant impact of transfer time over speedup such that by not considering transfer time speedup could be as large as 318-fold. One of the fastest R-tree implementation and querying on GPU has been recently described by our group in [9]. We have proposed five algorithms for batch MBR querying and the best performance comes out of the modified-DFS algorithm, which employs all-to-all search on parents of the R-tree leaves. Speedup gain for querying algorithms is in the range 76-fold to 153-fold which is much higher than previous algorithms in the literature. However, a key limitation is the small datasets it currently handles because of $O(n^2)$ space complexity.

CudaGIS [16] exhibits 20-40 fold speedup versus sequential CPU implementation for spatial indexing and some spatial join operations. Uniform grid-based approach has also been exploited to create various indexing data structures such as R-tree, quad-tree, CSPT-P-tree and BMMQ-tree data structures. [66]. The main idea is to assign rectangles to grid cells and compute the operations locally in each grid cell. The speedup results show up to 20-fold improvement over CPU implementation for end-to-end system.

An Impala-based in-memory Spatial Processing system has been designed in [67]. Most of their library has been implemented on top of Thrust parallel library in Cuda SDK. Single node performance test of their framework demonstrates speedup less than 2 for two different datasets.

2.5.3 Distributed spatial join systems

2.5.3.1 Load balancing for distributed systems The load balancing problem in different field of computer science including network and HPC has been long established. Researchers have tried to look at this problem from different angels from practical and experimental perspectives to theoretical aspects. In this section, we summarized some of the most significant theoretical load balancing works. We pointed out their pros and cons and their limitations compared to our proposed framework.

A general static load balancing model for job scheduling over a distributed computer network to minimize the mean response time of a job is proposed in [68]. They formulated the load balancing task as a nonlinear optimization problem with $n(n+1)/2$ variables where n is the number of nodes. Optimal solution is presented using a Lagrange multiplier approach. They also provided two efficient algorithms that determine optimal load for each host. parametric-study algorithm that generates optimal solution as a function of communication time and single-point algorithm that gives optimal solution for given system parameters including communication time. The framework has some limiting assumptions. First, they did not consider the problem of partitioning a big job into small task. In fact, they assumed all nodes have the same processing capabilities and a job may be processed completely at any node in the system. However, considering the big data era and jobs processing huge volume of data, this is an impractical assumption. Second, the model assumed the communication between different nodes is one way that is if node A transfers a job to node B, no job can be sent to node A from B. Third, as name implies, the framework is static and decision to transfer a job from one node to another does not depend on the state of the system.

[69] proposed a framework for load balancing iterative computations in clusters with heterogeneous computing nodes. The model assumes that application data is already partitioned between the processing nodes forming a virtual ring. In each iteration, the computation involves independent calculations carried out in parallel in each node followed by exchanging the partial results between neighboring nodes in the virtual ring. The goal is

to select a subset of processing nodes from all nodes and balance the load between them such that it minimizes execution time while these nodes are not fully connected and pairs may share physical communication links due resource limitations. The model considered two scenarios, 1) SharedRing that there may exist several messages sharing a link, 2) SliceRing that dedicated links are used for communications. Some heuristic algorithms is provided in this work to solve these optimization problems. The main limitation of this model is that it is suitable only for application iterative computations with local partial results exchange in each step. In fact, it is limited to ring topologies. Moreover, the problem of big data IO and initial partitioning of data between nodes is not addressed.

Dynamic load balancing on message passing multiprocessor has been studied in [70] as diffusion schemes. They provided converging conditions as well as convergence rate for arbitrary topologies. Hypercube network analysis is provided as a case example and they showed that diffusion approach to load balancing on a hypercube topology of multiprocessors is inferior to dimension exchange method. This well-presented model has several limitations. First, they quantified work in terms of *tasks* and assume all *tasks* require an equal amount of computational time and nonuniform task partitioning of heterogeneous data is not addressed. Second, although the model is designed for any topology it did not consider spatially related tasks that means it is not suitable for load balancing applications that need to maintain locality.

In [71] and [72] two data migration-based load balancing models are provided. In [71] a load balancing framework called Ursa is proposed for large scale cloud storage services. It formulates an ILP optimization problem that chooses a subsets of objects from heavy-loaded nodes called hot-spots and performs topology-aware migration to minimize latency and bandwidth. Ursa is designed to identify cost-optimal source-destination node pairs for dynamic and scalable load reconfiguration by applying 1) a workload-driven integer linear programming optimization approach to eliminate hot-spot nodes while minimizing reconfiguration costs, and 2) a divide and conquer technique to break down expensive computations to provide scalability. While this work provides a practical framework for load balancing in cloud storage services, because it is designed at a storage layer it is not application-aware and does not consider data locality or other application-specific requirements for distributing the work load between nodes. Furthermore, it assumes that architecture is organized as a spanning tree topology that makes unsuitable for other network architectures. SWAT [72], a load balancing algorithm, is proposed to address the problem of performance isolation of multi-tenant databases in cloud systems that caused by resource sharing among co-located tenants. Similar to [71], the general idea is to select tenant pairs for load-swap in a highly resource and time efficient manner. SWAT initially tries to eliminates all the hotspots and balance the load among all the nodes by *load leveling*. If it is not possible to balance the load, then, it eliminates the hotspots through *hotspot elimination* process. Finally, in case both load balancing and hotspot elimination fail, SWAT tries to minimize the overload rather than eliminating it by *hotspot migration*. Partitioning problem and maintaining locality while swapping workload have not been addressed in this work.

Chapter 3

GCMF: AN END-TO-END SPATIAL JOIN SYSTEM OVER LARGE POLYGONAL DATASETS

3.1 Introduction

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. For instance, Light Detection and Ranging (LiDAR) systems produced 40 PB of data in 2014 but domain scientists were able to handle only $30PB$ of data [3]. Researchers have predicted LiDAR technology will generate up to $1,200$ PB of data by 2020, while GIS workforce has the capacity to process only 50 PB. This demonstrates ever-increasing demand for High Performance Computing (HPC) in GIS domains. In particular, GPUs are very popular among the HPC technologies as they are widely available at low prices yet with powerful features. For example, GeForce GTX 1080, the latest NVIDIA GPU released in May 2016, is driven by the new NVIDIA Pascal$^{\text{TM}}$architecture which provides 2560 *Cuda Cores* operating at 1607 MHz base clock and 8 GB of the main memory with 320 GB/sec bandwidth that makes it feasible to handle larger data in a real time manner.

Spatial join is one of the most computationally intensive operations in spatial computing. For instance, spatial join of a polyline table with 73M records representing the contiguous USA with itself takes roughly 20 hours to complete on an Amazon EC2 instance [2]. Therefore, harnessing parallel processing capabilities of modern hardware platforms to perform join operation over big spatial datasets is essential. In general, spatial join can be defined as follows: given two spatial datasets R and S and a spatial join predicate $\bowtie$ (e.g., overlap, contain, intersect) as input, spatial join returns the set of all pairs (r, s) where $r \in R,\ s \in S$, and $\bowtie$ is true for (r, s) [10]. A typical application of a spatial join is "Find all pairs of rivers and cities that intersect." The focus of this paper is on polygonal data with *ST_Intersect* operation in which for a given pair of polygons, it returns true if and only if polygons share any portion of space [11].

Generally, spatial join algorithms over polygonal data follow a two-phase paradigm [10]:

- 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 refinement phase is significantly time-consuming. For instance, an analysis of join operation on CPU over more than 10,000 spatial objects in [14] 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. In this work, we plan to bridge this gap by introducing *GCMF*, a GPU-based spatial join system including both filtering and refinement steps. Our work can be distinguished in two ways: 1) To the best of our knowledge, there is no such system to process end-to-end polygonal intersection-based join over GPU, and, 2) comparable systems that proposed other spatial join predicates such as k-Nearest Neighbor [73] cannot handle the amount of data that we are able to process on a single GPU and they have reported less speedups.

In summary, our key contributions in this work are:

- *GCMF*: An end-to-end spatial join system built on a single GPU to generate cross-layer polygon pairs from two large datasets that satisfy spatial join condition in near real-time manner. Based on our experimental results, *GCMF* was able to handle real datasets as large as (not limited to) $3GB$ files up to 39 times faster than an *optimized GEOS* library within a few seconds.
- A sort-based MBR filtering algorithm with a suitable GPU-specific data structure that

yields up to 20-fold speedup compared to *optimized GEOS* library. Proof of correctness of this algorithm is also provided.

- Common MBR Filter (CMF) based on the MBR resulting from the intersection of MBR_{P1} and MBR_{P2} that makes *refinement* phase 28 times faster than the same implementation without *CMF* by 1) reducing the number of candidate polygon pairs by up to 66% and 2) making the polygon pairs 40-fold smaller in size by removing many of the non-intersecting edges. We provide proofs and performance analysis for this filter.

- A load-balanced implementation of parallel *point-in-polygon* test that is up to 9 times faster compared to the naive implementation over GPU. It also achieves 30-fold speedup compared to sequential implementation over CPU.

The remainder of this paper is organized as follows. In the next section, we summarize the current work in the literature with a focus on performance of algorithms. Then, in Section 3.2, we introduce *GCMF* system overview, its components, algorithms and theoretical analysis. Experimental results are presented in Section 5.5. Finally, we provide conclusions and point out our future work plan.

3.2 Algorithms

In this section, we present the problem definition and notations, datasets employed, the overall system design, and the filtering and refinement algorithms.

3.2.1 Problem Definition

Given a polygon P, $MBR_P = (x_{P,0},\ y_{P,0},\ x_{P,1},\ y_{P,1})$ is the minimum bounding rectangle of P that can be described by its bottom-left coordinate $(x_{P,0},\ y_{P,0})$ and top-right coordinate $(x_{P,1},\ y_{P,1})$. We also use x_P (or y_P) to refer to x-coordinates of MBR_P regardless of being left or right coordinate ($x_P \in \{x_{P,0},\ x_{P,1}\}$). 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.

As stated before, 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.

3.2.2 Datasets

We have used two real polygonal dataset pairs (*Urban-Admin*, *Water-Block*) from http://www.naturalearthdata.com and http://resources.arcgis.com from GIS domain with various sizes and characteristics (Urban-Admin and Water-Block). The third dataset (*Telecom*) comes from telecommunication domain. The details of the datasets are provided in Table 6.2. All the datasets are available online at the project site at http://grid.cs.gsu.edu/ daghajarian1/SpatialJoin.html in both shapefile and text formats.

Table (3.1) Three real datasets used in our experiments.

Label	Dataset	Polygons	Size
Urban-Admin	*ne_10m_admin_states*	11,878	$46MB$
	ne_10m_urban_areas	4,646	$41MB$
Telecom	*GA_telecom_base*	101,860	$171MB$
	GA_telecom_overlay	128,683	$240MB$
Water-Block	*US_block_boundaries*	219,831	$2.175GB$
	US_water_bodies	463,591	$921MB$

3.2.3 System Design Overview

ST_Intersect predicate requires both *edge-intersection* and *point-in-polygon* tests. Figure 3.1 illustrates a typical workflow of the spatial join algorithm which has been used in the literature. 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.

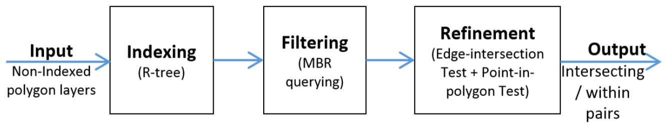

Input
Non-Indexed polygon layers
Indexing
(R-tree)
Filtering
(MBR querying)
Refinement
(Edge-intersection Test + Point-in-polygon Test)
Output
Intersecting / within pairs

Figure (3.1) Typical spatial join processing pipeline

Edge-intersection test is more compute-intensive than *point-in-polygon* test. We take advantage of this fact in our system design workflow by adding one more filtering phase based on Common MBR. Figure 4.1 shows the overview of the *GCMF* system. *GCMF* has two subsystems. The first subsystem includes two filtering components. The first is Sort-based MBR Filter (*SMF*) which reduces set of all cross-layer polygon pairs into the set of potentially intersecting polygon pairs (**C**) by overlap-test over their minimum bounding rectangles. The second component is Common MBR Filter (*CMF*) that applies intersection test to edges of each pair in **C** and their common MBR to classify polygon pairs into following three groups: 1) *Intersecting-Edge candidate set* (**I**), 2) *within candidate set* (**W**) and 3) disjoint pairs which can be discarded. We explain these two filters in more details in the following subsections. 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 4.1, 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.

3.2.4 Sort-based MBR Filtering

Tree-based data structures such as R-trees and interval trees have been used in MBR filtering. While these data structures are perfectly matched to sequential algorithms, they are not suitable currently for large datasets over GPUs mostly because of their hierarchical

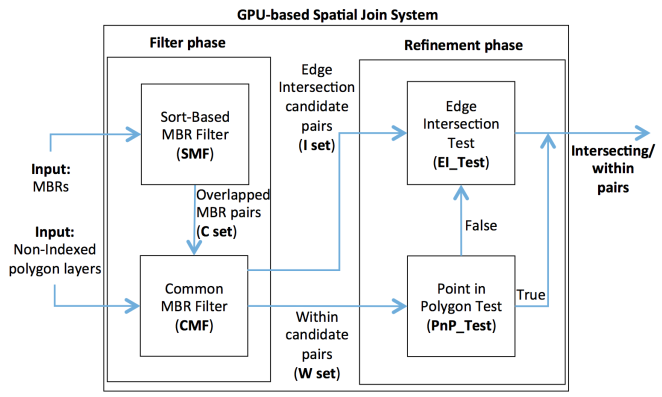

GPU-based Spatial Join System
Filter phase
Refinement phase
Input: MBRs
Input: Non-Indexed polygon layers
Sort-Based MBR Filter (SMF)
Overlapped MBR pairs (C set)
Common MBR Filter (CMF)
Edge Intersection candidate pairs (I set)
Within candidate pairs (W set)
Edge Intersection Test (EI_Test)
Point in Polygon Test (PnP_Test)
False
True
Intersecting/ within pairs

Figure (3.2) GCMF system design overview

structures and memory usage in current implementations. To address this issue, we introduce *SMF* which is a sort-based MBR filter algorithm, highly suitable for GPUs in particular.

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**. If two MBRs overlap then their (interval defined by) x-coordinates overlap and their y-coordinates overlap. In essence, this algorithm looks for two interval overlaps in x and y dimensions. It sorts the x coordinates of the MBRs from both layers (set **X**). Then for each MBR_i with x coordinates $(x_{i,0}, x_{i,1})$, it finds all the MBR_j from the other layer with x-coordinates $(x_{j,0}, x_{j,1})$ such that
$x_{i,0} \leq x_{j,0} \leq x_{i,1}$. Then MBR_i is tested against all such MBR_j for overlap in their y coordinates, thus yielding the output set C. Same can be done by sorting the y coordinates and then testing in x-dimension. Later, we prove that this algorithm neither generates a duplicate pair nor misses one.

Algorithm 1 describes the Sort-based MBR Filter suitable for GPUs. As mentioned above, **X** is a vector of x-coordinates of the MBRs in both layers. *CRadixSort* is our customized radix sort function which generates two vectors:

- *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 $\mathbf{X}$ in $O(1)$ time 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 $0 \leq i \leq m+n-1$:

$$rankIndex[\ sortIndex[\ i\ ]\ ] = isortIndex[\ rankIndex[\ i\ ]\ ] = i \tag{3.1}$$

To better understand the data structure, Figure 6.1 provides an example. Part (b) is $\mathbf{X}$ for 4 MBRs presented in part (a). For any MBR_i, $x_{i,0}$ and $x_{i,1}$ can be accessed through $2 \times i$ and $2 \times i + 1$ indices of $\mathbf{X}$ respectively. For instance, the first two values are left and right x-coordinates of MBR_0. Parts (d) and (e) represent *sortIndex* and *rankIndex*, respectively.

Generating *rankIndex* has two advantages. 1) For a given MBR_i, we can access its sorted indices in O(1). For example, position of $x_{3,0}$ at *sortIndex* (equal to 1) is the 6th element of $rankIndex$. 2) This vector is helpful for balancing the load. If we want to figure out how many elements may potentially lie in between an MBR range (an estimation of load of the block handing that MBR), we can subtract its corresponding values in the *rankIndex* vector. The value gives us an upper bound which also can be used as a relative measure of number of overlapping MBRs for a given MBR. For example, MBR_3 includes just one element in its x-interval $\{x_{0,0}\}$, while MBR_1 has three $\{x_{3,0}, x_{0,0}, x_{3,1}\}$.

To implement sort-based MBR filter, we launch a kernel with $m + n$ (total number of MBRs in both layers) blocks and each block handles the interval corresponding to one MBR. By evenly distributing the load among the threads within a block, we make the implementation load-balanced. In GPU $Block_i$, algorithm finds all the elements $x_{j,0} \in X$ which lie between $x_{i,0}$ and $x_{i,1}$ using *rankIndex* and *sortIndex* (Line 5 in Algorithm 1). Then if they also intersect in y-coordinate, the block produces (i, j) pair as output if MBR_i is the firs-layer MBR (Line 8), otherwise, (j, i) is generated (Line 10).

Lemma 3.2.1 proves the correctness of sort-based MBR filtering algorithm.

Lemma 3.2.1. ***SMF***: *Given two sets of MBRs,* ***R*** *and* ***S****, Algorithm 1 will generate all*

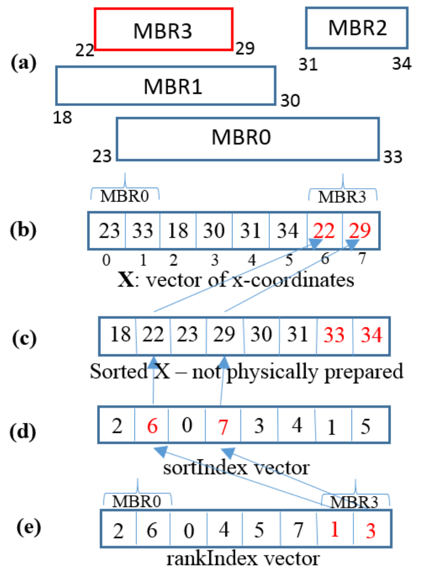


Figure (3.3) An example of data structure used for sort-based MBR filter. Part (a) is actual MBRs, (b) is **X** set, (c) is sorted **X**, (d) is sorted indices of **X** and (e) is sorted indices of MBRs. For example index of 2-th smallest coordinate (which is 22) can be retrieved from $sortIndex[1] = 6$. Part (e) is $rankIndex$ which keeps track of MBRs in $sortIndex$. For example, position of $(x_{3_0}, x_{3,1})$ (red MBR)in $sortIndex$ can be fetched from $rankIndex[3 \times 2] = 1$ and $rankIndex[3 \times 2 + 1] = 3$

overlapping MBR pairs without any false positives or duplicates.

Proof. The first part of proof can be derived from the algorithm by showing that $\mathbf{O} \subseteq \mathbf{A}$ and $\mathbf{A} \subseteq \mathbf{O}$ where $\mathbf{O}$ is output of algorithm and A is pairs of overlapping MBRs. For the second part, lets assume MBR pair (i, j) is generated twice. Pair (i, j) may be generated by $Block_i$ or $Block_j$. As such, one of these two blocks may generate this pair twice or each of the blocks may generate only one of the duplicated pairs. First case is impossible as we process each MBR just once in its corresponding block and no coordinate is duplicated in the data structures. Let us assume both $Block_i$ and $Block_j$ reports (i, j) as output. Given this, $Block_i$ implies that $x_{j,0}$ appears after $x_{i,0}$ and before $x_{i,1}$ in **X**. $Block_j$ also requires that $x_{i,0}$ appears between $x_{j,0}$ and $x_{j,1}$ in **X** at the same time which is impossible. Therefore, each intersecting pair (i, j) is exactly generated once in the output. $\square$

Algorithm 1 Sort-based MBR filtering algorithm

```
    Input: R and S set of MBRs        Output: set C

        Building data structure
 1: let $\mathbf{X} = \{x_i | x_i \in x\text{-}coordinate\ \ of\ \ \mathbf{R} \cup \mathbf{S}\}$
 2: $(sortIndex\ , rankIndex) \leftarrow$ CRadixSort($\mathbf{X}$)

 3: procedure (Filter for R and S MBRs)
 4:     for each $GPU\,Block_i,\ 0 \leq i < (m+n)$, do in parallel
 5:         for each $x_{j,0},\ \ x_{i,0} \leq x_{j,0} \leq x_{i,1}$ do
 6:             if $(y_{j,0}, y_{j,1})$ intersects $(y_{i,0}, y_{i,1})$ then
 7:                 if $MBR_j \in S$ then
 8:                     Add pair $(i, j)$ to the output set
 9:                 else
10:                     Add pair $(j, i)$ to the output set
11:                 end if
12:             end if
13:         end for
14:     end for
15: end procedure
```

3.2.4.1 SMF Analysis For average case analysis, we assume that average height and width of MBRs are $\bar{h}$ and $\bar{w}$ respectively and they are scattered in a $H_a \times W_a$ rectangle area. The sequential time complexity of *CRadixSort* is $O(n \cdot b)$ where b is the average number of digits of the coordinate values. Also, Algorithm 1 includes two nested loops. The outer loop has $n + m$ iterations. The inner loop goes through all the x-coordinates lying in the range $(rankIndex(x_{i,0}),\ rankIndex(x_{i,1}))$ for a given MBR_i which on the average has d_{avg} elements ($\bar{w}/W_a$ fraction of (n+m) MBRs).

$$d_{avg} = \frac{\bar{w} \times (n+m)}{W_a} \tag{3.2}$$

Thus, the algorithm's sequential complexity is:

$$= O((n+m) \cdot b + \frac{\bar{w}}{W_a} \cdot (n+m)^2) \tag{3.3}$$

The complexity of the algorithm depends on $\frac{\bar{w}}{W_a}$ factor which is proportional to the number of output pairs. If $\frac{\bar{w}}{W_a} = O(\frac{1}{n+m})$ then the second term in equation 4.10 becomes linear and therefore total complexity becomes $O((n+m) \times b)$. This is what usually happens in real datasets. If we have $\frac{\bar{w}}{W_a} = O(1)$, the complexity would be order of $O((n+m)^2)$. One of the scenarios that may lead to $O((n+m)^2)$ complexity, is the case with $\bar{w} \approx W_a$ which means each MBR is almost as wide as the entire area and therefore has potential overlap with almost all the other MBRs.

SMF has linear space complexity. As described in the preamble of Section 3.2.4, it estimates the maximum number of overlapping MBRs for a given MBR and then allocates the memory in advance. Although this estimation has some time-overhead, applying this strategy makes it possible for *SMF* to test large MBR layers for intersection. *SMF* could process two datasets each including more than $1M$ MBRs on a GPU node with 6 GB of the main memory.

3.2.4.2 SMF Performance

We have used a sequential optimized *GEOS* library as baseline to compare with *SMF*. Table 3.2 shows our experimental results using all three datasets introduced in Section 4.4.1.

Table (3.2) Running time of *SMF* and GEOS for MBR filtering

Dataset	Running time (ms)		♯ of Outputs
	GEOS	SMF	
Urban-Admin	197	16	28,687
Telecom	2,683	240	747,086
Water-Block	13,048	676	1,020,458

The results in this table shows up to 20-fold speedup. As we explained in the related work, RTree method introduced in [9] can achieve more speedup versus *SMF*, but it cannot query datasets with more than around 20,000 MBRs in the second layer because the memory space requirement of its current implementation is $O(n{\cdot}m)$ due to matrix based data structure for $O(1)$ access.

Also to show the scalability, Figure 3.4 plots *transfer* time and *SMF* running time versus input size in the largest dataset. As shown, when input size becomes larger, *transfer* time linearly increases while *SMF* growth is closer to linear order than $O((n+m)^2)$ which implies $\frac{\bar{w}}{W_a} = O(\frac{1}{n+m})$.

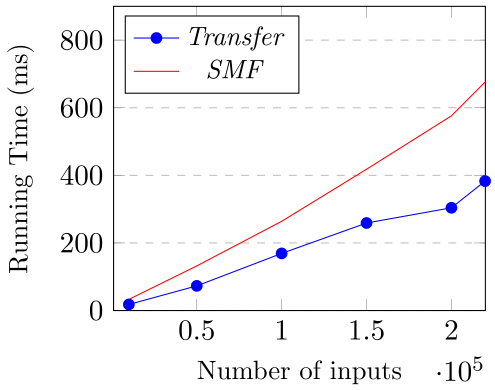


Figure (3.4) *Transfer* time and *SMF* running time for various input sizes. *SMF* is almost linear.

3.2.5 Common MBR 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 the Common MBR. Given a pair $(P1, P2) \in \mathbf{C}$, with corresponding MBRs, MBR_{P1} and MBR_{P2}, their Common MBR ($MBR_{P1 \cap P2}$) is defined as the area covered by both of them (see green rectangles in Figure 3.5).

Algorithm 2 shows how *CMF* eliminates more polygon pairs from *SMF* output set $\mathbf{C}$ and classifies the remaining pairs into two groups for *point-in-polygon* and *edge-intersection* tests for the refinement phase while it eliminates all the non-intersecting edges from each polygon which does not intersect with the respective Common MBRs. The correctness of Algorithm 2 will be shown through Lemma 3.2.2 and 3.2.3.

Lemma 3.2.2. ***CMF-Pre-PnP Test**: Given polygon pair $(P1, P2) \in \boldsymbol{C}$ with corresponding minimum bounding rectangles MBR_{P1} and MBR_{P2}, if $P1$ contains $P2$, then MBR_{P1} contains MBR_{P2}. In other words, $MBR_{P1 \cap P2} = MBR_{P2}$.*

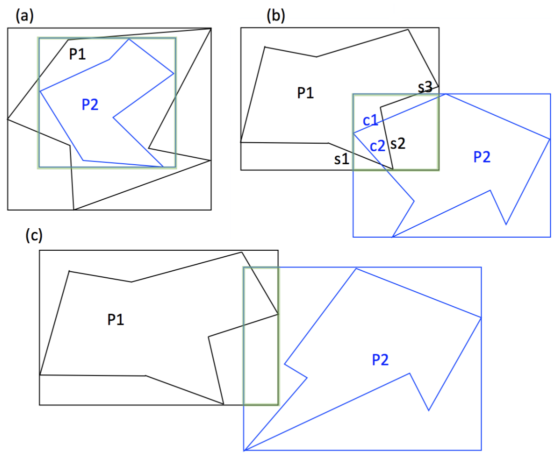


Figure (3.5) Examples for three *CMF* output classes: **(a)** $MBR_{P1\cap P2} = MBR_{P2}$ and CMF will tag $(P1, P2)$ for "P1 contains P2" *point-in-polygon* test. **(b)** $MBR_{P1\cap P2} \neq MBR_{P1}$ and $MBR_{P1\cap P2} \neq MBR_{P2}$ and $P1 \cap MBR_{P1\cap P2} \neq \emptyset$ and $P2 \cap MBR_{P1\cap P2} \neq \emptyset$, 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.

Proof. MBR_{P1} contains P_1 from definition. Also P_1 contains P_2 from Lemma assumption. Applying transitive property over contain relation leads to MBR_{P1} contains P_2.

Now lets assume $MBR_{P1\cap P2} = MBR_c \neq MBR_{P2}$. Because P_2 is inside of both MBR_{P1} and MBR_{P2}, it is also inside of MBR_c and since MBR_{P2} contains MBR_c and $MBR_c \neq MBR_{P2}$, MBR_c is minimum bounding rectangle of P_2 that is on the contrary with lemma assumption (minimum bounding rectangle of P_2 is MBR_{P2}). Thus, $MBR_{P1\cap P2} = MBR_{P2}$ □

Lemma 3.2.2 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 3.5). The first two classes can be added to *within candidate set* **W** for actual *point-in-polygon* test. By applying Lemma 3.2.2 before doing this test over all **C** elements, we gain performance due to the following reasons:

- For any given pair, verifying whether Lemma 3.2.2 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.

Algorithm 2 Common MBR filtering algorithm

```
    Input: set C        Output: sets W and I

 1: procedure CMF-FILTER
 2:     for each pair $(i, j) \in \mathbf{C}$ do
 3:         if $MBR_{i\cap j} == MBR_i$ then
 4:             $\mathbf{W} \leftarrow \mathbf{W} \cup (i, j)$ for polygon i inside polygon j test
 5:         else if $MBR_{i\cap j} == MBR_j$ then
 6:             $\mathbf{W} \leftarrow \mathbf{W} \cup (i, j)$ for polygon j inside polygon i test
 7:         else
 8:             $\hat{E_i} \leftarrow \{E_i(k) \mid E_i(k)\ intersects\ MBR_{i\cap j}\}$
 9:             $\hat{E_j} \leftarrow \{E_j(k) \mid E_j(k)\ intersects\ MBR_{i\cap j}\}$
10:             if $|\hat{E_i}| == 0$ or $|\hat{E_j}| == 0$ then
11:                 Discard (i,j)
12:             else
13:                 $\mathbf{I} \leftarrow \mathbf{I} \cup (i, j)$
14:             end if
15:         end if
16:     end for
17: end procedure
```

- Join predicate requires testing for both "P1 contains P2" and "P2 contains P1" cases, but Lemma 3.2.2 identifies which polygon may contain the other one that eliminates one unnecessary test.

Later in Section 3.2.5.1, we provide more analysis.

Lemma 3.2.3. ***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, their intersection point is not outside $MBR_{P1\cap P2}$.*

Proof. For the explanation refer to [10]. □

Lemma 3.2.3 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 3.2.2 and 3.2.3 and it can be used to detect some disjoint pairs in **C** before refinement phase.

Corollary 3.2.3.1. *Given pair* $(P1, P2) \in \boldsymbol{C}$*, 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* $(P1, P2) \notin \boldsymbol{W}$*, the within candidate set, and* $\hat{E}_{P1} = \emptyset$ *or* $\hat{E}_{P2} = \emptyset$

Proof. It can be derived directly from Lemma 3.2.2 and Lemma 3.2.3. □

To illustrate Lemma 3.2.2 and 3.2.3 and Colorrary 4.2.1, three different examples are shown in Figure 3.5. Figure 3.5 (a) shows a case for Lemma 3.2.2 where $MBR_{P1 \cap P2} = MBR_{P2}$. In this example, *CMF* assigns $(P1, P2)$ to *within candidate set* for point-in-polygon test for "P1 contains P2" case. In Figure 3.5 (b), $MBR_{P1 \cap P2}$ is equal to none of the MBRs. As such, using Lemma 3.2.3, *CMF* makes *intersecting-edge candidate sets* for $P1$ and $P2$ that are $\hat{E}_{P1} = \{s1, s2, s3\}$ and $\hat{E}_{P2} = \{c1, c2\}$ respectively. In this case, because neither of $\hat{E}_{P1}$ and $\hat{E}_{P2}$ is empty, *CMF* categorizes $(P1, P2)$ into *Intersecting-Edge candidate set.* Later, *EI_Test* will use only $\hat{E}_{P1}$ and $\hat{E}_{P2}$ instead of E_{P1} and E_{P2} for refinement phase. Figure 3.5 (c) is an example of Corollary 4.2.1 as $(P1, P2) \notin \mathbf{W}$ and $\hat{E}_{P2} = \emptyset$. Therefore, *CMF* identifies this pair as disjoint and just discards it.

CMF thus classifies elements of $\mathbf{C}$ as follows:

1. *Within candidate set* ($\mathbf{W}$): 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* ($\mathbf{I}$): 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 $\mathbf{W}$ nor in $\mathbf{I}$.

As *CMF* iterates through each edge once, the algorithm complexity is $O(n_e)$ which n_e is number of edges and its implementation is straightforward. For a polygon pair $(P1, P2)$, we assign a two dimensional GPU block to handle edges of each polygon in a separate block dimension. Polygon edges are evenly distributed among threads in each dimension to make the algorithm load-balanced. Each thread verifies Lemma conditions for its data and partially keeps list of potentially intersecting edges. Finally, using shared memory and reduction tree, algorithm classifies pairs in **C** based on results and prepares *intersecting-edge candidate sets* $\hat{E}_{P1}$ and $\hat{E}_{P2}$.

3.2.5.1 CMF Analysis

For a given candidate set **C**, we define *edge-reduction factor* as

$$R_E = \frac{\sum_{(i,j)\in\mathbf{C}} |E_i| + |E_j|}{\sum_{(i,j)\in\mathbf{C}} |\hat{E}_i| + |\hat{E}_j|} \tag{3.4}$$

In the worst-case, all edges may lie inside their *common MBR* or intersect it and $R_E = 1$, but based on our experimental results, shown in Table 4.3, $R_E \approx 40$ which shows effectiveness of *CMF* in pruning polygons before refinement phase. Table 4.3 shows timing and workload of *edge-intersection* test with and without *CMF*. *CMF* eliminates almost two-third of pairs by applying Lemma 3.2.3. It also, makes *edge-intersection* test almost 30 times faster by making *intersecting-edge candidate set* 40 times smaller than all the edges.

Table (3.3) CMF effect on reducing workload of the refinement phase for Water-Block 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

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

Lemma 3.2.4. ***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.*

Proof. Equation 3.5.a and 3.5.b represent the number of operations in *point-in-polygon* test with and without *CMF* filter.

$$k \cdot |\mathbf{C}| + \bar{N}_e \cdot |\mathbf{W}| \qquad (a) 2 \cdot \bar{N}_e \cdot |\mathbf{C}| \qquad (b) \tag{3.5}$$

where $\bar{N}_e$ is the average number of edges in polygons and k is a constant factor such that in general, $k < \bar{N}_e$ ($k \approx 8$ as we only need to test for two MBR equalities each with 4 coordinate values). We need to show $3.5.a <\ 3.5.b$. Since $0 \leq |\mathbf{W}| \leq |\mathbf{C}|$ and $k < \bar{N}_e$, we have $\bar{N}_e \cdot |\mathbf{W}| < (2 \cdot \bar{N}_e - k) \cdot |\mathbf{C}|$. As such the condition always holds true. □

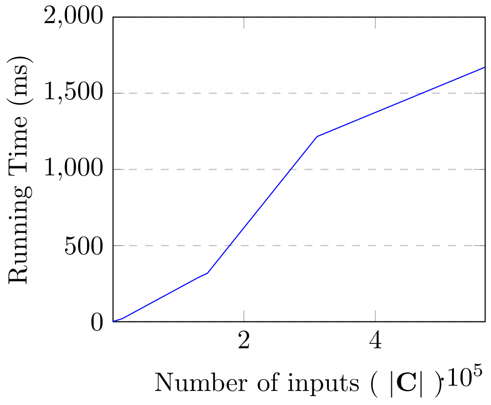


Figure (3.6) Running time of *CMF* versus various range of input sizes.

Figure 3.6 shows *CMF* running time for various data sizes. As shown, *CMF* running time increases linearly as data size becomes larger.

3.2.6 Refinement Algorithms

The refinement phase removes all the non-intersecting polygon pairs by finding the pairs with intersecting edges or detecting if one of the polygons lies inside the other one. Sequential plane sweep-based edge-intersection algorithms are generally used in the refinement phase but these methods have not been proven suitable for fine-grained data parallel processing over GPUs [44]. While some work has been done to parallelize the plane sweep on CPU [74], none of the proposed candidates result in an algorithm amendable to fine-grained SIMD parallelism such as with GPUs. Our approach is embarrassingly parallel. For a given polygon pair (i, j), it performs an all-to-all *edge-intersection* test with $O(|\hat{E}_i| \cdot |\hat{E}_j|)$ time complexity.

As we described in Section 3.2.3, our refinement phase includes two subsystems. Although we have not developed any new algorithm for these two components, efficiently parallelizing their sequential counterparts over GPU is not a trivial task and requires some design changes to fit them into shared memory model. In the following subsections, we explain some of their implementation details.

3.2.6.1 Parallel Point-in-Polygon Test

To detect if a polygon is inside the other polygon, we have used *point-in-polygon* test. We apply crossing test method to detect if a given test point is inside a polygon [75]. The technique is to shoot a ray from the test point along an axis and count number of crossings of the polygon edges to check if it is odd. Sequential implementation of this algorithm uses a *for loop* that iterates through all vertices of a given polygon. We have implemented this algorithm over GPU by breaking down this *for loop* by distributing equitably among different threads of a GPU-block. Finally, using a reduction tree algorithm, we combine the partial results from different threads.

We compare load-balanced PnPTest with 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 the entire *for loop* and there is no workload distribution for a given test point. The results are summarized in Table 3.4. *PnPTest* speedup is in the 28 to 30-fold range compared to the sequential version. Our load-balanced *point-in-polygon* test also achieved a good speedup in 8 to 9-fold range versus naive GPU version which demonstrates the importance of GPU-load-balancing.

Table (3.4) Running time of *PnPTest* versus sequential

Dataset	$\hat{N}_e.\|\mathbf{W}\|$	Running time (ms)		
		Seq.	GPU	
			Naive	PnPTest
Urban-Admin	11,266,110	672	210	24
Telecom	7,615,041	165	45	6
Water-Block	105,314,500	22,058	6,459	725

PnPTest running time of *Urban-Admin* dataset is greater than *Telecom* dataset while number of polygons in *Telecom* is much larger because as shown in Lemma 3.2.4, the time complexity of our *point-in-polygon* algorithm is $O(\hat{N}_e.|\mathbf{W}|)$.

3.2.6.2 Parallel Edge-Intersection Test As we explained in Section 3.2.3, elements of **I** and those pairs from **W** which do not pass *point-in-polygon* test successfully along with *intersecting-edge candidate sets* generated by *CMF* are sent to *edge-intersection* test component. We implemented a load-balanced *edge-intersection* algorithm using a shared memory model. The algorithm assigns a GPU-block to each pair of polygons. Within a block, edge pairs are distributed evenly among threads. For actual intersection test, the algorithm calculates intersection point of a given edge pair, then tests if this point lies on the both edge segments (and not outside them). We have implemented this algorithm efficiently by applying two optimization techniques:

- Calculating line intersection requires floating point computations which is the most time consuming operation in any processor. To optimize the algorithm, first we test if MBRs built from edges have overlap and then, calculate intersection point for only the MBR-overlapping edge pairs.

- All threads within a block work over the same polygon pair and once a thread finds two cross-layer intersecting-edges, the polygon pair can be identified as output. As a result, if a thread detects such a case, it sends signal to other threads in the block to terminate. This makes their resources available for other blocks.

3.2.6.3 Refinement Analysis Figure 3.7 shows running time for *PnPTest* and *EITest* for various input sizes.

As shown, *PnPTest* has a linear running time while *EITest* is $O(n^2)$. As *point-in-polygon* test iterates through all vertices of a given polygon, linear complexity is expected. On the other hand, the complexity of *EITest* is quadratic, because *edge-intersection* algorithm, for a given pair $(P1, P2)$ with *intersecting-edge candidate sets* $\hat{E}_{P1}$ and $\hat{E}_{P1}$, tests for all to all

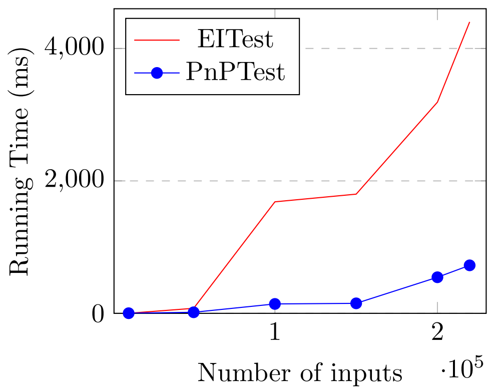


Figure (3.7) Linear running time of *PnPTest* versus quadratic complexity of *EITest* for different input sizes.

cross-layer edge-intersection. As a result, complexity is $O(|\hat{E}_{P1}| \cdot |\hat{E}_{P2}|)$.

3.3 Performance Evaluation

First, we describe the experimental setup and then we compare the results with other base-line methods.

3.3.1 Experimental Setup

We have done all the GPU experiments on a compute node that has 12-core Intel Xeon CPUE5-2650 CPU running at a clock speed of 2.0 GHz with 64GB of main memory. The node is equipped with a NVIDIA GTX 780 GPU that has 6GB of memory with 288.4 GB/sec memory bandwidth. We have verified that similar results are obtained on other NVIDIA GPU's including *Tesla K40* models.

To the best of our knowledge, there is no GPU-based work which has implemented $ST\_Intersect$ operation. Therefore, we used PostgreSQL version 9.4 with PostGIS version 2.2 and *GEOS* library version 3.4.2 [11] as sequential baselines for comparison with *GCMF*. PostGIS is a spatial database extender for the PostgreSQL object-relational database. It adds support for geographic objects allowing spatial queries to be run in SQL. We ran PostGIS on a desktop with 3.6 GHz processor with 16 GB of main memory. Since our datasets are shapefiles, we use *shp2pgsql* tool in *PostGIS* for converting shape files into database tables. $ST\_Intersect$ predicate was used in spatial join query [11]. We also used $Intersects$ method of *PreparedGeometry* class that is the optimized and indexed implementation of Geometry class of *GEOS* $C++$ library. *GEOS* experiments are done on a node equipped with *2.6 GHz Intel Xeon E5-2660v3* processor in the *Roger NCSA* cluster.

3.3.2 Results

Table 3.5 shows end-to-end running time of different algorithms on different datasets including CPU-GPU transfer times. The relative speedup gain for our GPU-based system is up to 39-fold versus both *GEOS* library and *PostGIS* software.

Table (3.5) End-to-end running time for three different methods

Dataset	Running time (ms)			♯ of Outputs
	PostGIS	GEOS	GCMF	
Urban-Admin	3,120	5,770	149	23,634
Telecom	17,900	8,200	560	581,351
Water-Block	232,122	148,040	7,856	539,974

3.3.2.1 System Component Analysis We have provided detailed running time of each system component in Table 3.6 and Figure 3.8 for all datasets. As value ranges are large, time in the bar chart is presented in the logarithmic scale for easier comparison.

Although by applying *CMF* filter, we tried to reduce *edge-intersection* processing time, this component still takes more than half of the total running time.

Table (3.6) Detailed running time of system components for all three datasets

Dataset	Detailed running time (ms)				
	Transfer	*SMF*	*CMF*	*PnPTest*	*EITest*
Urban-Admin	12	16	53	24	44
Telecom	55	240	89	6	170
Water-Block	383	676	1671	725	4401

Our real datasets are heterogeneous in size. They include polygons with various ranges, from less than 100 vertices up to 50,000 vertices, that makes load-balancing task hard.

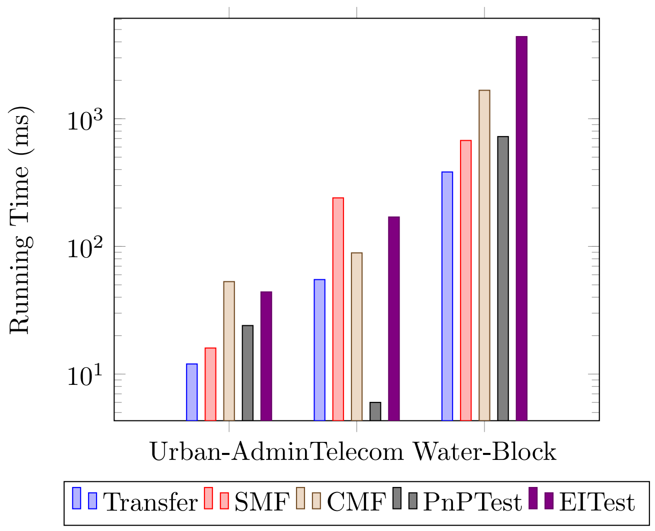


Figure (3.8) Detailed running timing of system components for Urban-Admin, Telecom and Water-Block datasets (log scale)

We tried evenly distributing the *edge-intersection* test over all threads across all blocks by assigning a constant number of tests to each thread. But because of inefficient use of memory bandwidth and other GPU resources, the performance was worse than our simpler current method. Determining the impact of this load-imbalance on the performance requires comprehensive understanding of the behavior of GPUs under these circumstances that is part of our future work.

Table (3.7) Running Time and Workload Reduction of Each Component for Water-Block dataset

	SMF	CMF	PnP	EI
Space Complexity	$O(k_1)$	$O(k_2)$	$O(n)$	$O(n)$
Time Complexity	$O(n \cdot b + k_1)$	$O(n)$	$O(n)$	$O(n^2)$
n: # of inputs (Polygon pairs)	$219,831 \times 463,591$	1,020,458	842,516	198,142
k: # of outputs (Polygon pairs)	1,020,458	198,142	453,802	86,172
Reduction ($\frac{k}{n}$)	$> 99\%$	65%	46%	57%
Time Fraction	8.6%	21.27%	9.23%	56.02%

Finally, Table 3.7 summarizes time and space complexities of each phase of *GCMF* system. In the filtering phases, *GCMF* estimates the required memory space for the output and then allocates the memory. To estimate space, we have used two strategies. 1) Roughly estimating the number of outputs in *SMF*. 2) Counting the exact number of outputs and then allocate the memory in *CMF*. Therefore, for these components, the space complexity is proportional to their corresponding outputs k_1 and k_2, respectively. Our space-optimized design lets the system handle large datasets on a single GPU. Also, time complexity of most of the phases is linear. *Edge-intersection* is an exception, however, because of effective linear filters (*SMF* and *CMF*) applied before it, workload of this operation significantly reduces. As Table 3.7 shows, for *Water-Block* dataset, more than 99% of all possible polygon pairs are eliminated by *SMF*, and then *CMF* further reduces 65% of the remaining pairs. Finally, the total running time of the filter phases is less than one-third of the overall execution time, which shows the effectiveness of the proposed filtering methods. However, *Edge-intersection* test is still the most time-consuming operation as it takes more than 50% of the execution time.

3.4 Conclusion

In this paper, we have introduced *GCMF*, an end-to-end spatial join system for non-indexed polygonal data over a single GPU platform. The system included 4 subsystems: two filtering components as well as *point-in-polygon* test and *edge-intersection* test subsystems. We proposed sort-based MBR filtering algorithm for GPU with linear average time complexity. Also, we introduced *CMF* with linear time complexity as an efficient filtering technique to reduce the number of polygon candidate pairs before the refinement phase. We also have shown that *CMF* reduces the size of remaining candidate pairs by pruning disjoint edges apriori. Our experimental results over real datasets yielded up to 39-fold relative speedup gain versus optimized sequential *GEOS* library and *Postgres* with *PostGIS* spatial database system. Moreover, it confirmed the efficiency of CMF in removing about two-third of pairs from the set of candidate polygons before *edge-intersection*. It also reduced the size of polygon pairs for refinement phase up to 40-fold smaller.

Our plan is to integrate this system into a MPI based system which can partition 1-3 order larger datasets among the compute nodes, such as our MPI-GIS system [55] [4] that has a potential for speeding up such systems by 1-2 orders of magnitude by effectively employing

GPUs.

Chapter 4

CMF-GRID: A NON-UNIFORM PARALLEL GRID-BASED FILTER

4.1 Introduction

The ever increasing volume of spatial data from various communities representing geographic location of features and boundaries, medical images or traffic on the one hand and the crucial need of realtime processing of these datasets in order to extract helpful information on the other hand makes it necessary to exploit High Performance Computing (HPC) in Geographic Information System (GIS) domains [1]. For instance, Light Detection and Ranging (LIDAR) systems will generate up to $1,200$ PB of data by 2020 [3]. Currently, the primitive-like overlay operation over two layers of spatial object, including less than *700,000* polygons using state of the art ArcGIS software takes more than 13 minutes on a single node [4]. To address these challenges, researchers have designed several distributed architectures to make HPC computing available for geospatial processing including cloud-based systems [5, 6], Message Passing Interface (MPI) systems [7, 55], and map-reduce systems [8]. Most of these systems make use of powerful and expensive computing clusters to break down the computations over several distributed nodes. Some of these applications handle tremendous volume of spatial data which requires using many nodes. Even with such parallelism, employing only CPUs in modern heterogenous architectures, typically equipped also with GPU, one to two orders of speedup remains unharnessed [9]. 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). Effective employment of CPU-GPU pair is critical for real-time spatial processing.

Generally, spatial join algorithms over two layers of polygonal (vector) datasets follow a two-phase paradigm [10]:

- Filtering phase: reduces all possible pairs of cross-layer spatial objects to a smaller set of potentially intersecting candidate pairs based on some computationally light algorithms such as Minimum Bounding Rectangle (MBR) overlap test.

- Refinement phase: removes any results produced during the filtering phase that do not satisfy the join predicate, such as ST_intersect, overlap, union, overlay, and etc.

The refinement phase is significantly time consuming as it typically involves $O(n^2)$ algorithms. Our *GCMF* algorithm recently reduced the refinement phase processing time for edge-intersection test down to about 50% of the total time [51]. Older analysis of join operations on CPU show that refinement phase can take up to five times more than the rest of the operations including filtering and parsing datasets [14]. In this work, we introduce *CMF-Grid*, a non-uniform grid-based filtering technique that is specifically designed based on GPU architecture to further reduce refinement processing time.

In our previous work, we designed our *GCMF* spatial join system based on 2-step filter and 2-step refinement phase as shown in Figure 4.1. A MBR sorting-based filter step identifies the set C of all pairs of cross-layer MBRs, which are potentially intersecting. The set C is further filtered by eliminating those pairs of MBRs which do not have edges from both polygons in the intersection of the two MBRs, called *Common MBR*. The set I of surviving pairs is processed for join predicate. Special caution is taken for spatial pairs wherein an MBR is inside the other (W set). Finally, *GCMF* processes only those polygonal edges from set I which are within the common MBR.

In this work, we delve 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. As a case study, we also present an improved grid-based algorithm for intersection test with ST_intersect operation.

Our key contributions in this chapter are:

- ***CMF-Grid***: A GPU-based non-uniform grid technique over *Common MBR*, the MBR resulting from the intersection of MBR_{P1} and MBR_{P2}, that can significantly reduce

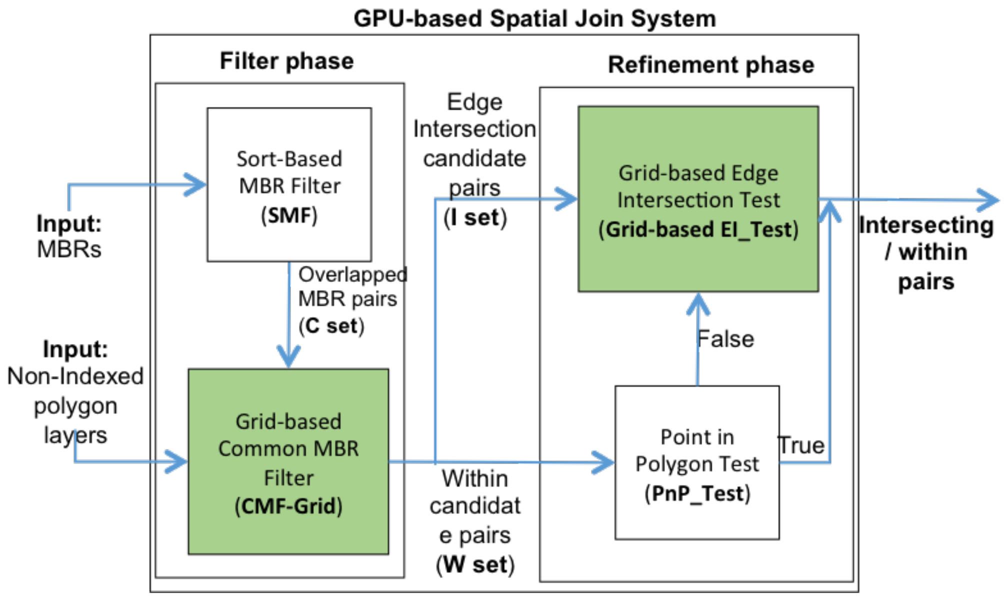

GPU-based Spatial Join System
Filter phase
Refinement phase
Input: MBRs
Sort-Based MBR Filter (SMF)
Overlapped MBR pairs (C set)
Input: Non-Indexed polygon layers
Grid-based Common MBR Filter (CMF-Grid)
Edge Intersection candidate pairs (I set)
Within candidate pairs (W set)
Grid-based Edge Intersection Test (Grid-based EI_Test)
False
Point in Polygon Test (PnP_Test)
True
Intersecting / within pairs

Figure (4.1) 2-step filter and 2-step refinement upgraded spatial join system with two new components (green boxes).

the workload of the refinement phase. It reduces this phase to the same extent as uniform grid with at least three times coarser grid-cells that implies less parallel overhead. It 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 [51].

- **Theoretical analysis**: We provide a mathematical analysis to estimate the refinement phase workload after applying *CMF-Grid* and provide proof of correctness for the algorithm.

- **Experimental analysis**: We present an extensive experimental analysis to study the impact of grid-cell size and grid-cell shape on the performance of *CMF-Grid* technique. We show 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.

- ***ST_intersect* spatial join use case**: To demonstrate the efficiency of *CMF-Grid* technique, we refine a GPU-based system for spatial join based on *ST_intersect* operation. The experimental results show that by applying *CMF-Grid* this system is 225% faster than *GCMF* [51], which is the state of art. The proposed system can process end-to-end spatial join over more than 600,000 polygons with over 2 billions edges in less than a second on a single *NVIDIA P100* GPU.

- **GPU-based edge-intersection**: A load balanced grid-based edge intersection test over GPU that is 10-times faster than its GPU-based predecessor in [51]. The significant improvements made by *CMF-Grid* reduce the edge-intersection processing time from more than 50% in *GCMF* to less than 15% of the total processing time for spatial join with ST_intersect operations.

The remainder of this chapter is organized as follows. In the next section, we summarize the relavant work in the literature about different grid techniques. Then, in Section 4.2, we introduce the *CMF-Grid* technique and a theoretical analysis of this algorithm. In Section 4.3, we present a spatial join use case with ST_intersect operation based on *CMF-Grid* to study the efficiency of this technique. Experimental results are presented in Section 5.5. We conclude the chapter and point out our future work plan in Section 4.5.

4.2 CMF-Grid

The main contribution of this chapter is to introduce a novel adaptive grid technique to further filter out potential pairs of spatial objects before the refinement phase to reduce total end-to-end running time. In this section, we introduce *CMF-Grid* technique including motivation, algorithm, proof of correctness and theoretical analysis.

4.2.1 Motivation

As we explained in Section 2.4, uniform grid algorithms are not among the best techniques for datasets with unevenly distributed spatial objects to achieve the highest performance. They perform well only if objects are distributed uniformly over the universe where by applying a uniform grid we can have equal size grid-cells with almost balanced workload per cell for further refinement phase in-cell all-to-all processing. Furthermore, classic grid-based techniques are not designed for many-core parallel architectures.

To address these issues, we introduce *CMF-Grid*, a non-uniform grid-based filter applicable to *Common MBR* area of potentially overlapping pairs. Figure 4.2 illustrates the difference 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 4.2-b).

2. In *CMF-Grid*, grid-cells may overlap with each other (in cases that *Common MBRs* of overlapping pairs overlap). Later on, we see that such overlaps are infrequent and therefore do not impact performance.

CMF-Grid is carefully designed to be an embarrassingly parallel algorithm to exploit key aspects of many-core parallel architectures.

Later in Section 4.4.3, we compare efficiency of our proposed technique with uniform grid.

4.2.2 Problem Definition and Notations

In this section, we state the notations and problem definition. Table 4.1 has summarized list of symbols that are used for the rest of this chapter.

CMF-Grid can be formulated as an adaptive grid method as follows. For every pair of polygons i, (P_{i_1}, P_{i_2}) in set C, set of MBR overlapping polygon pairs, we want to break down $CMBR_i$ into N_{g_i} equal-size cells (grid-cell) of w_{g_i} by h_{g_i} size and identify all the edges of P_{i_1} and P_{i_2} that lie in each cell. Let $E^k_{i_1}$ and $E^k_{i_2}$ be list of edges in cell k from P_{i_1} and P_{i_2} respectively for $0 \leq k < N_{g_i}$. we refer to these edges as edge-cells (edges that belong to cells). It is worth mentioning that each edge can belong to zero to several grid-cells. We want to partition each *Common MBR* in a way that the total workload for in-cell all-to-all refinement phase becomes small enough to minimize end-to-end processing time.

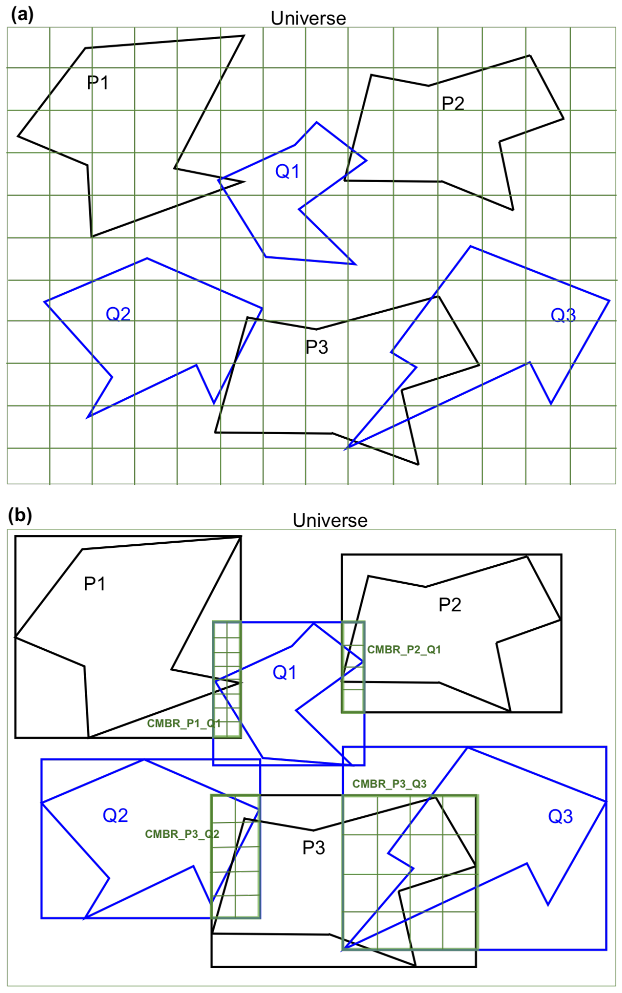


Figure (4.2) (a) uniform grid technique, and (b) *CMF-Grid*: In *CMF-Grid*, grid-cells are not of the same size and may not cover the whole universe.

As you can see in Table 4.1 for every pair i, $(P_{i_1}, P_{i_2}) \in C$, let $\hat{E}_{i_1}$ and $\hat{E}_{i_2}$ be the list of all the edges of P_{i_1} and P_{i_2} which lie either completely or partially inside $CMBR_i$. Let $\hat{\bar{E}}_{i_1}$ and $\hat{\bar{E}}_{i_2}$ be partial average of these two sets, respectively. The reason we refer them as partial average is that for those edges that are not completely inside $CMBR_i$, we only consider the portion of the length that is inside the *Common MBR*. Therefore:

$$\hat{\bar{E}}_l = c_l \cdot \bar{\hat{E}}_l = \frac{c_l}{|\hat{E}_l|} \sum_{j=1}^{|\hat{E}_l|} \hat{E}_l(j) \quad , \qquad for \quad l = i_1,\ i_2 \tag{4.1}$$

where $0 < c_l \leq 1$.

Note that grid-cell sizes in each *CMBR* are fixed while different *CMBRs* have different

Table (4.1) List of primitive symbols.

Symbol	Description
Set **C**	Pairs of MBR-overlapping polygons: (P_{i_1}, P_{i_2})
MBR_P	Minimum Bounding Rectangle of polygon P
E_P, $E_P(i)$	Edges in polygon P, and $E_P(i)$ is the i-th edge
$\bar{E}_P$	Average length of edges in polygon P
$CMBR_i$	$\forall\ pair\ i = (P_{i_1}, P_{i_2}) \in C$ The minimum bounding rectangle of MBR_{i_1} and MBR_{i_2}, also $MBR_{i_1 \cap i_2}$
w_i, h_i	Width and height of $CMBR_i$
$\hat{E}_{i_1}$	$Pair\ i = (i_1, i_2) \in C,\ \forall\ edge\ j \in E_{i_1}$ such that j intersects with/lies inside $CMBR_i$
$\hat{\bar{E}}_{i_1}$	$c_l \cdot avg(\hat{E}_{i_1})$: Partial average of edges in $\hat{E}_{i_1}$
E_i^k	List of edge-cells in $cell_k$ of $CMBR_i$
N_{g_i}	Number of cells in $CMBR_i$
w_{g_i}, h_{g_i}	Width and height of cells in $CMBR_i$
$k_i = \frac{w_{g_i}}{h_{g_i}}$	width-to-height ratio of cells in $CMBR_i$
$N_{g/e}$	Average grid-cells that an edge crosses
$W_i^E(w_{g_i})$	Average-case estimated refinement phase workload of pair i with w_{g_i}
$W_{total}^E(C)$	Average-case estimated refinement phase workload of the pairs in set C
$W_{total}^A(C)$	Actual refinement phase workload of the pairs in set C

grid sizes.

4.2.3 Grid-CMF Algorithm

For any given pair i in C, *CMF-Grid* partitions the $CMBR_i$ into N_{g_i} grid-cells and generates $2N_{g_i}$ lists, E_l^k where $l = i_1, i_2$ and $0 < k \leq N_{g_i}$ (one list for each polygon per cell), such that they hold all the edge-cells belonging to each cell separately. Then, potentially edge-intersecting polygon pairs can be identified as set of all polygon pairs $(i_1, i_2) \in \mathbf{C}$ such that $\exists k$ such that E_{i1}^k and $E_{i_1}^k$ are non-empty. Algorithm 3 provides an abstract view of *CMF-Grid*. Lemma 4.2.1 shows the correctness of *CMF-Grid*.

Lemma 4.2.1. ***CMF-grid Pre-Edge-intersection Test**: Given a pair i, $(P_{i_1}, Pi_2) \in$ **C**, P_{i_1} and P_{i_2} are not edge-intersecting if $\forall$ k $E^k_{i_1} = \emptyset$ or $E^k_{i_2} = \emptyset$ for all grid-cells k.*

Proof. Grid-cells, $cell_k$ where $1 \leq k \leq N_{g_i}$, are a partition of $CMBR_i$. Then $\forall$ k $E^k_{i_1} = \emptyset$ or $E^k_{i_2} = \emptyset$ means that if there is any intersecting edge from P_{i_1} and P_{i_2} the intersection point does not lie inside $CMBR_i$. However based on Lemma 3 in [51] this is not possible. □

It is worth mentioning that Lemma 4.2.1 is a stronger condition than Lemma 3 in [51]. We will see in Section 4.4.4 that *CMF-Grid* can filter out up to 700 times more edge-pair tests before refinement phase compared to *CMF* filter in [51] as a result of this stronger lemma.

Algorithm 3 CMF-Grid Algorithm

```
    Input: C            Output: Set I, Lists of edge-cells ($E^k_i$).
 1: procedure CMF-GRID FILTER
 2:     for each pair $(i_1, i_2) \in C$ do
 3:         $E^k_{i_1} \leftarrow \{E_{i_1}(l) \mid E_{i_1}(l)$ lies partially inside $Cell_k\}$
 4:         $E^k_{i_2} \leftarrow \{E_{i_2}(l) \mid E_{i_2}(l)$ lies partially inside $Cell_k\}$
 5:         if $\exists k \mid (E^k_{i_1} \neq \emptyset$ && $E^k_{i_2} \neq \emptyset)$ then
 6:             $I \leftarrow I \cup (i_1, i_2)$
 7:         else
 8:             Discard $(i_1, i_2)$
 9:         end if
10:     end for
11: end procedure
```

4.2.4 GPU-based Implementation

To better understand how we take advantage of GPU architecture to accelerate this algorithm, we provide some more detailed explanations as follows. Since processing of each pair in *CMF-Grid* is independent of others, we can process them in different asynchronous GPU-blocks. Furthermore, if we have access to more than one GPU, workload can be distributed among multiple GPUs. Even elements of of a pair (layer 1 and layer 2 objects) can be processed independently in different GPUs. These features make this technique embarrassingly parallel.

The key part of Algorithm 3 is how to design a parallel algorithm that calculates E_i^k and E_j^k lists in lines 3-4. There are two main approaches for this as follows:

1. **Cell-oriented**: In this technique, for each cell in a given pair i, (P_{i_1}, P_{i_2}), we launch a parallel thread in which it identifies the edges that are completely/partially inside that cell. Since each cell needs to iterate through all the edges of that pair, the total computations in each pair would be $O((|E_{i_1}| + |E_{i_2}|) \cdot N_{g_i})$.

2. **Edge-oriented**: In this method, edges are distributed among parallel threads and for each edge the corresponding thread determines the cells in which that edge passes through. For a given pair i, the computations would be $O((|E_{i_1}| \cdot N_{g/e_1} + |E_{i_2}|) \cdot N_{g/e_2})$ in this technique.

Although, the memory requirement of both designs is equal to $O((|E_{i_1}| \cdot N_{g/e_1} + |E_{i_2}|) \cdot N_{g/e_2})$ for a given pair of polygons, the time required for *Cell-oriented* method is greater than *Edge-oriented* ($N_{g_i} \gg N_{g/e_{p1}} + N_{g/e_{p2}}$ for all the pairs). On the other hand, GPU implementation of *Cell-oriented* design is more straightforward. In other words, computations within each thread (handling just one cell) are independent of others in a block and this makes *Cell-oriented* algorithm a lock-free data structure. However, in *Edge-oriented* technique, different thread-blocks may happen to work on the same cell simultaneously. As a result, implementation of *Edge-oriented* method requires use of locks that could potentially impact the performance of the code. However, due to light and efficient design of atomic operations in *CUDA* libraries, *Edge-oriented* implementation of *CMF-Grid* outperforms *Cell-oriented* approach over modern GPUs.

To implement either *Cell-oriented* or *Edge-oriented* technique, first we need to the count number of edges belonging to each cell and assign enough memory to each cell for storing the edge-cells. This is because assigning a maximum fixed memory to every cell is impractical due to memory limit of GPUs. Therefore, the algorithm applies a 2-step count-and-compute approach which is fairly common for GPU implementations [76, 51]. In the first step, it just counts the number of edges inside each cell without storing them. Then, it assigns required memory to all the cells, and, in the second step, it stores the edge-cell.

Algorithm 4 GPU-based Edge-oriented CMF-Grid

```
1: procedure EDGE-ORIENTED CMF-GRID (FIRST POLYGON OF $pair\ i$:$(i_1, i_2)$, $thread\ t$)
2:     ePT=$\left\lceil \frac{|E_{i_1}|}{|blockDim|} \right\rceil$
3:     $w_{g_i} = x \cdot min(\hat{\hat{E}}_{i_1}, \hat{\hat{E}}_{i_2})$
4:     $N_{g_i} = Function(w_{g_i})$
5:     for each $E_{i_1}(k) : ePT \cdot t \leq j < ePT \cdot (t+1))$ do
6:         if$(E_{i_1}(j) \cap CMBR_i = \emptyset)$continue
7:         Let $(a_1, b_1)$ and $(a_2, b_2)$: two end points of $E_{i_1}(j)$
             truncated by $CMBR_i$ boundaries.
8:         Let $cell_{(a_1,b_1)}$ the cell that $(a_1, b_1)$ belongs to.
9:         Let $cell_{(a_2,b_2)}$ the cell that $(a_2, b_2)$ belongs to.
10:        for each $cell_k$ between $cell_{(a_1,b_1)}$ and $cell_{(a_2,b_2)}$ such
               that $cell_k \cap E_{i_1}(j) \neq \emptyset$ do
11:            $E^k_{i_1} \leftarrow E^k_{i_1} \cup j$
12:        end for
13:    end for
14:    if $\exists k \mid |E^k_{i_1}| \neq 0$ then
15:        $I \leftarrow I \cup (i_1, i_2)$
16:    else
17:        Discard $(i_1, i_2)$
18:    end if
19: end procedure
```

Algorithm 4 is a thread-level *edge-oriented* implementation of *CMF-Grid* over GPU. The ePT variable in Line 2 holds the number of edges that need to be processed in each thread (edges are evenly distributed among all the threads in a thread-block). $|blockDim|$ is the number of threads in each block. N_{g_i} is calculated from Equation 4.3 as a function of w_{g_i}. Later in Section 4.4.3.1.2, we explain about calculation of w_{g_i}.

Each thread partially iterates through the corresponding edges of the given polygon. First, it makes sure that the edge is inside the *Common MBR* area and then it truncates two ends of the edge by $CMBR$ boundaries. As you can see in Figure 4.3 the left end of edge E is truncated by point (a_1, b_1) on the left side of $CMBR_i$. However, the right end does not need truncation.

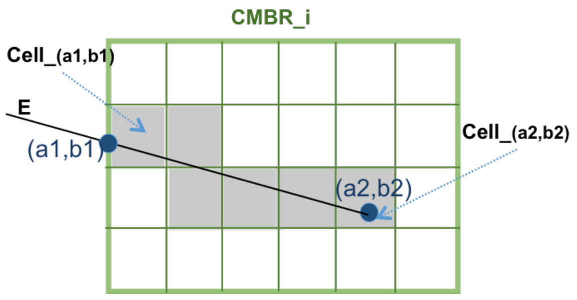


Figure (4.3) Edge-oriented *CMF-Grid* algorithm truncates each edge by $CMBR_i$ ((a_1, b_1) to (a_2, b_2)). Then, it identifies the cells the edge belongs to.

4.2.5 Theoretical Analysis of CMF-Grid

Here, we provide analysis for *CMF-Grid* technique. As mentioned earlier, the goal is to minimize the end-to-end running time by reducing total refinement phase workload. Our aim is to calculate the workload of *CMF-Grid* as a function of w_{g_i} (grid-cell width) based on average case analysis. In order to do so, we need to borrow some concepts from *Buffon's Needle* problem [77]. The method we apply here is inspired by the one used in [40] with two major differences. First, grid-cells are not necessarily of square shape (different height and width) and, second, grid-cell sizes are not equal in the whole universe (adaptive grid). *CMBRs* can be very skewed in shape depicted in Figure 4.2-b. To see the impact of this skewness on the grid-cell sizes and to generalize the formulation, we assume w_{g_i} and h_{g_i} sizes are not the same size as mentioned earlier. Therefore:

$$\frac{w_{g_i}}{h_{g_i}} = k_i \tag{4.2}$$

where w_{g_i} and h_{g_i} are grid-cell width and height of *Common MBR* respectively and k_i is width-to-height ratio. As a result, number of cells in $CMBR_i$ can be derived as:

$$N_{g_i} = Function(w_{g_i}) = \frac{k_i w_i h_i}{w_{g_i}^2} \tag{4.3}$$

where N_{g_i} is a function of w_{g_i}.

Lemma 4.2.2. *Given a uniform grid with w_g and h_g width and height cell sizes, respectively, and polygon P with the average length of edges in P, $\bar{E}_P$, the average number of grid-cells that each edge crosses is equal to*

$$N_{g/e} = 1 + \frac{2 \cdot \bar{E}_P \cdot (w_g + h_g)}{\pi \cdot w_g \cdot h_g} \tag{4.4}$$

Proof. Uniform grid can be seen as two sets of equally spaced parallel lines with w_g and h_g distances apart in x and y dimensions respectively. By following Buffon's Needle formulation, the average number of lines that each edge crosses would be $\frac{2\cdot\bar{E}_P}{\pi\cdot w_g}$ and $\frac{2\cdot\bar{E}_P}{\pi\cdot h_i}$ that divide x and y dimensions into $1 + \frac{2\cdot\bar{E}_P}{\pi\cdot w_g}$ and $1 + \frac{2\cdot\bar{E}_P}{\pi\cdot h_i}$ partitions respectively. Adding up these values (minus one because of one cell is counted twice) we end up having $1 + \frac{2\cdot\bar{E}_P\cdot(w_g+h_g)}{\pi\cdot w_g\cdot h_g}$ as number of cells crossed by an edge in the average. □

For any given pair i, $(P_{i_1}, P_{i_2}) \in C$, after applying *CMF-Grid* , $CMBR_i$ can be seen as a uniform grid over $\hat{E}_{i_1}$ and $\hat{E}_{i_2}$ sets of edges. Following lemma provides a basis for calculating total workload of all grid-cells in a pair after applying *CMF-Grid*.

Lemma 4.2.3. *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:*

$$W_i^E = Function(w_{g_i}) = \frac{|\hat{E}_{i_1}|.|\hat{E}_{i_2}|}{\pi^2 k_i h_i w_i} \times \left[\pi^2 k_i^2 w_{g_i}^2 + 2\pi k_i(k_i+1)(\hat{\bar{E}}_{i_1} + \hat{\bar{E}}_{i_2})w_{g_i} + 4(k_i+1)^2 \hat{\bar{E}}_{i_1}\hat{\bar{E}}_{i_2}\right] \tag{4.5}$$

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

Proof. Using Lemma 4.2.2 and Euations 4.2 and 4.3 we have:

$$N_{i_1} = |\hat{E}_{i_1}| \cdot N_{g/e_{i_1}} \quad \text{total number of edge-cell of } P_{i_1} \tag{4.6}$$

$$N_{e_{i_1}/g} = \frac{N_{i_1}}{N_{g_i}} \quad \text{average number of edges per cell for } P_{i_1} \tag{4.7}$$

$$workload_i^E = N_{e_{i_1}/g} \cdot N_{e_{i_2}/g} \cdot N_{g_i} \tag{4.8}$$

In the Equations 4.2 through 4.8, all the parameters but w_{g_i} and h_{g_i} are constant for a given pair. By replacing h_{g_i} by w_{g_i} from Equation 4.3 and combining Equations 4.6 and 4.7 with 4.8, we have workload function based on w_{g_i} variable as shown in Equation 4.9.

$$workload_i^E(w_{g_i}) = \frac{|\hat{E}_{i_1}|.|\hat{E}_{i_2}|}{\pi^2 k_i h_i w_i} \times \left[\pi^2 k_i^2 w_{g_i}^2 + 2\pi k_i(k_i+1)(\hat{\hat{E}}_{i_1}+\hat{\hat{E}}_{i_2})w_{g_i} + 4(k_i+1)^2 \hat{\hat{E}}_{i_1}\hat{\hat{E}}_{i_2}\right] \quad (4.9)$$

where $0 < w_{g_i} \leq w_i$ □

As a result of Lemma 4.2.3, the total estimated in-cell all-to-all refinement phase workload after applying *CMF-Grid* over set C can be formulated as a function as follow:

$$W_{total}^E(C) = \sum_{i=1}^{|C|}(W_i^E(w_{g_i})) \quad (4.10)$$

Also total actual in-cell all-to-all *edge-intersection* refinement phase workload can be stated as:

$$W_{total}^A(C) = \sum_{i=1}^{|C|}\sum_{k=1}^{N_{g_i}}(|E_{i_1}^k| \cdot |E_{i_2}^k|) \quad (4.11)$$

where set C is set of MBR-overlapping pairs of polygons.

Finally, uniform grid can also be considered a *Common MBR* that covers the whole universe. Therefore, its estimated workload can be derived as a special case of Equation 4.10 where $|C| = 1$, $k_i = 1$, $\hat{E}_{i_1} = E_{layer1}$, $\hat{E}_{i_2} = E_{layer2}$.

4.3 Use Case: ST_intersect Spatial Join

In this section, we demonstrate the efficiency of *CMF-Grid* by applying this filter to upgrade one previous GPU-based spatial join system referred to as *GCMF*[51]. To the best of our knowledge, *GCMF* is the state of art for *ST_intersect* join operation over GPU. For any given pair of polygons (P_1, P_2), *ST_intersect* is true if and only if 1) one polygon lies inside another one, or 2) there exists a pair of edges $E_{P1}(i)$ and $E_{P2}(j)$ such that they intersect, or overlap. In other words, *ST_intersect* is true if and only if two polygons share any space.

In the following subsections, we explain the system architecture and point out the changes we needed to apply in *GCMF* to make it functional with *CMF-Grid*.

4.3.1 System Design

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 [51].

Building on our previous work in *GCMF*, the improved system is shown in Figure 4.1. This system has two upgraded components (Figure 4.1-green boxes) compared to *GCMF* as follows:

- We replace *CMF* filter with *CMF-Grid* technique.
- A new grid-based *edge-intersection* test component substitutes all-to-all *Edge-intersection* test used in *GCMF*.

We already explained details of *CMF-Grid* filter in Section 4.2. Similar to its predecessor (*CMF* [51]), *CMF-Grid* classifies polygon pairs in set C into three groups as follows:

1. *Within candidate set* ($\mathbf{W}$): set of all the polygon pairs $(i_1, i_2) \in \mathbf{C}$ such that $MBR_{i_1 \cap i_2}$ is either equal to MBR_{i_1} or MBR_{i_2}.
2. *Intersecting-edge candidate set* ($\mathbf{I}$): set of all polygon pairs $(i_1, i_2) \in \mathbf{C}$ such that $(i_1, i_2) \notin \mathbf{W}$ and also $k \exists$ such that $0 \leq k < N_{g_i}$ and $E^k_{i_1}$ and $E^k_{i_1}$ are non-empty.
3. *Disjoint set*: Polygon pairs $(i_1, i_2) \in \mathbf{C}$ that are neither in $\mathbf{W}$ nor in $\mathbf{I}$.

4.3.1.1 Grid-based edge-intersection

The grid-based *edge-intersection* algorithm takes edge-cell lists from *CMF-Grid* as input and performs an in-cell all-to-all intersection test. As shown in Algorithm 5, each thread-block is assigned to a pair of polygons. Cells belonging to each pair of polygons are evenly distributed among the threads within each block, and every thread processes its assigned cells (outer loop in line 3 of Algorithm 5) by performing an all-to-all test (two inner loops in lines 4-5 of Algorithm 5). Since *ST_intersect* predicate requires the existence of at least one intersecting/overlapping edge pair, as soon as a thread detects such an edge pair, it sends a block-termination signal to other threads within its block. This ends the whole block and makes its GPU resources available to other waiting blocks.

Algorithm 5 Grid-based Edge-intersection Test over GPU

```
    For each $GPU\ Block\ i,\ 0 \leq i < |I|$, in parallel
 1: procedure GRID-BASED-EI-TEST($pair\ i : (i_1, i_2),\ thread\ t$)
 2:     cellPT=$\left\lceil \frac{N_{g_i}}{|blockDim|} \right\rceil$
 3:     for $t \cdot cellPT \leq k < (t+1) \cdot cellPT$ do
 4:         for $0 \leq j < |E^k_{i_1}|$ do
 5:             for $0 \leq l < |E^k_{i_2}|$ do
 6:                 if $E^k_{i_1}(j)\ intersects\ E^k_{i_2}(l)$ then
 7:                     Add $(i_1, i_2)$ to output
 8:                     Send block-termination signal
 9:                 end if
10:             end for
11:         end for
12:     end for
13: end procedure
```

4.4 Performance Evaluation

4.4.1 Datasets

We use two real datasets (*Urban-Admin*, *Water-Block*) each including two sets of polygonal objects from [51] for evaluation of our algorithms. They are originally from GIS domains from http://www.naturalearthdata.com and http://resources.arcgis.com with various sizes and characteristics. The details of the datasets are provided in Table 6.2 and they are available online on our website at http://grid.cs.gsu.edu/ daghajarian1/Grid-CMF.html in shapefile and text formats.

Table (4.2) Two real datasets used in our experiments.

Label	Dataset	# of Polygons	File Size
Urban-Admin	*ne_10m_admin_states*	11,878	46 MB
	ne_10m_urban_areas	4,646	41 MB
Water-Block	*US_block_boundaries*	219,831	2.175 GB
	US_water_bodies	463,591	921 MB

4.4.2 Experimental Setup

We carried out all the GPU experiments on Bridges cluster located at Pittsburgh Supercomputing Center (PSC), one of the XSEDE (Extreme Science and Engineering Discovery Environment) resources supported by the National Science Foundation (NSF) cyberinfrastructure program [78]. We used a Regular Shared Memory node (RSM-GPU) with 128 GB of RAM memory equipped with two *NVIDIA Tesla P100*, the latest GPUs from *NVIDIA* that has 16 *GB* of the main memory and it provides $3,584$ *CUDA cores* operating at 1480 MHz base clock that provides 5.3 *TFLOPS* of double precision floating point calculations. For spatial join use case, we used results of PostgreSQL version 9.4 with PostGIS version 2.2 and *GEOS* library version 3.4.2 [11] presented in [51] conducted on a *2.6 GHz Intel Xeon E5-2660v3* processor, as a sequential baseline. However for GPU baseline, we ran *GCMF* experiments on the *P100* GPU to make a fair comparison.

4.4.3 Performance Analysis of CMF-Grid

In this section, first we explore the impact of grid-cell size and grid-cell shape of *CMF-Grid* on refinement phase workload and end-to-end system running time. Second, we compare its performance to standard uniform grid in terms of workload, and finally average case estimated workload results is presented.

4.4.3.1 CMF-grid properties: Cell shape and size

4.4.3.1.1 Cell-shape:

As explained in Equation 4.2, we considered grid-cells as rectangles with an arbitrary width-to-height ratio (k_i). To see the impact of this ratio on performance of *CMF-Grid*, we choose two values for k_i in this experiment as follows:

- Square grid-cells: This is the most used grid-cell shape in the literature. Almost all the grid-based algorithms in the literature have used square-shape grid-cells that translates into $k_i = 1$.
- *CMBR-proportioned* grid-cells: *Common MBRs* can have various rectangular shapes. In particular, they may be skewed in one dimension (Figure 4.2-b). To see the impact of *CMBR* on the performance of *CMF-Grid*, we make dimensions of grid-cells proportional to their *Common MBR* dimensions by choosing $k_i = \frac{w_i}{h_i}$.

Figure 4.4 shows workload of *CMF-Grid* method for two grid-cell shapes. *CMF-Grid* with square grid-cells generates slightly less workload than *CMBR-proportioned*, however, this difference is less than 10%. Therefore, we can infer that *CMBR* shape does not have a significant impact on workload. Figure 4.5 also shows that using either grid-cell shapes does not make a considerable difference in number of active grid-cells. A cell is *active* if there are edges from both layers in that cell. In other words, cell k of pair (i_1, i_2) is active if and only if $|E_{i_1}^k| \neq 0$ and $|E_{i_2}^k| \neq 0$. We refer to the edges belonging to an active-cell, *active edge-cells.* Note that we only need to consider active-cells for the refinement phase. In summary, both schemes generate almost the same number of active-cells, however, workload of square-shape grid-cells is marginally smaller.

4.4.3.1.2 Cell-size:

Another challenging design issue for grid-techniques is how to determine the grid-cell size to optimize the workload. For any given pair of polygons, *CMF-Grid* chooses grid-cell size (w_{g_i}) of corresponding *Common MBR* as a function of average length of edges of those polygons, ($\bar{E}_{i_1}$ and $\bar{E}_{i_2}$). w_{g_i} can be chosen as minimum, maximum or average of $\bar{E}_{i_1}$ and $\bar{E}_{i_2}$. Our experimental results over all datasets confirm that the best function is based on minimum of the two as $w_{g_i}(x) = min(\hat{\bar{E}}_{i_1}, \hat{\bar{E}}_{i_2}) \cdot x$ where x is grid-cell size factor. Figure 4.6

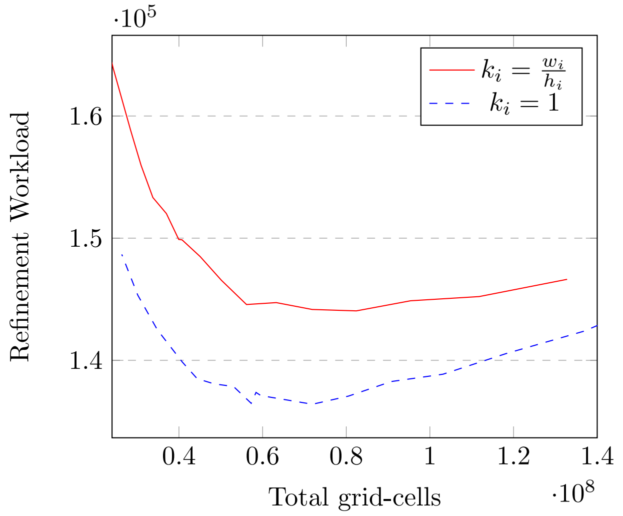


Figure (4.4) Impact of grid-cell shape on workload of edge-intersection refinement phase.

shows workload of *CMF-Grid* for wide range of this factor. As the plot illustrates, the minimum workload of refinement phase occurs around $x \approx 1$. In [40], $x \approx 1.33$ is reported that is consistence with our results.

Generally in sequential frameworks the smaller the workload the faster the processing time will be. However, in many-core parallel architectures this is not always true. Since these parallel frameworks have the ability to run thousands of threads, simultaneously exposing sufficient parallelism to the available resources and optimizing global and shared memory accesses are two of the most effective ways to achieve higher speedups. Furthermore, the smaller the workload the finer the grid; which means *CMF-Grid* requires more processing time. As a result, even though Figure 4.6 verifies that the minimum workload can be achieved by choosing $x \approx 1$, it does not necessarily mean it would lead to minimum running time. In fact to optimize the total running time, we need to make sure that the code exposes enough parallelism to the GPU (active grid-cells have enough workload to be processed in GPU thread-blocks) and *CMF-Grid* does not take much time applying too fine grid scheme. To see these impacts on running time, we show average workload per *active grid-cell* and total running time (time for *CMF-Grid* and refinement phase) versus grid-cell size factor in Figure 4.7. For fine grid schemes ($x < 2$), the running time is considerably high yet total workload is close to minimum in that range. Another plot in Figure 4.7 (red line, right y-axis)

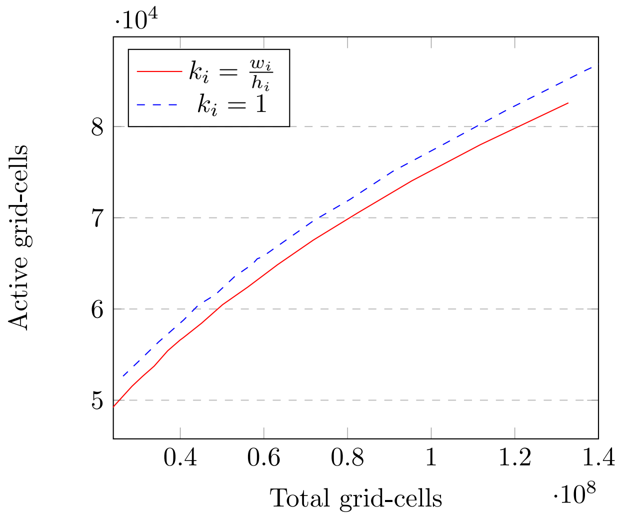


Figure (4.5) Impact of grid-cell shape on number of active grid-cells.

shows that workload per active grid-cell in range $x < 2$ is smaller than 5 edge-intersection tests per thread. This makes each grid-cell too lightly-loaded for GPU. Figure 4.7 shows that the best running time is achieved in grids with roughly 30 to 70 tests per active grid-cells ($3 < x < 6$). Figure 4.8 further breaks down the total running time into *CMF-Grid* and *edge-intersection* time. For fine grids (small cells), the overhead of applying *CMF-Grid* is too high. As we increase the grid-cell size this time decreases, however, *edge-intersection* running time increases. Therefore, on one hand, *CMF-Grid* performs faster with coarser grids, and on the other hand, *edge-intersection* test takes less time to process finer grids. Since the goal is to minimize the total time, as Figure 4.8 shows, we can see that this happens in the range $3 < x < 6$. In summary, in order to achieve highest speedup in *CMF-Grid*, we do not need to minimize the workload. Instead, by choosing a 3 to 6 times larger grid-cell size than the workload-optimized size, we can achieve the fastest running time. Finally, it is worth mentioning that the running time for two grid-cell shapes (square and CMBR-proportioned) did not show a significant difference (square scheme was 1% faster).

4.4.3.2 Average Case Analysis Figure 4.9 is provided to evaluate our average case analysis presented in Equation 4.10 versus the actual workload of *CMF-Grid* method shown in Equation 4.11. Although average case analysis does not yield the exact workload,

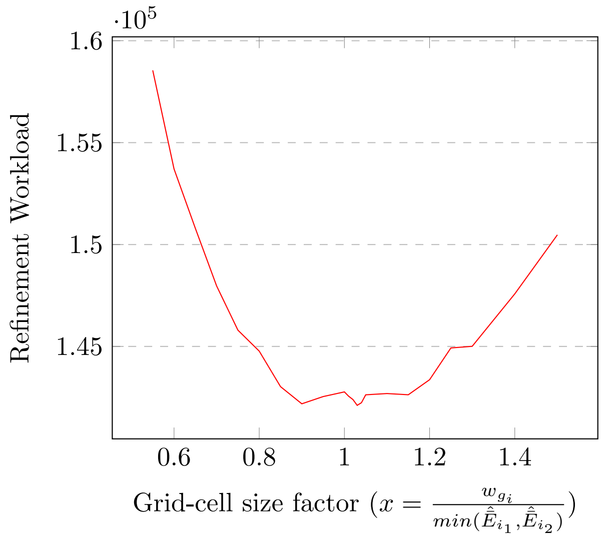


Figure (4.6) The impact of grid size factor on total workload. The minimum workload happens around $x \approx 1$ (grid-cell size: $w_{g_i} \approx min(\hat{\hat{E}}_{i_1}, \hat{\hat{E}}_{i_2})$).

Equation 4.10 experimentally provides an approximation of the expected workload. The more accurate estimation can be achieved by using a distribution model, such as Poisson, presenting the number of edges per cell. This is part of our future work.

4.4.3.3 CMF-grid vs. uniform grid Figure 4.10 shows the actual total workloads of *CMF-Grid* (Equation 4.11) and uniform grid for a wide range of grid-cell sizes (coarse to fine). The refinement workload in the case of *CMF-Grid* seems to be converging to the minimum value much faster than uniform grid. In other word, workload of uniform grid for the coarser grid-cells (smaller number of grid-cells) is much larger than *CMF-Grid*. However, as the number of grid-cells increases (finer grid) it converges toward *CMF-Grid*. Given that for a larger number of grid-cells more time is needed to process the whole grid and more memory is needed to keep its data structure, the efficiency of our method is significantly better than that of uniform grid. In fact, if we want to lower the workload of uniform grid down to optimal (converged point) we need to have almost three times more grid-cells (finer cells) than *CMF-Grid*.

We also provide Figure 4.11 to compare the number of active edge-cells of both methods. This plot shows that the number of active edge-cells does not change significantly for *CMF-*

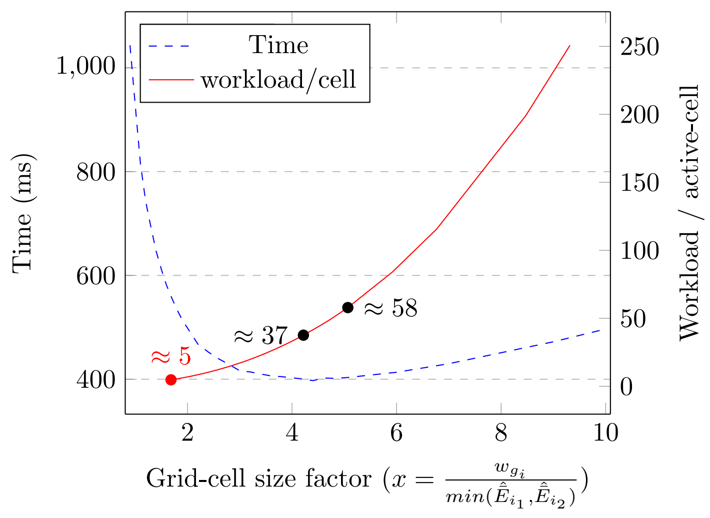


Figure (4.7) The impact of grid size (w_{g_i}) on total running time of CMF-grid and edge-intersection test. The minimum time happens in the range $4 \leq x \leq 6$.

Grid while it does increase for uniform grid as the grid becomes finer (more grid-cells). Even for a finer grid *CMF-Grid* has fewer active edge-cells than uniform grid. As a result, the refinement phase in *CMF-Grid* requires less processing time.

Figure 4.12 plots number of active grid-cells versus total number of grid-cells in *CMF-Grid* and uniform grid. Total number of active grid-cells in our method is up to 8 times greater than uniform grid for a fine grid scheme. The larger the number of grid-cells the greater this difference is. However, given that *CMF-Grid* does not need fine grids to optimize the running time, this difference is practically less than 2 times. It actually happens because of overlapping cells for different pairs in *CMF-Grid*.

4.4.4 CMF vs. CMF-grid

To compare *CMF-Grid* with *CMF* in terms of workload reduction, we define *edge-reduction* factor in *CMF-Grid* for a pair $i \in C$ similar to the one in [51] for *CMF* as:

$$R_{E_{grid-CMF}} = \frac{\sum_{i:(i_1,i_2)\in \mathbf{C}} |E_{i_1}| + |E_{i_2}|}{\sum_{i:(i_1,i_2)\in \mathbf{C}} \sum_{k=1}^{N_{g_i}} |\hat{E}_{i_1}^k| + |\hat{E}_{i_2}^k|} \tag{4.12}$$

For $N_{g_i} = 1$ (one grid-cell covering the whole $CMBR_i$) *CMF-Grid* and *CMF* are technically equivalent. Table 4.3 shows timing and workload of *edge-intersection* test for

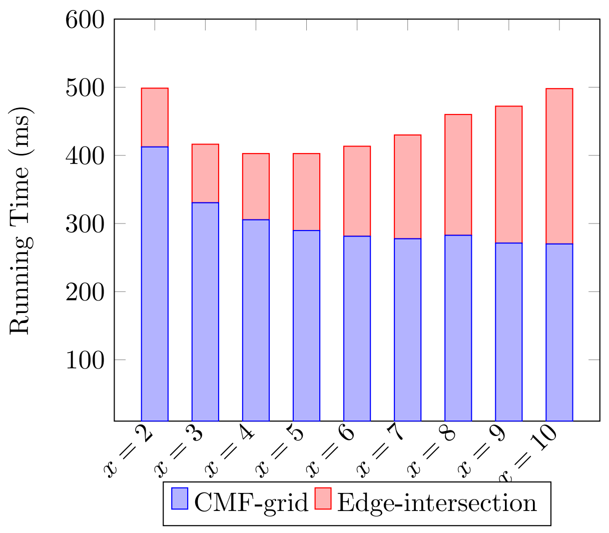


Figure (4.8) Running time of CMF-grid and Ednge-intersection for different gird-cell factors (x).

naive all-to-all, *CMF*, and *CMF-Grid* algorithms for Water-Block dataset. Both *CMF* and *CMF-Grid* eliminate almost two-third of pairs. *CMF edge reduction* factor is more than 43 times while *CMF-Grid edge* reduction factor is about 13 (third row in Table 4.3). Also $\bar{N}_{g/e}$ is equal to 3.17 which implies that each edge passes through almost three grid-cells on average. On the other hand, number of active edge-cells in *CMF-Grid* is less than that of edges in *CMF* by a factor of 3. As a result, total workload of *edge-intersection* test after applying *CMF-Grid* is more than 30,000 times smaller than all-to-all and also more than 730 times smaller than *CMF* filter. This tremendous decrease in workload leads to more than 10-fold speedup of *grid-based edge-intersection* test versus the one used in *GCMF* that brings down $\frac{Edge-intersection\ time}{Total\ time}$ ratio from 58% in *GCMF* to 12% in new system. It also achieves more than 800-fold speedup versus sequential *all-to-all* test.

Finally, Table 4.4 shows running time for two sequential PostGIS and GEOS optimized library systems reported in [51]. We also provide running time for our new system based on *CMF-Grid* and the current state of the art, GCMF. New end-to-end running time of GCMF is more than 4 times faster than its original results reported in [51] that is because of *P100 GPU* used in this experiment. The end-to-end running time for our new system with *CMF-Grid* shows a significantly impressive improvement of 225% compared to GCMF. It

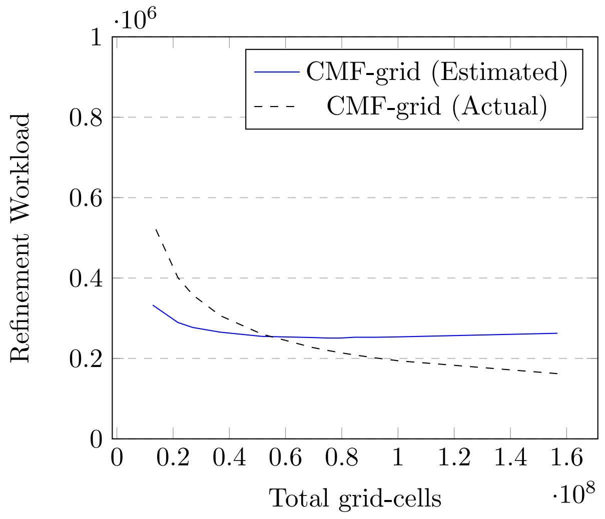


Figure (4.9) Estimated workload of CMF-grid versus actual workload of CMF-grid.

also shows up to 200-fold speedup gain compared to the best sequential system.

Table (4.3) Refinement workload reduction of Water-Block dataset for all-to-all, *CMF* and *CMF-Grid*.

	All-to-all	*CMF* [51]	***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**

4.5 Conclusion

In this chapter, we have introduced *CMF-Grid*, a new non-uniform grid technique that is designed with consideration of many-core GPU architectures. We provided comprehensive experimental results based on real datasets to analyze the impact of grid-cell properties such as grid-cell size and grid-cell shape on workload and running time. We showed that to achieve the highest running time we need a grid scheme with 4 to 6 times larger grid-cells than a workload-optimized scheme. Finally, we designed a system for spatial join with ST_intersect operation as a case study that improved the end-to-end running time of the current state

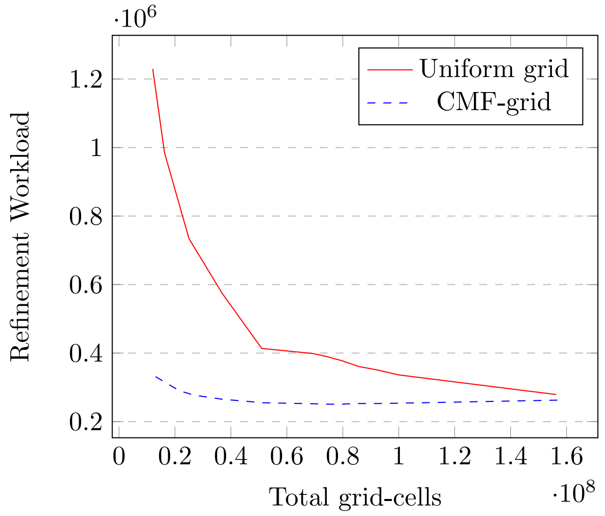


Figure (4.10) Workload of CMF-Grid versus uniform grid for a wide range of grid-cells.

Table (4.4) End-to-end running time of spatial join with ST_intersect operation for four different systems.

Dataset	Running Time (ms)				CMF-grid Speedup	
	Sequential		Parallel		Best Sequential	Parallel ($\frac{GCMF}{CMF-grid}$)
	PostGIS	GEOS	GCMF [51]	**CMF-grid**		
Urban-Admin	3,120	5,770	52	31	101 (PostGIS)	1.68
Water-Block	232,122	148,040	1,663	739	200 (GEOS)	2.25

of the art GPU-based system (GCMF) by 225%. The system was upgraded in two ways, *CMF-Grid* component was able to reduce the edges for refinement phase by a factor of 800 compared to GCMF and a new grid-based edge-intersection algorithm was able to achieve more than 10-fold speedup compared to the one in GCMF.

Our future work plan is to scale the system up by exploiting MPI to distribute the workload across nodes making it possible to process much larger datasets among the distributed GPU nodes.

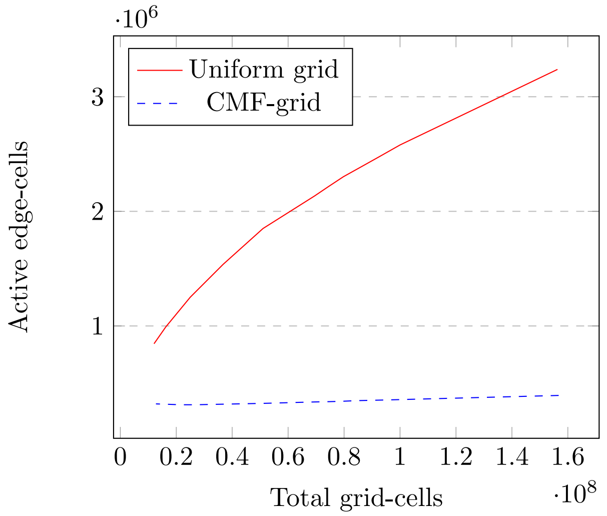


Figure (4.11) Number of active edge-cells of CMF-grid versus uniform grid.

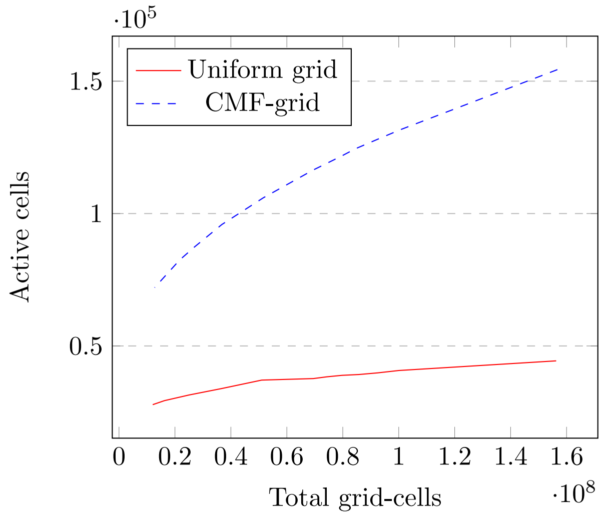


Figure (4.12) Number of active cells of CMF-Grid versus uniform grid for a wide range of grid-cells.

Chapter 5

GPU-JOIN+: AN OPTIMIZED GRID-BASED SPATIAL JOIN ALGORITHM FOR COLOCATION MINING OF BIG SPATIAL EVENT DATA OVER GPU

5.1 Introduction

This chapter provides an example of the applications of spatial join in data mining, called mining colocation patterns from big spatial event data (e.g., up to millions of spatial instances). Given a set of spatial instances with their feature types and locations, the colocation mining problem [79] aims to identify subsets of feature types whose instances are frequently located in close spatial proximity. For example, in ecology, species with symbiotic relationships tend to live close with each other in geographic space (e.g., Nile Crocodiles and Egyptian Plover). In public health, diseases (e.g., lung cancer) may co-occur with certain environmental factors (e.g., air pollution).

Colocation mining is important in many applications that aim to find associations between different spatial event types and environmental factors. For example, in public safety, law enforcement agencies are interested in finding relationships between different crime event types and potential crime generators [80]. In ecology, scientists analyze common spatial footprints of various species to capture their interactions and spatial distributions [81, 82]. In public health, identifying colocation patterns can help study disease transmission and environmental factors can help reveal the causes [83, 84]. In climate science, colocation patterns help reveal relationships between the occurrence of different climate extreme events. In location based services, colocation patterns help identify travelers that share the same favorite locations to promote effective tour recommendation [85].

Mining colocation patterns from big spatial event data poses several computational challenges. First, the number of candidate colocation patterns are exponential to the number of spatial features. Evaluating a large number of candidate patterns can be computationally prohibitive. Second, the number of pattern instances can be enormous when the number of event instances is large and instances are clumpy (i.e., there are many instances within the same spatial neighborhoods). For example, given 100,000 event instances and assume that the clumpiness is 10 (i.e., every event instance has 10 neighboring instances of the same event type), the number of colocation pattern instances of cardinality 5 (i.e., there are five event types in the pattern) can be up to $100,000 \times 10^5 = 10^{10}$. This not only makes pattern instance generation a computationally challenging task, but also creates a memory bottleneck.

Colocation pattern mining has been studied extensively in the literature. Most of existing colocation mining algorithms are sequential, including early work on spatial association rule mining [86, 87] and colocation patterns based on event-centric model [79].

Recently [88] proposed parallel grid-based colocation mining algorithms on GPUs. The algorithms include a novel cell-aggregate-based upper bound filter and two parallel implementation of refinement algorithms. Proposed cell-aggregate-based filter computes upper bounds of the interest measure of colocation patterns based on aggregated counts of event instances in neighborhood cells. The upper bound can be computed in parallel on GPU without generating pattern instances, and is also insensitive to pattern *clumpiness* (the average number of overlaying colocation instances for a given colocation instance) compared with the existing multi-resolution filter. The main goal of this chapter is to focus only on issues related to computational efficiency of parallel colocation mining algorithms on GPUs as an example spatial join. This chapter extends this recent work [88] with following contributions:

- **GPU-grid-join+**: we carried out a GPU profiling over their code and foud out that the problem is memory bound for GPU architecture. Then, based on detected bottlenecks, we proposed GPU optimizations including reducing unnecessary memory transfer and using batch memory transfer to replace frequent small memory transfers. We also redesigned preprocessing and filter kernels.

- We conducted experimental evaluations. Results on real data show that our optimized GPU algorithm achieves 4 to 12-fold speedup over our conference version (GPU-grid-join) on Nvidia P100 GPU, as well as 56 to 126-fold speedup over OpenMP implementation on Intel(R) Xeon(R) CPU with 12 cores. On synthetic data, the speedup is in ranges of 3 to 7-fold and 9 to 42-fold respectively.

The rest of this chapter is organized as follows:

5.2 Problem Statement

5.2.1 Basic Concepts

This subsection reviews some basic concepts based on which the colocation mining problem can be defined. More details on the concepts are in [79].

Spatial feature and instances: A *spatial feature* is a categorical attribute such as a crime event type (e.g., assault, drunk driving). For each spatial feature, there can be multiple *feature instances* at the same or different point locations (e.g., multiple instances of the same crime type "assault"). In the example of Figure 5.1(a), there are three spatial features (A, B and C). For spatial feature A, there are three instances (A_1, A_2, and A_3). Two feature instances are *spatial neighbors* if their spatial distance is smaller than a threshold. Two or more instances form a *clique* if every pair of instances are spatial neighbors.

Spatial colocation pattern: If the set of instances in a clique are from different feature types, then this set of instances is called a *colocation (pattern) instance*, and the corresponding set of features is a *colocation pattern.* The *cardinality* or *size* of a colocation pattern is the number of features involved. For example, in Figure 5.1(a), (A_1, B_1, C_1) is an instance of colocation pattern (A, B, C) with a size or cardinality of 3. If we put all the instances of a colocation pattern as different rows of a table, the table is called an *instance table.* For example, in Figure 5.1(b), the instance table of colocation pattern (A, B) has three row instances, as shown in the third table of the bottom panel. A spatial colocation pattern is *prevalent* (significant) if its feature instances are *frequently* located within the same neighborhood cliques. In order to quantify the prevalence or frequency, an interestingness measure called participation index has been proposed [79].

The *participation ratio* of a spatial feature within a candidate colocation pattern is the ratio of the number of unique feature instances that participate in colocation instances to the total number of feature instances. For example, in Figure 5.1, the participation ratio of B in candidate colocation pattern (A, B) is $\frac{2}{3}$ since only B_1 and B_2 participate into colocation instances $((A_1, B_1), (A_3, B_2))$. The *participation index* (PI) of a candidate colocation pattern is the minimum of participation ratios among all member features. For example, the participation index of the candidate colocation pattern (A, B) in Figure 5.1 is the minimum of $\frac{3}{3}$ and $\frac{2}{3}$, and is thus $\frac{2}{3}$. We use "candidate colocation patterns" to refer to those whose participation index values are undecided.

5.2.2 Problem Definition

We now introduce the formal definition of colocation mining problem [79].

Given:

- A set of spatial features and their instances
- Spatial neighborhood distance threshold
- Minimum threshold of participation index: θ

Find:

- All colocation patterns whose participation index are above or equal to θ

Objective:

- Minimize computational time cost

Constraint:

- Spatial neighborhood relationships are defined in Euclidean space

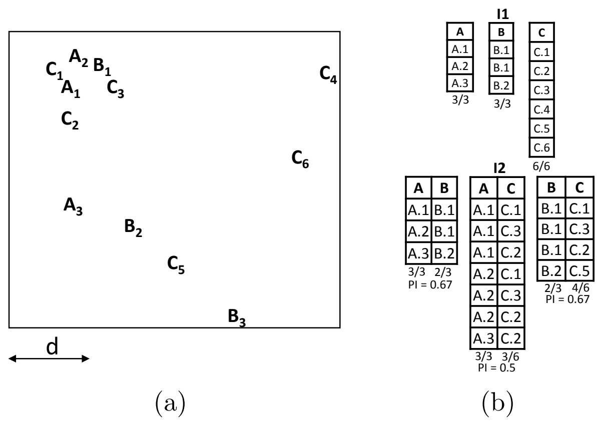

C_1 A_2 B_1 A_1 C_3 C_4 C_2 C_6 A_3 B_2 C_5 B_3
d
(a)
I1
A: A.1, A.2, A.3 — 3/3
B: B.1, B.1, B.2 — 3/3
C: C.1, C.2, C.3, C.4, C.5, C.6 — 6/6
I2
A B: A.1 B.1, A.2 B.1, A.3 B.2 — 3/3 2/3 PI = 0.67
A C: A.1 C.1, A.1 C.3, A.1 C.2, A.2 C.1, A.2 C.3, A.2 C.2, A.3 C.2 — 3/3 3/6 PI = 0.5
B C: B.1 C.1, B.1 C.3, B.1 C.2, B.2 C.5 — 2/3 4/6 PI = 0.67
(b)

Figure (5.1) A problem example with inputs and outputs. (a) Input spatial features and instances; (b) Candidate and prevalent colocation patterns, instance tables

5.3 Algorithm Overview

There are several challenges in the design and implementation of GPU parallel colocation mining algorithms. First, the number of row instances of a candidate colocation pattern can grow significantly with pattern cardinality when instances are dense, creating both intensive computational load and memory bottleneck. Second, memory coalesce is also a challenge due to the fact that the number of row instances produced by each kernel thread cannot be predetermined, and thus it is non-trivial to map a kernel thread index to the corresponding row numbers of the output instance table in GPU global memory.

Recently proposed GPU implementations [88] did not address most of these challenges. The algorithm generates and evaluates candidate colocation patterns by iterations over pattern cardinality (similar to sequential the algorithm in [79]). Specifically, the instance table of a size $k+1$ pattern is generated by join operations over instance tables of size k patterns. Their implementation uses both CPU and GPU (not in parallel), and it consists of three major components: **preprocessing**, **filtering**, and **refinement**.

1. *preprocessing* phase: it is done in CPU sequentially. It generates size 2 candidate patterns, creates a grid index over instances (cell to instance index), and counts the number of instances under each feature type in each cell ($CountMap$). After the

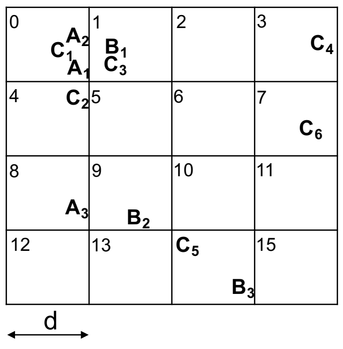
computation, the grid index and $CountMap$ are transfered from host memory to device memory.

2. *filtering* phase: it is done in GPU mostly. For each candidate pattern c with a cardinality of $k+1$ (in the $k+1$th iteration), the filter kernels (ParallelCellAggregateFilter) will be launched, before which the candidate pattern c is transferred from host to device. After kernel execution, an upper bound of participation index (PI) of c will be returned by the kernel from device to host.

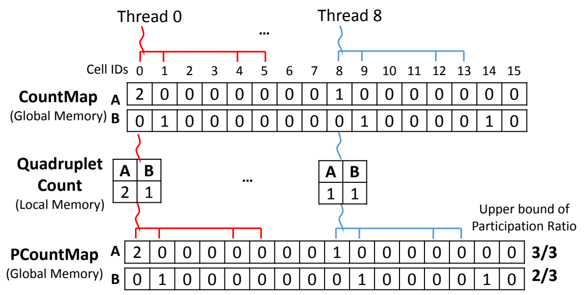
3. *refinement* phase: it is executed only if the upper bound of PI of candidate pattern c is above the minimum threshold. It generates the instance table of c and computes the exact PI through GPU refinement kernels (either grid-join based or prefix-join based). Since each kernel thread is used to generate a number (unknown before execution) of rows in the output instance table of c, there is a memory coalesce issue between threads. We address the issue with two rounds execution and a slot count array for kernel threads.

5.3.1 Cell-Aggregate-Based Upper Bound Filter

To introduce proposed cell aggregate based filter, we define a key concept of **quadruplet**. A quadruplet of a cell is a set of four cells, including the cell itself as well as its neighbors on the right, bottom, and right bottom. For a cell that is located on the right and bottom boundary of the grid, not all four cells exist and its quadruplet is defined empty (these cells will still be covered by other quadruplets). For example, in Figure 5.2, the quadruplet of cell 0 includes cells $(0, 1, 4, 5)$, while the quadruplet of cell 15 is an empty set. For each quadruplet, our filter computes the aggregated count of instances for every feature in the candidate pattern. If the aggregated count for any feature is zero, then there cannot exist colocation instances in the quadruplet. Otherwise, we pretend that all these feature instances participate into colocation pattern instances. This tends to overestimate the participating instances of a colocation pattern (an "upper bound"), but avoids expensive spatial



(a)


Thread 0
Thread 8
Cell IDs 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
CountMap (Global Memory)
Quadruplet Count (Local Memory)
Upper bound of Participation Ratio
PCountMap (Global Memory)
3/3
2/3

(b)

Figure (5.2) Grid-aggregate based Upper Bound Filter: (a) A regular grid (b) An execution trace of upper bound filter

join operations. Compared with the existing multi-resolution filter [79], which computes the upper bound based on generating coarse scale instance tables, the cell aggregate based filter has two advantages: first, it is embarrassingly parallel and can leverage the large number of GPU cores; second, its performance does not rely on the assumption that pattern instances are clumpy into a small number of cells, which is required by the existing multi-resolution filter.

5.4 GPU Optimization

GPU Profiling and Bottleneck Analysis: After profiling our recent GPU implementations [88], we found that the problem is a memory-bound with a significant amount of memory allocation and transfer. The bottleneck is due to the limited amount of available device memory, as well as the significant growth of the size of instance tables with pattern cardinality when event instances are spatially dense. The memory transfer between host and device in our conference GPU implementations is summarized in Table 5.1. Based on the identified memory bottleneck, we further optimize our recent GPU implementation with refined memory management, including reducing unnecessary memory transfer between host and device, replacing a large number of small memory transfer into a small number of batch memory transfer, and reducing excessive use of pinned memory. We also design a new kernel function for the preprocessing, and redesign the filtering kernel (e.g., utilizing GPU shared memory, reducing kernel calls). GPU optimization procedures in different phases are summarized in Table 5.2.

Table (5.1) Memory Data Structures Transfered Between Host and Device

Phases	**Host to Device**	**Device to Host**
Preprocessing	* Cell to Instance Index * CountMap	N/A
Filter	* Candidate Pattern c	Upper Bound of PI
Refine	**Round R1**: * Candidate Pattern c (size k+1) * Instance Table of size k **Round R2**: *Slot Counts	**Round R1**: * Participation Index (PI) of c * Slot Counts **Round R2**: *Instance Table of c

Table (5.2) Summary of GPU Optimization in different algorithm phases

	Reducing mem. trans. (1)	Batch mem. trans./alloc.(2)	Replacing pinned mem.(3)	New kernel function(4)	Redesigning kernel functions(5)
Preprocessing	✔		✔	✔	
Filter		✔	✔		✔
Refine		✔	✔		

1. Reducing unnecessary host-to-device and device-to-host memory transfers: In the preprocessing phase, our recent implementation conducts preprocessing in CPU and transfer results (CountMap and Cell to Instance Index) from host to device. Preprocessing may take up a significant ratio of total time cost. Thus, we design a new kernel function for preprocessing (details are in item **4** below), so results were directly generated in GPU memory without the need of memory transfer.

2. Replacing frequent small memory transfers with batch memory transfers or allocation: In the filtering phase, our recent implementation launches the filtering kernels for each candidate pattern c in the iteration. We need to transfer the candidate pattern c from host to device before kernel launch, and transfer upper bound of PI back from device to host after kernel launch. Since the number of candidate patterns in each iteration (cardinality) is combinatorial, there are frequent small memory transfers. To address this, we redesign the filtering kernel (details are in item **5** below) so that upper bounds of all candidate patterns in each iteration are computed all together in only two kernel launches. Thus, the candidate patterns and their upper bounds of PI are transferred all together once (batch mode) between host and device. In the refinement phase, our recent implementation allocates a fix amount of global memory for slot count arrays for each candidate pattern. We reduce that by allocating a bigger global memory buffer only once, and manage the buffer to write slot count arrays for different candidate patterns contiguously. This uses more memory but reduce the number of memory allocations.

3. Reducing the use of pinned memory: In all phases, our recently GPU implementation used pinned memory on the host. We used pinned memory to reduce the cost of memory transfer. However, since pinned memory is shared between operating system and applications and it is a limited resource. Therefore, excessive allocation of this memory (e.g., the big instance tables up to several gigabytes) may lead to degrading the performance of the system. We optimized pinned memory usages by replacing unnecessary cudaMallocHost allocations with system malloc API.

4. Adding new kernel function for preprocessing: The grid processing is an embarrassingly parallel problem and if it is efficiently implemented over GPU, it can be achieved at least an order of magnitude speedup over the CPU implementation. In our recent GPU versions, the preprocessing phase of applying uniform grid over data is handled over CPU. Although the preprocessing time is not large, by designing a kernel to do the computation over GPU we are able to gain significant speedup for this part over our recent versions. Each thread handles one feature instance to calculate which cell that instance belongs to. Therefore, total number of threads is equal to total number of feature instances. Since the input vector (x and y coordinate of feature instances) is linear, we use a linear kernel configuration to provide coalesce memory access in each block.

5. Redesigning filtering kernel function: As discussed earlier, the number of kernel function calls in the filtering phase of our recent implementations is combinatorial to the number of features. There are two kernel launches for each candidate pattern (one launch to compute $PCountMap$, and another launch to compute the upper bound of PI). For example, if there are 13 features, there will be 156 and 572 kernel calls for filtering all the degree-2 and degree-3 patterns respectively. The number of kernel calls can potentially go as high as 3432 for these features. To address these issues, we redesigned the GPU-based filter algorithm such that only two kernel calls are needed to filter out all the uninteresting patterns of the same cardinality (in the same iteration). In particular, we assign each three dimensional GPU block to count the number of feature instances of a given candidate pattern in one quadruplet with X-dimension size of block defined by the number of features and Y and Z-dimensions form a 2×2 strucure for processing each cell in quadruplet. Then, by using GPU shared memory, the first kernel aggregates the results within each GPU block (quadruplet) and the second kernel simply eliminates all the patterns that are below the given threshold. In the first kernel, each thread counts the number of instances of one feature within each GPU block. Then, each GPU block determines if all the features presented in the pattern have non-zero counts in the quadruplet using shared memory signaling. Therefore, the kernel configuration is a function of grid structure, the number of features and the number of candidate patterns for any given degree. In the second kernel, the algorithm uses a reduction tree for each candidate pattern in one 1-dimensional GPU block to find out if the minimum feature instance count passes the threshold. The block size of this kernel (only x-dimension) is a function of the number of features.

In the above, we discussed our further GPU optimization proposed in this paper based on refined memory management and kernel redesign. We acknowledge that there is still space of further improvement, such as redesigning refinement kernels, which could be done in future work.

5.5 Performance evaluation

5.5.1 Experimental setup

We performed GPU experiments on Bridges cluster located at Pittsburgh Supercomputing Center (PSC). We used one *NVIDIA Tesla P100* GPU on a GPU Shared Memory node. Nvidia $P100$ GPU has 16 *GB* of the main memory and it provides $3,584$ *CUDA cores* operating at 1480 MHz base clock that provides 5.3 *TFLOPS* of double precision floating point calculations. Furthermore, we ran CPU sequential and multi-core (OpenMP) experiments on Dell Precision Tower 7910 equipped with Intel(R) Xeon(R) CPU E5-2687w v4 @ 3.00GHz, 64GB main memory, and Ubuntu operating system. Algorithms were implemented in C++ and CUDA and compiled using g++ (without optimization flags) and nvcc compilers. For each experiment, we measure the average time cost of five runs. We compared the following candidate colocation algorithm implementations.

- **CPU Sequential:** the baseline presented in [79] (multi-resolution filter, grid-based instance table generation for size $k = 2$, and sort-merge based instance table generation for size $k > 2$).
- **CPU Multi-core (OpenMP):** OpenMP implementation of grid-join based refinement in [88] in multi-core CPU.
- **GPU-prefix-join** and **GPU-grid-join**: GPU algorithms presented in [88].
- **GPU-grid-join+:** Our optimized version of **GPU-grid-join** based on optimizations proposed in this chapter.

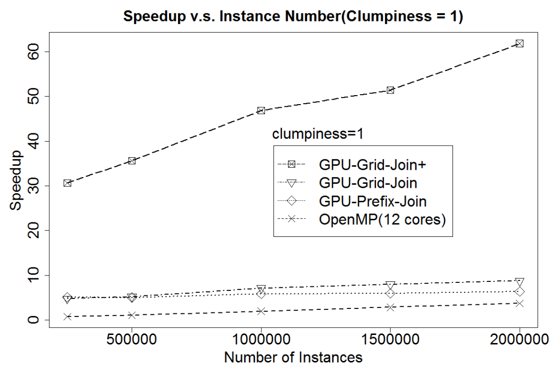
5.5.2 Dataset

We use the synthetic data generated similarly to [79]. We first chose a study area size of 10000×10000, a neighborhood distance threshold (also the size of a grid cell) of 10, a maximal pattern cardinality of 5, and the number of maximal colocation patterns as 2. The total number of features was 12 (5×2 plus 2 additional noise features). We then generated a *number of instances* for each maximal colocation pattern. Their locations were randomly distributed to different cells according to the **clumpiness** (i.e., the number of overlaying colocation instances within the same neighborhood, higher clumpiness means larger instance tables). In our experiments, we varied the number of instances and clumpiness to test sensitivity.

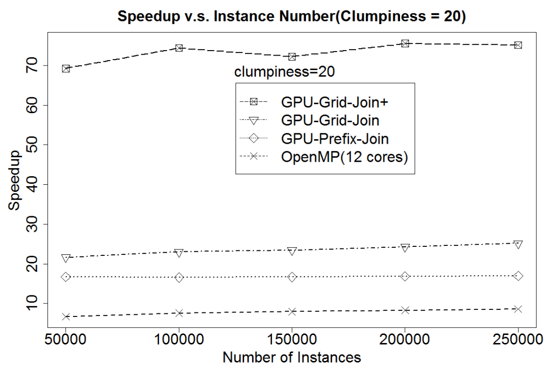
5.5.3 Results

5.5.3.1 Effect of the number of instances We conducted this experiment with two different parameter settings. For both settings, the minimum participation index threshold was 0.5. In the first setting, we set the clumpiness to 1 (very low clumpiness), and varied the number of feature instances as 250,000, 500,000, 1,000,000, 1,500,000 and 2,000,000. Results are summarized in Figure 5.3(a). With the number of event instances increasing, the speedups of all parallel methods increase due to increasing computational load. We find that our recent GPU implementations (GPU-Grid-Join and GPU-Prefix-Join) outperform OpenMP parallelization. Among all parallel methods, GPU-Grid-Join+ persistently outperforms the others (30 to 60 versus below 10), due to our proposed further GPU optimizations.

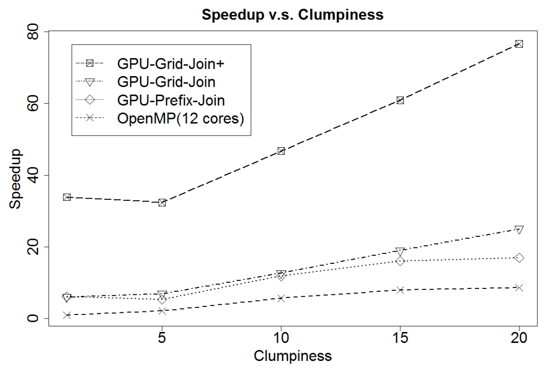
In the second setting, we set the clumpiness value as 20, and varied the number of feature instances as 50,000, 100,000, 150,000, 200,000, and 250,000. The number of feature instances were set smaller in this setting due to the fact that given the same number of feature instances, a higher clumpiness value results in far more colocation pattern instances but we only have limited memory. The results are summarized in Figure 5.3(b). With the number of instances increasing, the speedups of different parallel methods remain relatively stable. The reason is that on highly clumpy data, the main computational bottleneck is the significantly growing intermediate instance table size, instead of the number of event instances. GPU-Grid-Join+ method persistently outperforms the other methods (over 70 versus less than 30).



(a)



(b)



(c)

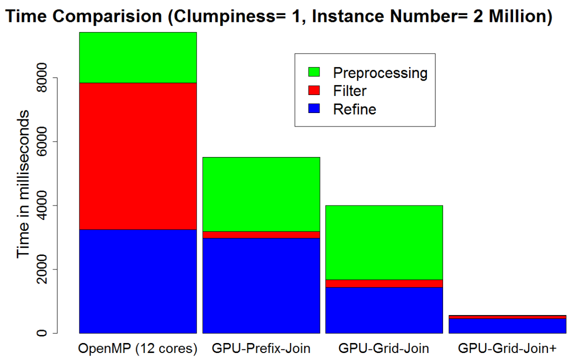
Figure (5.3) Comparison of parallel methods on synthetic datasets: (a) effect of the number of instances with clumpiness as 1 (b) effect of the number of instances with clumpiness as 20 (c) effect of clumpiness with the number of instances as $250,000$

5.5.3.2 Effect of Clumpiness

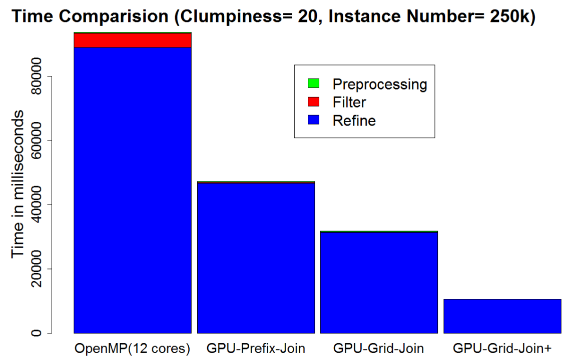
We set the number of instances to 250k, and the prevalence threshold to 0.5. We varied the clumpiness value as 1, 5, 10, 15, and 20. The results are summarized in Figure 5.3(c). With the clumpiness increasing, the speedups of all parallel method increase, due to the increasing computational load. The speedup of GPU-Grid-Join+ increases faster and is persistently higher than the others, primarily due to improved host and device memory management under increasing intermediate data sizes.

5.5.3.3 Comparison on Preprocessing, Filtering and Refinement Phases

We also compared the computational time costs of preprocessing, filtering and refinement phases of all parallel methods in the results of Figure 5.3(a-b). Results are shown in Fig-



(a)



(b)

Figure (5.4) Comparison of methods in their preprocessing, filter and refinement time costs on synthetic data: (a) when clumpiness is 1 and the number of instances is $2,000,000$ (b) when clumpiness is 20 and the number of instances is $250,000$

ure 5.4. Figure 5.4(a) shows results on input data with clumpiness as 1 and the number of instances as 2 million. In the results, our recent GPU implementations have slightly higher preprocessing cost than OpenMP (due to the extra memory transfer from host to device after preprocessing), but significantly lower costs in filtering (due to the utilization of GPU cores). The proposed method (GPU-Grid-Join+) significantly reduces the preprocessing cost due to parallelization on GPU, reducing memory transfer, as well as the refinement cost due to batch memory allocation and elimination of improper usage of pinned memory. Figure 5.4(b) shows the results on the data with clumpiness as 20 and the number of instances as $250,000$. In the results, all methods have tiny preprocessing and filtering costs compared with the refinement costs. The refinement costs in GPU implementation is significantly lower than the OpenMP method. GPU-Grid-Join+ shows the minimum refinement cost due to batch memory allocation and elimination of improper usage of pinned memory.

5.6 Conclusion and Future Work

This chapter provides an example of spatial join problem in pattern colocation mining. We studied recently proposed GPU-based cell-aggregate-based upper bound filter that less sensitive to data clumpiness compared with the existing multi-resolution filter. We profiled the code and diagnosed the bottleneck of current GPU implementation. Then, we proposed further GPU optimization based on identified memory bottleneck. Results on synthetic data on various parameter settings showed that proposed optimized version (GPU-join+) significantly improves the speedups.

The idea of *CMBR* filter can be applied in this spatial join problem as well. For $k > 2$, instead of applying refinement phase after each filter phase, for any two instances that already passed the filter in previous phase ($k = 2$), we calculate their *CMBR* and for $k = 3$ we consider filter operation applied to this *CMBR* and new layer instances. We can perform this filtering by recursively calculating *CMBR*s of previous iteration and apply it to the next phase.

Chapter 6

A PRIORI RESOURCE-AWARE LOAD BALANCING ALGORITHM FOR SPATIAL BIG DATA OVER HETEROGENEOUS SYSTEMS

Given two large datasets of spatial vector data representing spatial location of features and boundaries, medical images, traffic, electronic circuit etc, finding cross layer objects satisfying a join predicate such as edge intersect, overlay, point in polygon, etc., is called spatial join operation. While this class of spatial operations is significantly important in many applications including but not limited to Geographic Information System (GIS), Weather forecasting systems, Biology, VLSI and Social sciences [1, 2], building efficient processing systems for it is not a trivial task and any system needs to address some serious challenges. The ever increasing volume of these spatial data stored in text formats such as Well-Known Text (WKT) or in spatial databases such as PostgreSQL with PostGIS and Neo4j-spatial [89, 90] on the one hand and the crucial need to their real-time processing make it necessary to exploit full-fledged High Performance Computing (HPC) in such domains that efficiently addresses all the processing aspects from disk IO to filter and refinement handling. To address these challenges, researchers proposed several parallel and distributed solutions to make HPC computing available for spatial processing including cloud-based systems [5, 6], Message Passing Interface (MPI) systems [7], and map-reduce systems [8] most of which use powerful and relatively expensive computing clusters to break down the computations over several compute nodes. While these works have mainly focused on the multicore architecture system design to achieve acceptable speedups, even with such parallelism, employing only CPUs in modern heterogeneous architectures, typically equipped also with Graphic Processing Units (GPU), one to two orders of speedup remains unharnessed [9]. In particular, GPU computing has become more available by emerging powerful yet low-cost many-core architectures, e.g. Tesla, Pascal, Volta GPUs from NVIDIA. As a results, recently some embarrassingly parallel techniques suitable to these architectures are introduced [51, 52] to efficiently reduce search space and decrease computing time without using additional computing resources.

Maintaining spatial locality is another challenge encountered while processing spatial data. Reading the huge amount of data by sequential IO techniques is inefficient. Thus, it is common to apply parallel IO approaches such as MPI-IO for fetching the data off the disk. For instance, MPI-IO library uses a set of parallel processes, each reading a continuous chunk of a large file by performing some collective read operations [91]. All the read requests from different processes are sent to the parallel file system at once by the library. While using these techniques significantly reduces the IO time [91, 92], it has one limitation when it applies to spatial data domains; It cannot guarantee spatial locality in the chunks read by one process. This is because neighboring spatial objects are not necessarily stored closed to each other on the disk. Figure 6.1 illustrates the problem. As shown, after applying a uniform grid partitioning, each chunk read by each process is partially mapped to different grid cells, not necessarily neighboring cells, and objects belonging to a given cell may be distributed between different processes. Therefore, after performing IO, the processes must exchange data between to gather all the objects in a cell. Typically, this is performed after data partitioning. There are several approaches to partition data into spatially related clusters. Griding techniques such as uniform grid [40] is one class in which the space is partitioned into same-size cells and objects belong to each cell are categorized as one group. If one object belongs to more than one cell, it will appear in all the cells that makes data duplication is unavoidable. After data partitioning, the objects within a cell will be sent to the corresponding process. Regardless of the applied partitioning technique, there are some challenges that come from the nature of spatial data listed as follows:

- Cell sizes are arbitrary different and this rises load imbalance problem. This is because real data are usually non-uniform.

- Communication between processes to exchange data can take considerable time especially in clusters with low-speed network infrustructres.

- Processes may have different computing capacity and in order to minimize the total processing time, load balancing algorithm should take system heterogeneity into account.

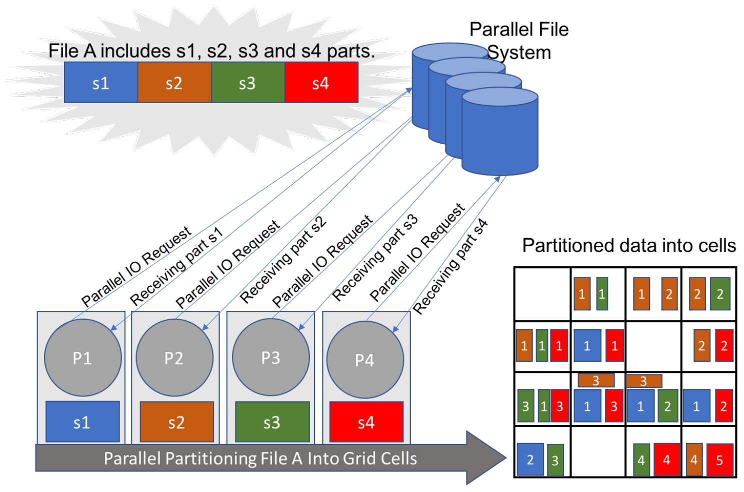


Figure (6.1) An example of distributed computing. Four processes (P_1, P_2, P_3 and P_4) read continuous parts of a big files (s_1, s_2, s_3 and s_4) using a parallel IO technique such as MPI-IO. Then, each process maps its data into a spatial grid and finally, the local grids merge into one global grid.

In this chapter, we address some of the above challenges by proposing an Integer Linear Programming (ILP) model that formulates load balancing and processing of spatial data and then, rigorously approximating the model by an intuitive heuristic load balancing algorithm. The contributions of this chapter are as follows:

- We formulate the spatial data load balancing and processing over distributed clusters equipped with heterogeneous compute nodes as a general ILP.

- We propose two resource-aware heuristic algorithms to approximate the ILP based on information available of compute nodes. 1) Proportional-cell algorithm (PC), and, 2) Proportional-load (PL) algorithm that assign cells to processes proportional to their compute power. We compare these algorithms with round-robin (RR) algorithm and show the efficiency of PL load-balancer in processing non-uniform datasets.

- Finally, we develop *MPI-cuda-GIS*, a distributed system that effectively integrates CPU and GPU to accelerate spatial computing based on MPI and CUDA programming languages. We also perform an extensive set of experiments over OpenStreetMap real datasets to demonstrate the efficiency of our system.

The remainder of this chapter is organized as follows. In the next section, we summarize load balancing techniques in the literature. Section 6.2 formulates the load balancing and processing of spatial data as an ILP. Then, we propose two heuristics in Section 6.3 to approximate the proposed ILP. The experiments, dataset and system setup are explained in Section 6.5. Finally, we conclude the chapter in the last section.

6.1 Related Work

The load balancing problem in different field of computer science including network and HPC has been long established. Researchers have tried to look at this problem from different angels from practical and experimental perspectives to theoretical aspects. In this section, we summarized some of the most significant theoretical load balancing works. We pointed out their pros and cons and their limitations compared to our proposed framework.

A general static load balancing model for job scheduling over a distributed computer network to minimize the mean response time of a job is proposed in [68]. They formulated the load balancing task as a nonlinear optimization problem with $n(n+1)/2$ variables where n is the number of nodes. Optimal solution is presented using a Lagrange multiplier approach. They also provided two efficient algorithms that determine optimal load for each host. parametric-study algorithm that generates optimal solution as a function of communication time and single-point algorithm that gives optimal solution for given system parameters including communication time. The framework has some limiting assumptions. First, they did not consider the problem of partitioning a big job into small task. In fact, they assumed all nodes have the same processing capabilities and a job may be processed completely at any node in the system. However, considering the big data era and jobs processing huge volume of data, this is an impractical assumption. Second, the model assumed the communication between different nodes is one way that is if node A transfers a job to node B, no job can be sent to node A from B. Third, as name implies, the framework is static and decision to transfer a job from one node to another does not depend on the state of the system.

[69] proposed a framework for load balancing iterative computations in clusters with heterogeneous computing nodes. The model assumes that application data is already partitioned between the processing nodes forming a virtual ring. In each iteration, the computation involves independent calculations carried out in parallel in each node followed by exchanging the partial results between neighboring nodes in the virtual ring. The goal is to select a subset of processing nodes from all nodes and balance the load between them such that it minimizes execution time while these nodes are not fully connected and pairs may share physical communication links due resource limitations. The model considered two scenarios, 1) SharedRing that there may exist several messages sharing a link, 2) SliceRing that dedicated links are used for communications. Some heuristic algorithms is provided in this work to solve these optimization problems. The main limitation of this model is that it is suitable only for application iterative computations with local partial results exchange in each step. In fact, it is limited to ring topologies. Moreover, the problem of big data IO and initial partitioning of data between nodes is not addressed.

Dynamic load balancing on message passing multiprocessor has been studied in [70] as diffusion schemes. They provided converging conditions as well as convergence rate for arbitrary topologies. Hypercube network analysis is provided as a case example and they showed that diffusion approach to load balancing on a hypercube topology of multiprocessors is inferior to dimension exchange method. This well-presented model has several limitations. First, they quantified work in terms of *tasks* and assume all *tasks* require an equal amount of computational time and nonuniform task partitioning of heterogeneous data is not addressed. Second, although the model is designed for any topology it did not consider spatially related tasks that means it is not suitable for load balancing applications that need to maintain locality.

In [71] and [72] two data migration-based load balancing models are provided. In [71] a load balancing framework called Ursa is proposed for large scale cloud storage services. It formulates an Integer Linear Programming (ILP) optimization problem that chooses a subsets of objects from heavy-loaded nodes called hot-spots and performs topology-aware migration to minimize latency and bandwidth. Ursa is designed to identify cost-optimal source-destination node pairs for dynamic and scalable load reconfiguration by applying 1) a workload-driven ILP optimization approach to eliminate hot-spot nodes while minimizing reconfiguration costs, and 2) a divide and conquer technique to break down expensive computations to provide scalability. While this work provides a practical framework for load balancing in cloud storage services, because it is designed at a storage layer it is not application-aware and does not consider data locality or other application-specific requirements for distributing the work load between nodes. Furthermore, it assumes that architecture is organized as a spanning tree topology that makes unsuitable for other network architectures. In [72] SWAT ,a load balancing algorithm, is proposed to address the problem of performance isolation of multi-tenant databases in cloud systems that caused by resource sharing among co-located tenants. Similar to [71], the general idea is to select tenant pairs for load-swap in a highly resource and time efficient manner. SWAT initially tries to eliminates all the hotspots and balance the load among all the nodes by *load leveling*. If it is not possible to balance the load, then, it eliminates the hotspots through *hotspot elimination* process. Finally, in case both load balancing and hotspot elimination fail, SWAT tries to minimize the overload rather than eliminating it by *hotspot migration*. Partitioning problem and maintaining locality while swapping workload have not been addressed in this work.

6.2 ILP Problem Formulation

In this section, first, we state the definitions and notations used for the rest of this paper. Then, we formulate our load balancing scheme as an ILP. Table 6.1 has summarized the notations.

Load balancing problem can be stated as follow. There are N distributed processes each

Table (6.1) List of primitive symbols.

Symbol	Description
N	The number of processes
K	The number of cell partitions of the whole area
s_i , $1 \leq i \leq N$	The size of data chunk initially read by process i
m_i , $1 \leq i \leq N$	The maximum memory capacity of process i for processing data
p_i , $1 \leq i \leq N$	Process i compute rate
w_i , $1 \leq i \leq N$	Assigned workload to process i after load balancing
l_j , $1 \leq j \leq K$	The workload of cell j
$a_{ij},\ 1 \leq j \leq K,\ \ 1 \leq i \leq N$	The partial size of $cell_j$ owned by process i
$b_{ij} \in \{0,1\},\ 1 \leq j \leq K,\ \ 1 \leq i \leq N$	Boolean variable determining if $cell_j$ is assigned to process i
$0 \leq c_{ij} \leq 1,\ 1 \leq j \leq K,\ \ 1 \leq i \leq N$	The network transfer bandwidth from process i to j

owned part (of size s_i) of a large file with a total file size of S. Let a_{ij} determines the size of $cell_j$ in process i. Therefore:

$$s_i = \sum_{j=1}^{K} a_{ij} \quad 1 \leq i \leq N \tag{6.1}$$

We also assume the data is divided into K partitions of different sizes (l_j) such that $K \gg N$ and $0 < l_j \leq S$. Typically, l_j is distributed over various processes such that:

$$l_j = \sum_{i=1}^{N} a_{ij} \quad 1 \leq j \leq K \tag{6.2}$$

For generality, we assume that the system is heterogeneous and each process may have a different compute capability. Let m_i be the maximum data size that process i can handle at rate p_i while it keeps all the data in its global memory. Also, let $P = \{i,\ 1 \leq i \leq N\}$ be set of processes and $L = \{l_j,\ 1 \leq i \leq K\}$ be the set of partitioned data(cells). Then, the load balancing algorithm is a surjection (onto) from L to P, $f : L \rightarrow P$, formulated as the following:

$$w_i = \sum_{j=1}^{K} b_{ij} l_j \quad , \; 1 \leq i \leq N \tag{6.3}$$

$$b_{ij} \in \{0, 1\} \tag{6.4}$$

$$\sum_{i=1}^{N} b_{ij} = 1 \qquad , 1 \leq j \leq K \tag{6.5}$$

$$w_i \leq m_i \tag{6.6}$$

where w_i is the workload assigned to process i by load balancing algorithm and b_{ij} is a binary decision variable determining if l_j has been assigned to process i. The Equation 6.6 is the constraint that makes sure that each cell is allocated to exactly one process.

The load balancing algorithm has apriori knowledge of partitionings (a_{ij} are known to the algorithm). We also assume that the compute and memory parameters of each process (p_i and m_i) are given (resource-aware algorithm). Then, the goal of load balancing algorithm is to find matrix $B = [b_{ij}]$ such that it distributes the workload among all the processes in a way that total processing time stated in Equation 6.8 is minimized.

The processing time of process i as shown in Equation 6.7 includes two components: 1) the time required to collect data of assigned cells from other processes, and 2) the time to process those cells.

$$t_i = p_i.w_i + \sum_{j=1}^{K} (b_{ij} (\sum_{l=1, l \neq i}^{N} c_{li}.a_{lj})) \tag{6.7}$$

where $C_{n \times n} = [c_{ij}]$ is the network bandwidth matrix and c_{ij} presents transfer rate from process i to j. For the most of networks C is symmetric. Later on, we will see that in modern clusters data exchange time is negligible compared to processing time. Finally, the running time of algorithm can be defined as:

$$T = \max_{1 \leq i \leq N} (t_i) \tag{6.8}$$

The objective function of the load balancing algorithm in our model is to find matrix $B = [b_{ij}]$ such that it minimizes T as follows.

$$Objective = \min_{B_{N \times K}} \max_{1 \leq i \leq N} (t_i) \tag{6.9}$$

Lemma 6.2.1. *Apriori Resource-Aware Load Balance formulation is a NP-hard problem.*

Proof. Since the problem is formulated as ILP, the proof is trivial. □

6.3 Approximation algorithms

Since our model is NP-hard and the number of variables are too much, we cannot solve the ILP by available softwares such as IBM CPLEX. Instead, we try to approximate the model heuristically. In the following subsections, we explain three heuristics.

6.3.1 Round robin load-balancer (RR)

In our model, cells are the smallest allocatable unit of workload; However, by applying different griding techniques or changing the hyperparameters such as cell sizes, we can control the size and the number of cells. This heuristic simply assigns non-empty (*active*) cells to processes in a round robin fashion as shown in Algorithm 6. A cell is *active* if and only if it contains at least one object from each layer. The algorithm counts the number of *active* cells using *nzCounter* variable and in line 7 it applies the modulus function to assign current *active* cell to the next process (round robin fashion).

While this linear heuristic is straightforward to implement it has some drawbacks as follows:

1. If the dataset is not uniformly distributed, the cell sizes could potentially vary in a

Algorithm 6 Round robin load-balancer

```
Input: $N$, $K$.
Output: $b_{ij}$, $0 \le i < N$ and $0 \le j < K$.

procedure ROUND ROBIN LOAD-BALANCER
    $nzCounter = 0$
    for each $cell_j$ $0 \le j < K$ do
        if $cell_j$ is not $active$ then
            continue
        end if
        $i = (nzCounter \bmod N)$
        $b_{ij} = 1$
        $nzCounter = nzCounter + 1$
    end for
end procedure
```

wide range. This makes the algorithm load-imbalance. Later on, we will see RR is not suitable for real datasets that are typically non-uniform.

2. Even if the data is uniform (cells are evenly-loaded), evenly distributing the load between different processes is inefficient in a heterogeneous cluster where nodes have different compute power.

3. RR is inefficient in managing the available memory and is the least scalable scheme. Because it does not consider the fact that the available memory is different in various processes and assigns the same number of cells to all processes. As a result, processes are bounded by the one whose memory is the least. Thus, the total workload is bounded by $N \times min(m_i)$, $0 \le i < N$

6.3.2 Heuristic 1: Proportional-Cell load-balancer (PC)

RR can be improved by considering the compute and memory limitations of processes when assigning the cells. The idea is still based on modified round robin. In each round, PC algorithm might assign more than one cell to the corresponding process, proportional to its compute capability while keeping track of available memory of the process. Algorithm 7 describes proportional round robin algorithm.

Algorithm 7 Resource-aware round robin load-balancer

```
Input: $N$, $K$, $m_i$, $p_i$: $0 \le i < N$, $l_j$: $0 \le j < K$.
Output: $b_{ij}$, $0 \le i < N$ and $0 \le j < K$.

 1: procedure RESOURCE-AWARE ROUND ROBIN LOAD-BALANCER
 2:     Set $activeCellList$
 3:     Set $availableProcessList$
 4:     while ( NOT $activeCellList.empty()$ ) do
 5:         $i$ = GetNextAvailableProcess()
 6:         $cellCounter = 0$
 7:         $roundCellLimit = L_1(s_i)$
 8:         while NOT $activeCellList.empty()$
 9:             AND $cellCounter$ ¡ $roundCellLimit$
10:             AND totalWorkload$_i \le memLimit_i$ do
11:             $j$ = GetNextActiveCell()
12:             $b_{ij} = 1$
13:             $cellCounter = cellCounter + 1$
14:             $totalWorkload_i = totalWorkload_i + l_j$
15:             $activeCellList().remove(cell_j)$
16:         end while
17:     end while
18: end procedure
```

Because of two nested while loops Algorithm 7 may seem quadratic time, but it is still linear. The loops iterate through all the *active* cells with a shared loop variable (j) and the conditions of two *while* loops is the same. The outer loop selects the next available process in a round robin fashion and then the inner loop allocates some cells to it proportional to its compute power while making sure the process does not reach to its memory limit. Therefore, the sum of the iterations over the two *while* loops is K. Extracting the next object from *activeCellList* and *availableProcessList* (*GetNextAvailableProcess()* and *GetNextActiveCell()* functions) or removal a processed object from them can have constant running time complexity using proper data structures such as doubly link list and keeping track of the current position.

Although, Algorithm 7 addresses issue 2 in Section 6.3.1, it still has scalability and load-imbalance problems with non-uniform datasets.

6.3.3 Heuristic 2: Proportional-Load load-balancer (PL)

PL is similar to PC algorithm while it is more scalable and performs a better load balancing over data with irregular distributions. The basic idea is to assign heavy-loaded cells to the processes with high compute capability, e.g. GPUs, and lightly-loaded cells to those with lower compute power, e.g. single core CPUs. As such, we assign a compute factor to each process such that it represents its compute capability relative to other processes. To do so, we can set the single core CPU compute power to 1 and use available speedup information, e.g. those that are partially provided in Chapter 3 and 4, to estimate the compute factor of other nodes. This calculation does not need to be exact and may be dynamically tuned by real dataset experiments. Given compute factor information and applying a threshold, PL algorithm splits the processes into two categories: 1) low-compute, and, 2) high-compute nodes. It also needs to keep track of the current assigned workload to each process. On the other hand, the algorithm must be able to estimate the workload of each cell. As used in Algorithm 7, the total number of objects in each cell is an appropriate indicator for the workload of that cell.

The Algorithm 8 describes PL load-balancer. The key data structure in this algorithm is based on min-heap data structure. There are some parallel implementations of min-heap or priority queue data structures[93, 94, 95, 96] over many-core architectures in the literature that can be efficiently exploited in this algorithm, however, the experimental results show good performance with our sequential implementation since the number of active cells is reasonably small.

First, Algorithm 8 sorts the cells by their workload ($O(K. \log K)$). It also builds a min-heap of *process* objects by their current scaled assigned workload as the key, initially set to zero ($N \log N$). To scale the workload of process i, the algorithm divides the workload by its compute factor. The intuition behind this scaling is that the higher the compute capability the more workload can be handled in a given fixed time. On the other hand, load balancer goal is to distribute the cells such that processing time of all processes is as close to each other as possible. Therefore, the algorithm pick the process whose current processing time

Algorithm 8 Resource-aware priority-based load-balancer

```
    Input: $N$, $K$, $m_i$, $p_i$: $0 \le i < N$, $l_j$: $0 \le j < K$.
    Output: $b_{ij}$, $0 \le i < N$ and $0 \le j < K$.

 1: procedure RESOURCE-AWARE PRIORITY-BASED LOAD-BALANCER
 2:     Sort $activeCellVector$ by workload
 3:     $p = 0$
 4:     $q = activeCellVector.size()$ - 1
 5:     Split processes into High and Low compute classes
 6:     Make Min-heap($processList$)
 7:     while ( $p \le q$ ) do
 8:         $proc$ = Min-heap.extract-min()
 9:         $i = proc.index$
10:         if $proc.isHighCompute()$ then
11:             $cell = activeCellVector[q]$
12:             $j = cell.index$
13:             $q = q$ - 1
14:         else
15:             $cell = activeCellVector[p]$
16:             $j = cell.index$
17:             $p = p + 1$
18:         end if
19:         $b_{ij} = 1$
20:         $proc.scaledWorkload \mathrel{+}= \frac{l_i}{proc.computeFactor}$
21:     end while
22: end procedure
```

is minimum. In other words, the scaled workload estimates the current processing time. After the extraction of the process with the smallest workload ($O(1)$), only if it has high compute power, PL heuristic allocates a heavy-loaded cell to prevent making bottleneck at low-compute nodes by assigning them large cells. Finally, at the end of *while* loop, the current assigned workload to the process is updated and the algorithm pushes it back to the min-heap ($O(\log N)$). The variables p and q keep track of small and large cells respectively and once they pass each other ($p > q$) all the cells are processed. Since the loop iterates through each cell once, K times in total and each min-heap update takes $O(\log N)$ then, the total loop complexity is $O(K \log N)$. As a result, the running time complexity of PL heuristic algorithm is $O(N \log N + K \log K + K \log N)$.

6.4 MPI-cuda-GIS system

In this section, we propose *MPI-cuda-GIS*, a HPC system based on our ILP model, that is designed based on MPI to integrate CPU and GPU computing power to gain as high speedup as possible. We use *GCMF* system presented in [51] for GPU computing. We also develope plane-sweep based libraries for CPU computing. The overall system workflow is shown in Figure 6.2. The following is the series of communications/operations in *MPI-cuda-GIS*.

1. **Parallel-IO**: First, each process calculates the size and the offset of data to read. The chunk size is almost the same across all the processes approximately equal to $fileSize/N$ where N is the number of processes. The chunks read by processes are overlapping to prevent the objects located in the boundaries of each chunk from scattering between processes. If an object belongs to several processes, eventually only one of them will own it the rest will not store the object. MPI-IO library (*MPI_File_read_at_all* function) is used for performing parallel IO. At the end of this step, the data will be parsed to be ready for later usage.

2. **Data partitioning**: Each process locally applies a uniform grid technique to its data. Process i calculates row i of matrix $A_{N\times K}$ where a_{ij} is size of data mapped to $cell_j$ and held by process i.

3. **Cell information gathering**: The master collects all the information of matrix A from other processes (one row from each) using *MPI_gather* operation.

4. **Load balancing**: Master performs load balancing using one of the algorithms presented in Section 6.3. The outputs of load balancing algorithm are process-cell mapping matrix $B_{N\times K}$ (b_{ij} is 1 if and only if $cell_j$ is mapped to process i) and vector $L_{1:K}$ (l_j represents the total workload of $cell_j$). Master broadcasts these information to all the processes using *MPI_Bcast* function.

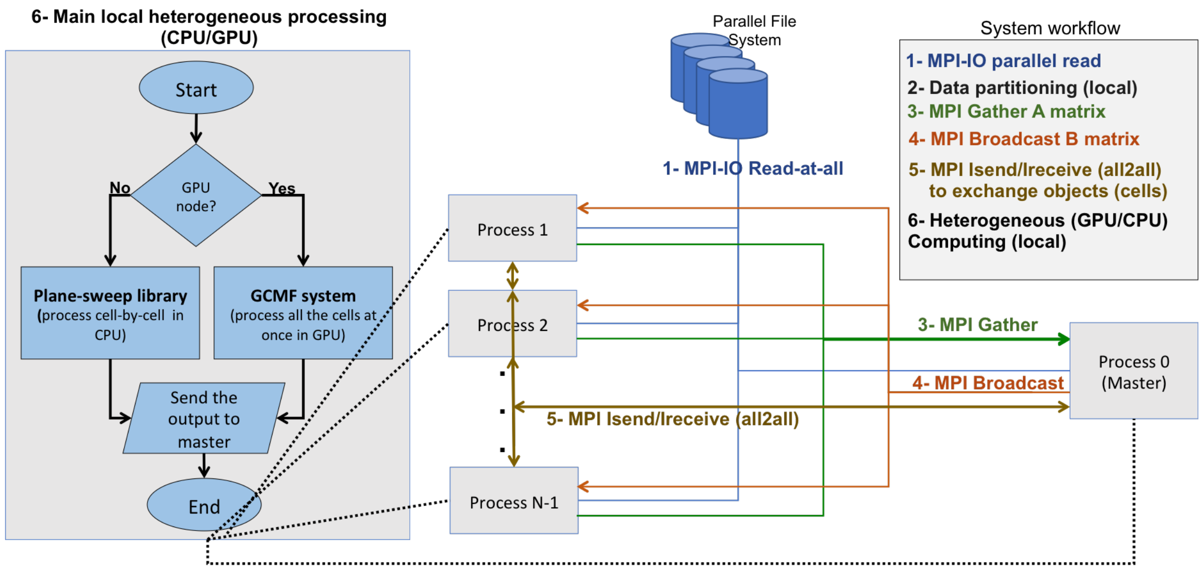

6- Main local heterogeneous processing (CPU/GPU)
Start
GPU node?
No
Yes
Plane-sweep library (process cell-by-cell in CPU)
GCMF system (process all the cells at once in GPU)
Send the output to master
End
Parallel File System
1- MPI-IO Read-at-all
Process 1
Process 2
Process N-1
Process 0 (Master)
3- MPI Gather
4- MPI Broadcast
5- MPI Isend/Ireceive (all2all)
System workflow
1- MPI-IO parallel read
2- Data partitioning (local)
3- MPI Gather A matrix
4- MPI Broadcast B matrix
5- MPI Isend/Ireceive (all2all) to exchange objects (cells)
6- Heterogeneous (GPU/CPU) Computing (local)

Figure (6.2) MPI-cuda-GIS system workflow.

5. **Data exchange**: After receiving matrix B, process i_1 sends $a_{i_1 j}$ to process i_2 if and only if $b_{i_2 j} == 1$. This phase potentially needs all-to-all communication. Since data sizes in each cell owned by each process is variant and to optimize the data exchange, we use asynchronous buffered MPI functions for communications in this phase. The protocol is as follows: first, using all-to-all *MPI_Allgather* processes exchange size of their data whose belong to other processes calculated from A and B matrices. Then, each process can preassign the separate buffers to receive data from others. Finally, the actual data are exchanged by asynchronous send and receive functions (*MPI_Isend* and *MPI_Irecv*).

6. **Data processing**: Each process depends on whether it is a GPU or CPU compute node handles its assigned cells differently using either *GCMF* GPU-based system or the plane-sweep-based CPU library.

6.5 Performance Evaluation

In this section, we describe the real datasets and cluster setup used in our experiments in the first two subsections. Then, we explain the results and comparison of proposed load balancing algorithms and MPI-cuda-GIS system.

6.5.1 Datasets

We use new version of OpenStreetMap real vector dataset provided by SpatialHadoop community [97] for our experiments. This dataset contains various files with different sizes each representing some features such as road networks, all the buildings, boundaries of sports areas, etc. for the whole world. Table 6.2 summarizes the layers used in our experiments.

Table (6.2) OpenStreetMap real dataset representing features of the whole world used in our experiments.

Label	Description	# of objects	File size
Lakes	Boundaries of all water areas in the world	8.4 Million	9 GB
Parks	Boundaries of all parks or green areas in the world	10 Million	9.3 GB

6.5.2 Experimental setup

We perform our experiments on Cheetah, a heterogeneous Linux cluster with several compute nodes each equipped with a multicore CPU and up to four GPUs. Each compute node has a multicore CPU of either Intel Xeon or AMD Opteron (various number of cores) and $64GB$ of main memory. Nodes are connected through high-speed InfiniBand and gigabit Ethernet switches. Table 6.3 describes the four compute nodes used in our setup.

Table (6.3) System information of Cheetah cluster used in our experiments.

Node	CPU	GPU
1	16-core Intel Xeon, 2.1GHz	4 × GTX 770, 2GB
2	16-core Intel Xeon, 2.1GHz	4 × Tesla K20, 4.5GB
3	16-core Intel Xeon, 2.1GHz	1 × GTX Titan, 12GB
4	16-core Intel Xeon, 2.1GHz	1 × GTX Titan, 6GB

As summarized in Table 6.3, the setup contains heterogeneous compute nodes including 10 GPUs with different computational and memory capabilities as well as 64 cpu-cores. We use 64 MPI ranks (rank 0 to 63) including 10 GPU-based processes (ranks 0-3, 16-19, 32 and 48) and 54 CPU-based processes (other ranks).

6.5.3 Results

To evaluate our heuristics, we use two parameters as follows:

1. The total processing time of MPI-cuda-GIS explained in Equation 6.8.
2. We also consider the variance of running time of all the processes (stated in Equation 6.10). While processing time, cannot distinguish between the quality of two load balancing algorithms whose maximum running times are equal, the variance measures how far the processing times are from average. The smaller variance the more even workload distribution.

$$\sigma^2(t_i) = \frac{\sum_{i=1}^{N}(t_i - t_{avg})^2}{N} \tag{6.10}$$

6.5.3.1 Data partitioning We apply uniform grid technique over the whole world data with more than $65,000$ equal-size cells for the initial data partitioning within each process. This technique performs the best when the data is uniformly distributed over the whole space. However, real datasets do not guarantee uniformity. The Figure 6.3 illustrates this problem by showing the histogram of *active* cells in various ranges multiple of 10. The plot shows that more than 80% of the cells are not *active* cells in **Lake-Park** dataset pair and only less than $11,000$ cells need further processing. As shown, the number of objects within *active* cells ranges from 1 to $1000,000$ that verifies uneven distribution of objects in cells.

6.5.3.2 Evaluation of load-balancing heuristics To compare the performance of two proposed load balancing algorithms as well as Round robin technique (RR), Proportional-Cell algorithm (PC) and Proportional-Load algorithm (PL), we incorporate them into MPI-cuda-GIS system to process **Lake-Park**.

Round-robin heuristic (RR)

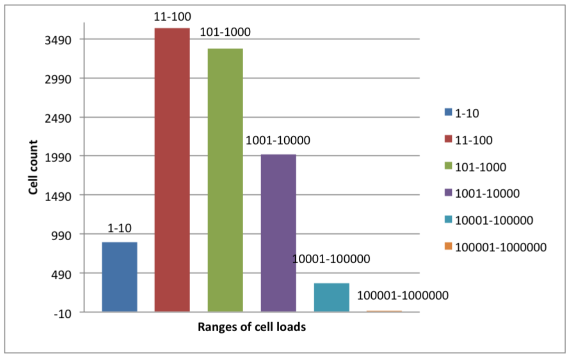


Figure (6.3) Histogram of load of the cells in ranges multiple of 10 for **Lake-Park** dataset pair. The load ranges from 1 to a $1,000,000$ that shows non-uniformity of dataset in the whole space.

Figure 6.4 shows the running time for processing the load distributed by the simplest heuristic. Although cells are evenly distributed, the processing times are variant from less than a second to more than 40 seconds because 1) compute nodes have different compute capabilities, and, 2) cells are not evenly-loaded.

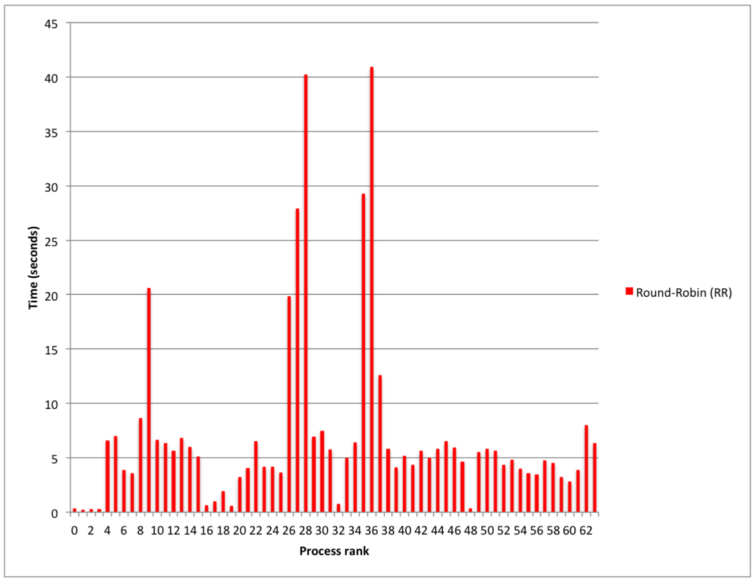


Figure (6.4) The processing time of 64 heterogeneous (CPU/GPU) processes by applying round-robin (RR) load balancing technique over **Lake-Park** dataset pair.

Proportional-Cell heuristic (PC)

As shown in Figure 6.5, by applying PC technique the processing time of ranks ranges from less than a second to more than 37 seconds. While the plot shows a better distribution of the load rather than round robin technique, because of existence of outliers, the total running time (maximum time) is still high. This is certainly because non-uniformity of objects over different cells that is shown in Figure 6.3.

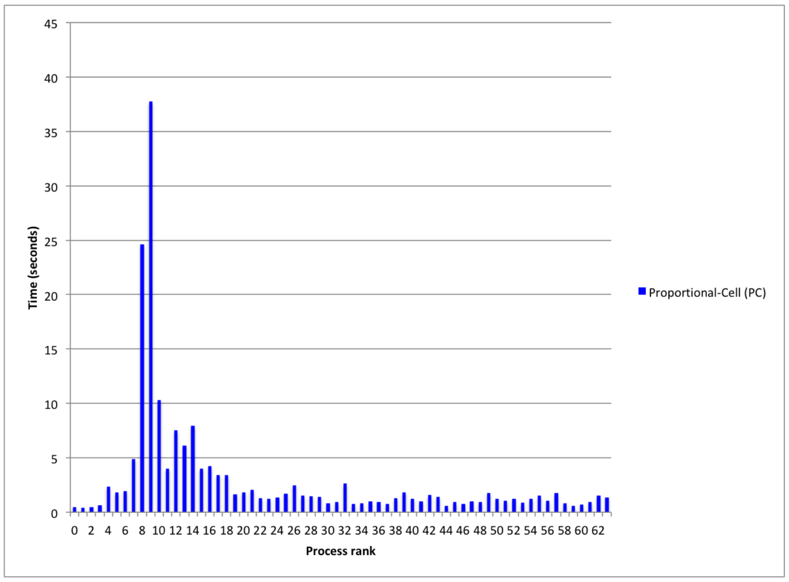


Figure (6.5) The processing time of 64 heterogeneous (CPU/GPU) processes by applying proportional-cell (PC) load balancing technique over **Lake-Park** dataset pair.

Proportional-Load heuristic (PL)

By applying PL, the processing times change in a narrow range of less than a second to 3 seconds as shown in Figure 6.6. This heuristic takes heterogeneity of compute nodes and non-uniformity of objects distributed over the space.

RR, PC and PL comparison

Finally, to compare all the three approaches, we plot them together in Figure 6.7. PL algorithm significantly outperforms the other two techniques.

Table 6.4 provides more statistics for comparison. The total processing time of the system is equal to maximum time of all the ranks. PL is remarkably faster than the other two algorithms. Although both RR and PC have large running times, the variance of PC is less than the half of RR that shows PC does a much better job in distributing the load.

6.5.3.3 System running time analysis Table 6.5 shows the running time of the key components in MPI-cuda-GIS system over **Lake-Park** dataset. As shown, load balanc-

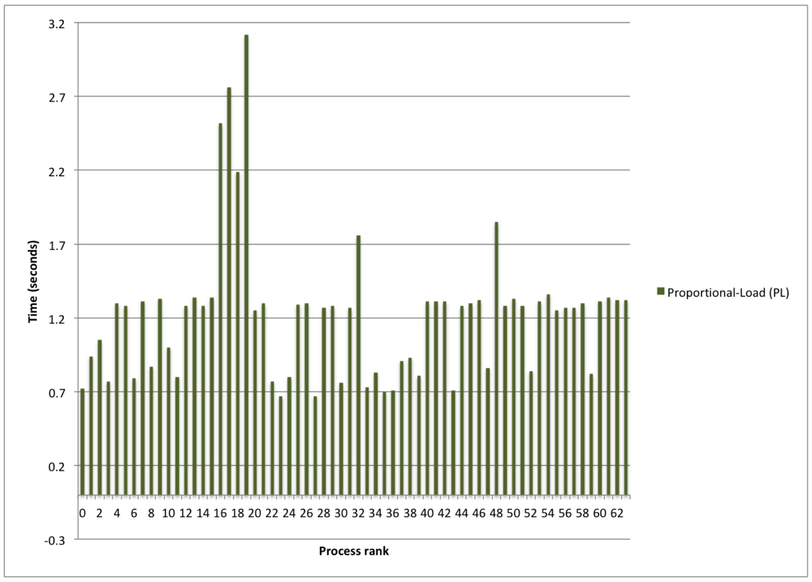


Figure (6.6) The processing time of 64 heterogeneous (CPU/GPU) processes by applying proportional-load (PL) load balancing technique over **Lake-Park** dataset pair.

Table (6.4) Average and maximum processing time , range and variance of time of ranks in MPI-cuda-GIS system by applying three different heuristics.

Heuristic	Processing time			
	Avgerage	Maximum	Range	Variance
RR	7 sec	41 sec	41.7 sec	66.7
PC	2.9 sec	37.8 sec	37.3 sec	31.2
PL	1.2 sec	3.1 sec	2.5 sec	0.2

ing techniques have a small overhead and no matter what algorithm is used, the exchange time is in the almost the same. In the era of high-speed network technologies the communication over the networks is no longer the main bottleneck for extensive computations. This is the reason why we did not consider the communication overhead in our heuristic algorithms. Finally, while the results verify the efficiency of parallel MPI-IO library, the data read by it is in text format and needs to be converted to proper data type. e.g. float, for further processing. The parsing phase mostly involves this conversion and has a significant overhead.

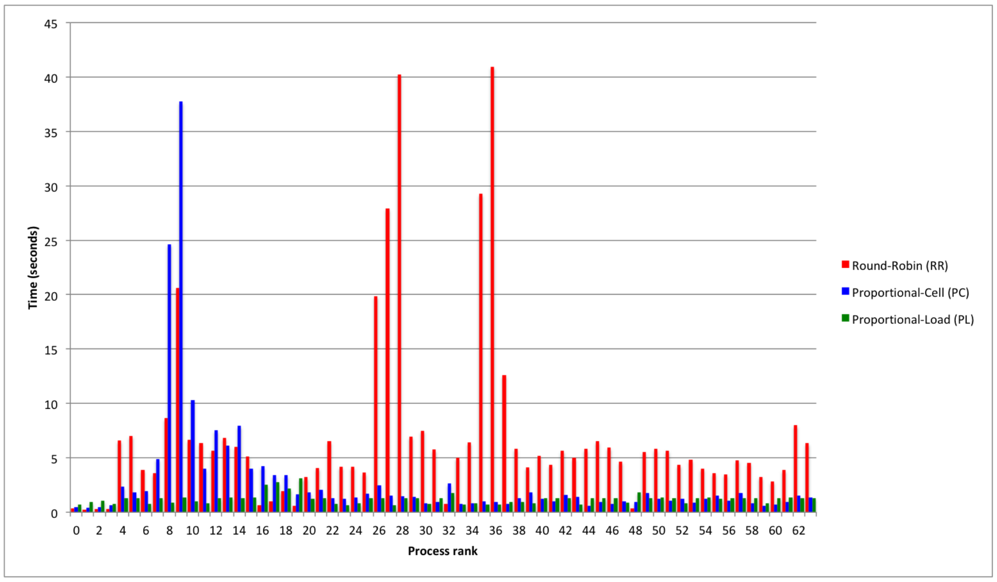


Figure (6.7) The processing time of 64 heterogeneous (CPU/GPU) processes by applying three different heuristics (H1, H2, H3) over **Lake-Park** dataset pair.

Table (6.5) The running time (seconds) of key components in MPI-cuda-GIS system.

Heuristic	IO	Parsing	Partitioning	Gathering in master	Load balancing	Exchange	Main processing	Total
RR	1.5	12.3	0.1	0.3	0.1	0.5	41	55.8
PC	1.5	12.3	0.1	0.3	0.1	0.5	37.8	52.6
PL	1.5	12.3	0.1	0.3	0.1	0.5	3.1	17.9

6.6 Conclusion

In this chapter, we formulate the load balancing and processing of spatial data over heterogeneous computing clusters equipped with CPUs and GPUs as a general ILP problem. Then, we provided two heuristic, Proportional-Cell (PC) and Proportional-Load (PL) algorithms to effectively approximate the ILP model. We showed that both algorithms distribute the load more evenly rather than Round-Robin (RR) technique for unevenly distributed datasets. The experimental results revealed that PL significantly outperforms PC and RR in load balancing by reducing the workload processing time from more than 40 seconds to 3 seconds. Finally, we proposed MPI-cuda-GIS system, a heterogeneous distributed framework built upon MPI and cuda languages to integrate CPU and GPU computing to achieve faster and more scalable HPC system compared to big cluster of computing nodes equipped with multicore CPUs.

Chapter 7

CONCLUSION AND FUTURE WORK

In this chapter, first we summarize our research and then, point out the aspects that we did not cover or explored a little. We categorize these directions as the future work for our next steps.

7.1 Summary

Generally speaking, this dissertation addressed two general classes of problems applicable to spatial domains such as GIS involved with spatial join processing as follows: 1)Theoretical and algorithmic aspects of spatial join problem, and, 2) HPC system design. In the following subsections, we summarize each part.

7.1.1 Theoretical and algorithmic aspects

The main contribution of this part were 1) developing intuitive space reduction linear filters (*CMF*, *CMF-Grid*) suitable to many-core computing such as GPU architectures for spatial join processing, and, 2) proposing load balancing algorithms (PC, PL) for spatial data processing.

First, we introduced *CMF*, a linear filter that effectively eliminates workload of refinement phase by at least an order of magnitude. We provided mathematical analysis to prove correctness of all the algorithms. Then, we extended *CMF* to *CMF-Grid* , a new non-uniform grid technique that is designed with consideration of many-core GPU architectures. We provided comprehensive experimental results based on real datasets to analyze the impact of grid-cell properties such as grid-cell size and grid-cell shape on workload and running time. We showed that to achieve the highest running time we need a grid scheme with 4 to 6 times larger grid-cells than a workload-optimized scheme. Finally, we presented an example of spatial join problem in colocation mining. Our proposed GPU optimizations showed a significant improvement over recently introduced grid-based algorithm.

Second, we proposed an ILP formulation for heterogeneous computing model. The objective of ILP was to optimize load balancing scheme such that it minimizes the total processing time. We provided two heuristics to approximate ILP: 1) Proportional-Cell algorithm (PC), and, 2) Proportional-Load algorithm (PL). Our experiments over OpenStreetMap real datasets showed a great approximation by PL algorithm even for unevenly distributed data.

7.1.2 HPC system design

In this part, first, we proposed *GCMF*, an end-to-end spatial join system (ST_intersect operation) for non-indexed polygonal data over a single GPU platform. The system included 4 subsystems: two filtering components as well as *point-in-polygon* test and *edge-intersection* test subsystems. We proposed sort-based MBR filtering algorithm for GPU with linear average time complexity. The experiments over real data yielded up to 39-fold speedup over optimized sequential *GEOS* library and *Postgres* with *PostGIS* spatial database system. We also showed that replacing *CMF* by *CMF-Grid* in *GCMF* system, its running time improves 225% by reducing the workload by more than two orders of magnitude factor.

Second, we developed *MPI-cuda-GIS*, a HPC computing system for distributed processing of spatial data using heterogeneous compute nodes (CPU + GPU). We designed a distributed framework using MPI language. We integrated *GCMF* system into the system to take advantage of GPU as well as CPU computing.

7.2 Future Work

While two aspects spatial data processing (theory and HPC system design) are well studied in this research and we made remarkable contribution to the domain by exploring linear search space reduction filters and developing distributed heterogeneous (CPU+GPU) computing systems, there are still many unexplored tracks in this field. In the last part of this manuscript, we list some of these challenges as our future works. We hope readers find those challenges interesting and try to propose efficient methods to address them.

- Scalability: Our model with PL load balancing algorithm shows a very good scalability. However, in order to process large spatial data ($> 100GB$), we should have appropriate heterogeneous cluster equipped with enough the CPU-GPU resources. Also, it is important to have access to the sate of art GPUs since there are significant improvements in their computational power as well as their memory limitations. for instance, nvidia $P100GPU$ with $Pascal$ architecture has $16GB$ of memory that is almost the same as using all four $K20$ GPUs with Kepler architecture yet not considering the faster Tesla architecture.

- CMBR-based filters: the *CMBR*-based filter can be generalized to many spatial search problem including the one discussed in Chapter 5 regarding colocation mining problem.

- More spatial join operations: In this research, we did not our libraries to handle all the spatial join operations. There are still some important operations such as overlay that need to be addressed in terms of heterogeneous computing.

- Spatiotemporal join problem: One significantly important aspect of spatial data processing is adding the time series to the problem. Problem formulations and data structures are fundamentally different and more complex in spatiotemporal domains yet they are applicable to more applications and domains.

REFERENCES

[1] S. K. Prasad, D. Aghajarian, M. McDermott, D. Shah, M. Mokbel, S. Puri, S. J. Rey, S. Shekhar, Y. Xe, R. R. Vatsavai *et al.*, "Parallel processing over spatial-temporal datasets from geo, bio, climate and social science communities: A research roadmap," in *Big Data (BigData Congress), 2017 IEEE International Congress on.* IEEE, 2017, pp. 232–250.

[2] S. K. Prasad, M. McDermott, S. Puri, D. Shah, D. Aghajarian, S. Shekhar, and X. Zhou, "A vision for GPU-accelerated parallel computation on geo-spatial datasets," *SIGSPATIAL Special*, vol. 6, no. 3, pp. 19–26, 2015.

[3] "NASA earth science data: https://aws.amazon.com/blogs/aws/process-earth-science-data-on-aws-with-nasa-nex/."

[4] S. Puri and S. K. Prasad, "A parallel algorithm for clipping polygons with improved bounds and a distributed overlay processing system using mpi," in *Cluster, Cloud and Grid Computing (CCGrid), 2015 15th IEEE/ACM International Symposium on.* IEEE, 2015, pp. 576–585.

[5] S. You, J. Zhang, and L. Gruenwald, "Large-scale spatial join query processing in cloud," in *Data Engineering Workshops (ICDEW), 2015 31st IEEE International Conference on.* IEEE, 2015, pp. 34–41.

[6] S. Ray, B. Simion, A. D. Brown, and R. Johnson, "A parallel spatial data analysis infrastructure for the cloud," in *Proceedings of the 21st ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems.* ACM, 2013, pp. 284–293.

[7] S. Puri, D. Aghajarian, and S. Prasad, "MPI-GIS : High Performance Computing and IO for Spatial Overlay and Join," *The International Conference for High Performance Computing, Networking, Storage and Analysis (SC-16)*, 2016 (Research Poster).

[8] A. Eldawy and M. F. Mokbel, "A demonstration of spatialhadoop: An efficient mapreduce framework for spatial data," *Proceedings of the VLDB Endowment*, vol. 6, no. 12, pp. 1230–1233, 2013.

[9] S. K. Prasad, M. McDermott, X. He, and S. Puri, "Gpu-based parallel r-tree construction and querying," in *Parallel and Distributed Processing Symposium Workshop (IPDPSW), 2015 IEEE International.* IEEE, 2015, pp. 618–627.

[10] E. H. Jacox and H. Samet, "Spatial join techniques," *ACM Transactions on Database Systems (TODS)*, vol. 32, no. 1, p. 7, 2007.

[11] PostGIS, "http://postgis.net/."

[12] D. H. Jonassen, "Instructional design models for well-structured and iii-structured problem-solving learning outcomes," *Educational Technology Research and Development*, vol. 45, no. 1, pp. 65–94, 1997.

[13] J. F. Voss, "On the solving of iii-structured problems," *The nature of expertise*, p. 261, 2014.

[14] A. Aji, G. Teodoro, and F. Wang, "Haggis: Turbocharge a mapreduce based spatial data warehousing system with gpu engine," in *Proceedings of the 3rd ACM SIGSPATIAL International Workshop on Analytics for Big Geospatial Data.* ACM, 2014, pp. 15–20.

[15] Y. J. García, M. A. Lopez, and S. T. Leutenegger, "On optimal node splitting for r-trees," in *Proceedings of the 24rd International Conference on Very Large Data Bases.* Morgan Kaufmann Publishers Inc., 1998, pp. 334–344.

[16] J. Zhang and S. You, "Cudagis: report on the design and realization of a massive data parallel gis on gpus," in *Proceedings of the Third ACM SIGSPATIAL International Workshop on GeoStreaming.* ACM, 2012, pp. 101–108.

[17] M. Schneider, "Spatial data types for database systems(finite resolution geometry for geographic information systems)," *Lecture notes in computer science*, 1997.

[18] ——, "Spatial plateau algebra for implementing fuzzy spatial objects in databases and gis: Spatial plateau data types and operations," *Applied Soft Computing*, vol. 16, pp. 148–170, 2014.

[19] G. S. Taylor, J. Ousterhout, G. Hamachi, R. Mayo, and W. Scott, "Magic: A vlsi layout system," in *Proceedings of the 21th Design Automation Conference*, 1984, pp. 152–159.

[20] S. Shekhar, S. Chawla, S. Ravada, A. Fetterer, X. Liu, and C.-t. Lu, "Spatial databases-accomplishments and research needs," *IEEE transactions on knowledge and data engineering*, vol. 11, no. 1, pp. 45–55, 1999.

[21] H. Samet, *Foundations of multidimensional and metric data structures.* Morgan Kaufmann, 2006.

[22] H. Zhu, J. Su, and O. H. Ibarra, "On multi-way spatial joins with direction predicates," in *International Symposium on Spatial and Temporal Databases.* Springer, 2001, pp. 217–235.

[23] H. Samet, "Applications of spatial data structures," 1990.

[24] S. Berchtold, D. Keim, and H. Kriegel, "An index structure for high-dimensional data," *Readings in multimedia computing and networking*, vol. 451, 2001.

[25] N. Roussopoulos and D. Leifker, "Direct spatial search on pictorial databases using packed r-trees," in *ACM Sigmod Record*, vol. 14, no. 4. ACM, 1985, pp. 17–31.

[26] J. T. Robinson, "The kdb-tree: a search structure for large multidimensional dynamic indexes," in *Proceedings of the 1981 ACM SIGMOD international conference on Management of data.* ACM, 1981, pp. 10–18.

[27] W. G. Aref and H. Samet, "Efficient processing of window queries in the pyramid data structure," in *Proceedings of the ninth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems.* ACM, 1990, pp. 265–272.

[28] H. Samet and R. E. Webber, "Storing a collection of polygons using quadtrees," *ACM Transactions on Graphics (TOG)*, vol. 4, no. 3, pp. 182–222, 1985.

[29] A. Guttman, *R-trees: a dynamic index structure for spatial searching.* ACM, 1984, vol. 14, no. 2.

[30] D. Comer, "Ubiquitous b-tree," *ACM Computing Surveys (CSUR)*, vol. 11, no. 2, pp. 121–137, 1979.

[31] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger, "The r*-tree: an efficient and robust access method for points and rectangles," in *Acm Sigmod Record*, vol. 19, no. 2. Acm, 1990, pp. 322–331.

[32] T. Sellis, N. Roussopoulos, and C. Faloutsos, "The r+-tree: A dynamic index for multi-dimensional objects." Tech. Rep., 1987.

[33] H. Samet, "The quadtree and related hierarchical data structures," *ACM Computing Surveys (CSUR)*, vol. 16, no. 2, pp. 187–260, 1984.

[34] R. A. Finkel and J. L. Bentley, "Quad trees a data structure for retrieval on composite keys," *Acta informatica*, vol. 4, no. 1, pp. 1–9, 1974.

[35] R. K. V. Kothuri, S. Ravada, and D. Abugov, "Quadtree and r-tree indexes in oracle spatial: a comparison using gis data," in *Proceedings of the 2002 ACM SIGMOD international conference on Management of data.* ACM, 2002, pp. 546–557.

[36] W. R. Franklin, *Combinatorics of hidden surface algorithms.* Center for Research in Computing Techn., Aiken Computation Laboratory, Univ., 1978.

[37] A. Appel, F. J. Rohlf, and A. J. Stein, *The haloed line effect for hidden line elimination.* ACM, 1979, vol. 13, no. 2.

[38] W. R. Franklin, "Efficient polyhedron intersection and union," in *Proc. Graphics Interface*, vol. 82, 1982, pp. 73–80.

[39] W. R. Franklin and P. Y. Wu, "A polygon overlay system in prolog," in *Proceedings, AutoCarto*, vol. 8, 1987, pp. 97–106.

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

[41] W. R. Franklin, N. Chandrasekhar, M. Kankanhalli, M. Seshan, and V. Akman, "Efficiency of uniform grids for intersection detection on serial and parallel machines," in *New Trends in Computer Graphics.* Springer, 1988, pp. 288–297.

[42] C. B. Walton, A. G. Dale, and R. M. Jenevein, "A taxonomy and performance model of data skew effects in parallel joins." in *VLDB*, vol. 91, 1991, pp. 537–548.

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

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

[45] W. R. Franklin, V. Sivaswami, D. Sun, M. Kankanhalli, and C. Narayanaswami, "Calculating the area of overlaid polygons without constructing the overlay," *Cartography and Geographic Information Systems*, vol. 21, no. 2, pp. 81–89, 1994.

[46] L. Arge, O. Procopiuc, S. Ramaswamy, T. Suel, and J. S. Vitter, "Scalable sweeping-based spatial join," in *VLDB*, vol. 98. Citeseer, 1998, pp. 570–581.

[47] S. T. Leutenegger, M. Lopez, J. Edgington *et al.*, "Str: A simple and efficient algorithm for r-tree packing," in *Data Engineering, 1997. Proceedings. 13th International Conference on.* IEEE, 1997, pp. 497–506.

[48] M. Pavlovic, F. Tauheed, T. Heinis, and A. Ailamakit, “Gipsy: joining spatial datasets with contrasting density,” in *Proceedings of the 25th International Conference on Scientific and Statistical Database Management*. ACM, 2013, p. 11.

[49] X. Zhou, D. J. Abel, and D. Truffet, “Data partitioning for parallel spatial join processing,” *Geoinformatica*, vol. 2, no. 2, pp. 175–204, 1998.

[50] S. You, J. Zhang, and L. Gruenwald, “Parallel spatial query processing on gpus using r-trees,” in *Proceedings of the 2nd ACM SIGSPATIAL International Workshop on Analytics for Big Geospatial Data*. ACM, 2013, pp. 23–31.

[51] D. Aghajarian, S. Puri, and S. K. Prasad, “GCMF: An efficient end-to-end spatial join system over large polygonal datasets on gpgpu platform,” *SIGSPATIAL*, 2016.

[52] D. Aghajarian and S. K. Prasad, “A spatial join algorithm based on a non-uniform grid technique over gpgpu,” in *Proceedings of the 25th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*. ACM, 2017, p. 56.

[53] T. Yampaka and P. Chongstitvatana, “Spatial join with r-tree on graphics processing units,” *KMUTNB: International Journal of Applied Science and Technology*, vol. 5, no. 3, pp. 1–7, 2013.

[54] L. Luo, M. D. Wong, and L. Leong, “Parallel implementation of r-trees on the gpu,” in *Design Automation Conference (ASP-DAC), 2012 17th Asia and South Pacific*. IEEE, 2012, pp. 353–358.

[55] S. Puri and S. K. Prasad, “MPI-GIS: New parallel overlay algorithm and system prototype,” 2014.

[56] S. Puri, “Efficient parallel and distributed algorithms for gis polygon overlay processing,” 2015.

[57] S. Puri, D. Agarwal, and S. K. Prasad, “Polygonal overlay computation on cloud, hadoop, and mpi,” *Encyclopedia of GIS*, pp. 1–9, 2015.

[58] A. Eldawy and M. F. Mokbel, “Spatialhadoop: A mapreduce framework for spatial data,” in *Data Engineering (ICDE), 2015 IEEE 31st International Conference on*. IEEE, 2015, pp. 1352–1363.

[59] F. Baig, M. Mehrotra, H. Vo, F. Wang, J. Saltz, and T. Kurc, “Sparkgis: Efficient comparison and evaluation of algorithm results in tissue image analysis studies,” in *Biomedical Data Management and Graph Online Querying*. Springer, 2015, pp. 134–146.

[60] F. Baig, H. Vo, T. Kurc, J. Saltz, and F. Wang, “Sparkgis: Resource aware efficient in-memory spatial query processing,” in *Proceedings of the 25th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*. ACM, 2017, p. 28.

[61] D. Agarwal, S. Puri, X. He, S. K. Prasad, and X. Shi, “Crayons-a cloud based parallel framework for gis overlay operations,” *Distributed & Mobile Systems Lab*, 2011.

[62] D. Agarwal, S. Puri, X. He, and S. K. Prasad, “A system for gis polygonal overlay computation on linux cluster-an experience and performance report,” in *Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2012 IEEE 26th International*. IEEE, 2012, pp. 1433–1439.

[63] D. Agarwal, S. Puri, and S. K. Prasad, “Crayons: Empowering cybergis by employing cloud infrastructure,” in *CyberGIS for Geospatial Discovery and Innovation*. Springer, 2019, pp. 115–141.

[64] M. Stonebraker, J. Frew, K. Gardels, and J. Meredith, “The sequoia 2000 storage benchmark,” in *ACM SIGMOD Record*, vol. 22, no. 2. ACM, 1993, pp. 2–11.

[65] B. Simion, S. Ray, and A. D. Brown, “Speeding up spatial database query execution using gpus,” *Procedia Computer Science*, vol. 9, pp. 1870–1879, 2012.

[66] J. Zhang, S. You, and L. Gruenwald, "High-performance spatial query processing on big taxi trip data using gpgpus," in *Big Data (BigData Congress), 2014 IEEE International Congress on.* IEEE, 2014, pp. 72–79.

[67] S. You, J. Zhang, and L. Gruenwald, "Scalable and efficient spatial data management on multi-core cpu and gpu clusters: A preliminary implementation based on impala," in *Data Engineering Workshops (ICDEW), 2015 31st IEEE International Conference on.* IEEE, 2015, pp. 143–148.

[68] A. N. Tantawi and D. Towsley, "Optimal static load balancing in distributed computer systems," *Journal of the ACM (JACM)*, vol. 32, no. 2, pp. 445–465, 1985.

[69] A. Legrand, H. Renard, Y. Robert, and F. Vivien, "Mapping and load-balancing iterative computations," *IEEE Transactions on Parallel and Distributed Systems*, vol. 15, no. 6, pp. 546–558, 2004.

[70] G. Cybenko, "Dynamic load balancing for distributed memory multiprocessors," *Journal of parallel and distributed computing*, vol. 7, no. 2, pp. 279–301, 1989.

[71] G.-w. You, S.-w. Hwang, and N. Jain, "Scalable load balancing in cluster storage systems," in *Proceedings of the 12th International Middleware Conference.* International Federation for Information Processing, 2011, pp. 100–119.

[72] H. J. Moon, H. Hacıgümüş, Y. Chi, and W.-P. Hsiung, "Swat: a lightweight load balancing method for multitenant databases," in *Proceedings of the 16th International Conference on Extending Database Technology.* ACM, 2013, pp. 65–76.

[73] M. D. Lieberman, J. Sankaranarayanan, and H. Samet, "A fast similarity join algorithm using graphics processing units," in *Data Engineering, 2008. ICDE 2008. IEEE 24th International Conference on.* IEEE, 2008, pp. 1111–1120.

[74] M. McKenney and T. McGuire, "A parallel plane sweep algorithm for multi-core sys-

tems," in *Proceedings of the 17th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems.* ACM, 2009, pp. 392–395.

[75] M. Shimrat, "Algorithm 112: position of point relative to polygon," *Communications of the ACM*, vol. 5, no. 8, p. 434, 1962.

[76] J. Zhang, S. You, and L. Gruenwald, "High-performance online spatial and temporal aggregations on multi-core cpus and many-core gpus," in *Proceedings of the fifteenth international workshop on Data warehousing and OLAP.* ACM, 2012, pp. 89–96.

[77] M. Aigner and G. M. Ziegler, *Proofs from the Book.* Springer, 2010, vol. 274.

[78] "XSEDE resource in Pittsburgh Supercomputing Center: https://www.psc.edu/homepage/about-psc."

[79] Y. Huang, S. Shekhar, and H. Xiong, "Discovering colocation patterns from spatial data sets: a general approach," *IEEE Transactions on Knowledge and data engineering*, vol. 16, no. 12, pp. 1472–1485, 2004.

[80] P. Phillips and I. Lee, "Crime analysis through spatial areal aggregated density patterns," *Geoinformatica*, vol. 15, no. 1, pp. 49–74, 2011.

[81] S. E. Donovan, G. J. Griffiths, R. Homathevi, and L. Winder, "The spatial pattern of soil-dwelling termites in primary and logged forest in sabah, malaysia," *Ecological Entomology*, vol. 32, no. 1, pp. 1–10, 2007.

[82] P. Haase, "Spatial pattern analysis in ecology based on ripley's k-function: Introduction and methods of edge correction," *Journal of vegetation science*, vol. 6, no. 4, pp. 575–582, 1995.

[83] A. Sadilek, H. A. Kautz, and V. Silenzio, "Predicting disease transmission from geo-tagged micro-blog data." in *AAAI*, 2012.

[84] G. M. Vazquez-Prokopec, D. Bisanzio, S. T. Stoddard, V. Paz-Soldan, A. C. Morrison, J. P. Elder, J. Ramirez-Paredes, E. S. Halsey, T. J. Kochel, T. W. Scott *et al.*, "Using gps technology to quantify human mobility, dynamic contacts and infectious disease dynamics in a resource-poor urban environment," *PloS one*, vol. 8, no. 4, p. e58802, 2013.

[85] L.-A. Tang, Y. Zheng, J. Yuan, J. Han, A. Leung, W.-C. Peng, and T. L. Porta, "A framework of traveling companion discovery on trajectory data streams," *ACM Transactions on Intelligent Systems and Technology (TIST)*, vol. 5, no. 1, p. 3, 2013.

[86] K. Koperski and J. Han, "Discovery of spatial association rules in geographic information databases," in *International Symposium on Spatial Databases.* Springer, 1995, pp. 47–66.

[87] Y. Morimoto, "Mining frequent neighboring class sets in spatial databases," in *Proceedings of the seventh ACM SIGKDD international conference on Knowledge discovery and data mining.* ACM, 2001, pp. 353–358.

[88] A. M. Sainju and Z. Jiang, "Grid-based colocation mining algorithms on gpu for big spatial event data: A summary of results," in *International Symposium on Spatial and Temporal Databases.* Springer, 2017, pp. 263–280.

[89] P. NEUBAUER, "Neo4j spatial-gis for the rest of us," 2012.

[90] T. Płuciennik and E. Płuciennik-Psota, "Using graph database in spatial data generation," in *Man-Machine Interactions 3.* Springer, 2014, pp. 643–650.

[91] P. Corbett, D. Feitelson, S. Fineberg, Y. Hsu, B. Nitzberg, J.-P. Prost, M. Snirt, B. Traversat, and P. Wong, "Overview of the mpi-io parallel i/o interface," in *Input/Output in Parallel and Distributed Computer Systems.* Springer, 1996, pp. 127–146.

[92] W. Yu, S. Oral, J. Vetter, and R. Barrett, "Efficiency evaluation of cray xt parallel io stack," in *Cray User Group Meeting (CUG 2007)*, 2007.

[93] N. Deo and S. Prasad, “Parallel heap,” in *ICPP (3)*, 1990, pp. 169–172.

[94] ——, “Parallel heap: An optimal parallel priority queue,” *The Journal of Supercomputing*, vol. 6, no. 1, pp. 87–98, 1992.

[95] S. K. Prasad and S. I. Sawant, “Parallel heap: A practical priority queue for fine-to-medium-grained applications on small multiprocessors,” in *Parallel and Distributed Processing, 1995. Proceedings. Seventh IEEE Symposium on.* IEEE, 1995, pp. 328–335.

[96] X. He, D. Agarwal, and S. K. Prasad, “Design and implementation of a parallel priority queue on many-core architectures,” in *High Performance Computing (HiPC), 2012 19th International Conference on.* IEEE, 2012, pp. 1–10.

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